MAKINA CORPUS

PostgreSQL9

Développement et Administration par la pratique

Version 1.0 Juin 2012

Regis Leroy

1. Sommaire

2. Makina Corpus

Makina Corpus conçoit, développe et intègre des solutions innovantes s'appuyant exclusivement sur des logiciels libres.

La société a développé une expertise particulière en applications web complexes, dans le domaine des portails, le traitement de données géographiques (SIG) et l'analyse décisionnelle (Business Intelligence).

Makina Corpus intervient sur :

- une expertise technologique de haut niveau ;
- l'intégration d'applications hétérogènes ;
- une offre sur mesure, conçue et développée en interne pour répondre spécifiquement à vos besoins ;
- une réduction du coût d'acquisition logiciel, grâce à l'utilisation de logiciels du monde libre ;
- un service complet clés en main.

2.1. Nos domaines de compétences

Makina Corpus propose des solutions innovantes pour :

- la gestion de contenus (CMS, intranet, extranet, internet) ;
- la géomatique ou SIG (webmapping et clients lourds) ;
- les outils décisionnels (traitement de données, analyses, reporting).

2.2. Notre philosophie

Makina Corpus croit fermement aux valeurs d'ouverture et de partage du logiciel libre et s'implique comme ardent promoteur et contributeur de la communauté du libre.

La valeur ajoutée est forte et concrète pour nos clients : les logiciels libres leur garantissent en effet une totale indépendance par rapport à leurs fournisseurs, réduisent les coûts, et assurent une grande pérennité.

2.3. L'équipe projet et technique : moyens humains

Notre équipe est faite d'hommes et de femmes ayant chacun une individualité. L'entreprise reconnaît et valorise la diversité de tous ces talents, et en exploite au mieux les complémentarités. Dans cette démarche de respect des choix individuels, Makina Corpus a choisi d'être présente sur de nombreuses implantations géographiques et d'accepter le télé-travail.

Pour une nouvelle approche de la relation client-fournisseur, Makina Corpus met en œuvre la méthode agile pragmatique et pro-active **SCRUM**. Les échanges fréquents et programmés entre le client et l'équipe de développement permettent d'identifier et corriger au plus tôt d'éventuels problèmes et surtout de répondre au mieux aux **besoins réels du client**.

2.4. Nos clients

Makina Corpus a réalisé des prestations web pour de nombreuses collectivités territoriales :

Nantes, Angers, Cannes,Cap Atlantique, Lille , Sevran, Challans, La Montagne, Aigrefeuille... Pays des Vals de Saintonge, Arrondissement de Briey, Pays-Haut - Val d'Alzette, Pays du Gard Rhodanien. Conseils Généraux : Vaucluse, Bas-Rhin, Essonne. Conseils Régionaux : Aquitaine, Picardie, Communauté francophone (Belgique)

Dans le domaine des portails web et des intranets, Makina Corpus est également intervenue auprès :

- du Ministère des Affaires Étrangères, de la Préfecture d'Île et Vilaine, de l'Aéroport Nantes-Atlantique et de l'Association des Aéroports Francophones, des Chambres d'agriculture, et de la SAFER de Haute Normandie, de l'Établissement Public Territorial du Fleuve Charente, et de l'Association des Établissements Publics Territoriaux de Bassin, de l'Union Sociale de l'Habitat, de l'INSEE, de l'Agence Nationale des Fréquences, de l'OTAN et des Nations Unies, de la Commission Européenne....

3. Licence

Vous pouvez contribuer à ce document nous signalant les erreurs, en apportant vos remarques, ajouts et commentaires. Ceux—ci pourront être intégrés au documents, avec leur paternité.Il vous est permis d'effectuer quelques adaptations détaillées au chapitre « Réutilisation du contenu de ce support de formation ».

Les captures d'écran, les codes sources SQL et PHP sont normalement disponibles avec les sources.

La licence couvre les documents pdf, bureautique (odt), les captures d'écrans, les sources SQL et les dumps de bases dans divers formats fournis avec ce présent document. Les sources PHP et bsh sont quand à elles diffusées sous licence BSD modifié (Simplified BSD licence ou BSD-2).

Les fichiers sources sont disponibles sur https://github.com/regilero/PostgreSQL-formation-pratique

Retrouvez les livres blancs Makina Corpus et les contributions diverses sur http://makina-corpus.com et http://makina-corpus.org.

4. Réutilisation du contenu de ce support de formation

Ce support de formation est soumis à la licence détaillée dans le chapitre 3.Licence.

Cependant, afin de pouvoir réutiliser ce support de formation dans vos démarches commerciales et afin de pouvoir modeler le plan de cours en fonction des attentes des apprenants , nous autorisons sans republication des modifications :

- la suppression du chapitre 2.Makina Corpus et du chapitre 4.Réutilisation du contenu de ce support de formation
- l'intégration de l'intégralité du support de formation(hormis le chapitre 2.Makina Corpus et 4.Réutilisation du contenu de ce support de formation) dans un modèle de document différent de celui de Makina Corpus.
- La suppression de chapitres complets (comme les derniers chapitres dédiés aux aspects administration PostgreSQL). Il est par contre nécessaire d'indiquer le retrait de ces chapitres en listant dans ce support les titres des chapitres retirés du support original.

Toute autre modification substantielle ou contribution à ce support devrait être accompagnée d'une publication de ces modifications ou de l'intégralité du support sur un support librement disponible à tous (dépôt de sources publiques type github par exemple). Afin de mutualiser les efforts nous vous invitons bien sûr à contribuer à l'oeuvre originale dans la mesure du possible.

Il est interdit de retirer les références aux auteurs et à Makina Corpus du chapitre 3.Licence.

5. Auteur

Régis Leroy (aka regilero)<regis.leroy@makina-corpus.com>

Administrateur Système, développeur web, architecte web, formateur sur des sujets divers (supervision libre, bases de données libres, architectures web complexes) depuis plusieurs années. Guidé par le besoin de disposer d'un support de formation complet sur PostgreSQL il s'est naturellement dirigé vers l'écriture et la publication d'un tel support, afin que d'autres puisse en profiter et contribuer ensemble à améliorer la qualité des supports de formation PostgreSQL.

6. Organisation de ce support de formation

Ce support de formation est très orienté sur les aspects pratiques.

Dans le monde PostgreSQL les aspects théoriques sont très bien documentés. La documentation en ligne de PostgreSQL est sans doute un modèle du genre. Chaque version de PostgreSQL possède sa propre version de la documentation, dans plusieurs langues, et quand Google vous amène sur une page de cette documentation il suffit le plus souvent de changer le numéro de version dans l'url pour obtenir la même page mais adaptée à votre version. Par exemple:

- http://docs.postgresql.fr/9.0/INSTALL.html#install-getsource
- http://docs.postgresql.fr/8.4/INSTALL.html#install-getsource
- http://docs.postgresqlfr.org/9.0/plpgsql-expressions.html

- http://docs.postgresqlfr.org/8.2/plpgsql-expressions.html

Le point d'entrée en français pour la documentation est: http://docs.postgresqlfr.org/

Il est possible de télécharger des versions PDF de la documentation ou de la consulter en ligne.

Cette documentation couvre des aspects très larges et très détaillé, mais aussi des procédures et des articles généraux. On trouvera aussi des articles intéressants sur le wiki PostgreSQL, comme http://wiki.postgresql.org/wiki/Working_with_Dates_and_Times_in_PostgreSQL

Les possibilités techniques et fonctionnelles de PostgreSQL sont très étendues, il serait donc inutile d'essayer de toutes les lister et de toutes les expérimenter. Les documents d'explication en français existent et sont très complets. Nous choisissons donc d'orienter la formation sur des retours d'expériences réelles, sur de l'expérimentation, afin d'ouvrir les débats et de poser les principales questions.

Nous utiliserons donc des installations packagées de PostgreSQL, nous testerons la ligne de commande `psql` mais nous utiliserons principalement le client d'administration graphique pgAdmin. Nous travaillerons sur des bases d'exemples dont les fichiers SQL doivent être fournis avec le support de formation.

7. Installer PostgreSQL

Pour installer PostgreSQL, le point de départ est http://www.postgresql.org/download/, il y a principalement trois solutions:

- compiler sois-même un serveur PostgreSQL. Les sources sont disponibles sur le site http://www.postgresql.org/ftp/source/ . De nombreuses documentations sont disponibles sur Internet. Il s'agit d'une méthode ardue mais qui peut permettre d'obtenir des performances meilleures pour un matériel spécifique, à condition cependant d'être un expert en matière de compilation, ce qui ne peut pas raisonnablement être présenté dans le cadre d'une formation.
- Utiliser les assistants d'installation fournis par EnterpriseDB, nommés « Postgres Plus » http://www.enterprisedb.com/products-services-training/postgres-plus/download. Il s'agit sans doute de la meilleure option pour une installation su un serveur Windows. On obtient un serveur PostgreSQL en 32 ou 64 bits, fournit avec des modules de contributions utiles et le client lourd d'administration pgAdmin III.
- Sur une distribution Linux la meilleure solution est d'utiliser les packages de la distribution. Les Distribution de type Debian (comme Ubuntu) possèdent des packages très bien organisés qui permettent de faire tourner plusieurs instances de PostgreSQL en fonction des versions. On pourra trouver des « ppa », sites de maintient de versions plus avancées des packages PostgreSQL si la distribution utilisée ne contient pas la version voulue de PostgreSQL.

Dans le cadre de cette formation nous utiliserons une distribution Linux capable d'installer un serveur PostgreSQL de version 9.0 ou supérieure. La dernière distribution Ubuntu par exemple.

Première étape passons root sur le système:

```
> sudo su -
```

Pour trouver les packages disponibles d'une distribution en dehors des outils graphiques de type synaptic ou semi-graphiques comme aptitude), il est possible de taper cette commande:

```
# apt-cache search --names-only postgresql
```

Puis une fois qu'on a reperé le package voulu il suffit de taper la commande `apt-get install` et d'installer les dépendances affichées. On pourra y ajouter les paquets suggérés.

```
# apt-get install postgresql-9.0
```

ou

```
# apt-get install postgresql-9.1
```

7.1. Arrêt démarrage et initialisation

Le serveur est alors installé. Il tourne peut-être déjà. Pour contrôler l'arrêt et le démarrage du démon des scripts ont été normalement installés dans le système. Ces scripts sont une abstraction des vraies commandes d'arrêt et démarrage qui utilisent le programme pg_ctl ou pg_ctlcluster (comme « PostgreSQL control »). Ils permettent de ne pas avoir à taper de longues lignes de commandes indiquant les répertoires et fichiers de configuration. Ces scripts sont normalement inclus dans les « niveaux de démarrage » de l'OS ce qui fais qu'un arrêt ou un démarrage du serveur est capable d'ordonner l'arrêt et le démarrage du démon PostgreSQL. Il est bien sûr possible de le faire sois-même à la main, soit en étant l'utilisateur **root**, soit en étant l'utilisateur **postgres**.

Un utilisateur postgres a en effet été créé, que ce soit sur Linux ou Windows, c'est l'utilisateur 'système' chargé de faire tourner le service.

La commande de démarrage d'un démon sous Linux est classiquement:

```
/etc/init.d/nom-du-service start
```

Et la commande d'arrêt

```
/etc/init.d/nom-du-service stop
```

Sur une distribution de type Debian si vous commencez à taper « `/etc/init.d/` » puis vous tapez la touche tabulation plusieurs fois, des propositions d'auto-complétion vous seront proposées. Vous pouvez-voir que le service PostgreSQL est suffixé avec son numéro de version majeur, ce qui permet de piloter indépendamment plusieurs versions majeures sur le système. Essayez alors ces diverses commandes:

```
/etc/init.d/postgresql-9.0
/etc/init.d/postgresql-9.0 stop
/etc/init.d/postgresql-9.0 start
```

ou

```
/etc/init.d/postgresql
/etc/init.d/postgresql stop
/etc/init.d/postgresql start
```

Suivant les distributions il est possible que le service refuse de démarrer dans un premier temps (sur les distributions de type red hat par exemple) en indiquant que l'initdb doit être effectué. Cette phase d'initialisation se fais ainsi:

```
/etc/init.d/postgresql-9.0 initdb
```

L'**initdb** constitue le processus initial d'un serveur PostgreSQL. Lors de ce processus le répertoire de stockage du cluster est créé, avec ses principaux dossiers et ses fichiers requis.

Il n'est pas nécessaire par définition d'effectuer cette commande plusieurs fois, et c'est pourquoi certaines distributions incluent cette phase dans l'installation du package.

Si vous changez le répertoire dans le lequel PostgreSQL doit travailler vous devrez initialiser ce répertoire avec la commande `initdb`.

Pour vérifier que le démon tourne nous allons regarder si les processus PostgreSQL existent. Sur Windows cela se ferait avec le gestionnaire des tâches. Sur Linux nous pouvons utiliser cette commande:

```
ps auxf
```

Que nous allons filtrer pour ne conserver que les lignes contenant le mot « postgres »

```
ps auxf|grep postgres
```

Sur une installation à partir d'un package extérieur à la distribution j'obtiens ceci (vous devriez obtenir quasiment la même chose):

```
postgres  1995  0.0  0.1  44300  4160 ?  S    09:12   0:00
/opt/PostgreSQL/9.0/bin/postgres -D /opt/PostgreSQL/9.0/data
postgres  2086  0.0  0.0  12328   872 ?  Ss   09:12   0:00  \_
postgres: logger process
postgres  2091  0.0  0.0  44300  1280 ?  Ss   09:12   0:00  \_
postgres: writer process
postgres  2092  0.0  0.0  44300  1064 ?  Ss   09:12   0:00  \_
postgres: wal writer process
postgres  2093  0.0  0.0  45108  2168 ?  Ss   09:12   0:00  \_
postgres: autovacuum launcher process
postgres  2094  0.0  0.0  12812  1340 ?  Ss   09:12   0:00  \_
postgres: stats collector process
postgres  6586  0.0  0.1  45492  5576 ?  Ss   09:32   0:00  \_
postgres: postgres postgres 127.0.0.1(41496) idle
```

Les lignes sont un peu longues, si je simplifie la sortie nous avons en fait:

```
postgres  1995  /bin/postgres -D /ici/chemin/data/directory
postgres  2086    \_ postgres: logger process
postgres  2091    \_ postgres: writer process
postgres  2092    \_ postgres: wal writer process
postgres  2093    \_ postgres: autovacuum launcher process
postgres  2094    \_ postgres: stats collector process
postgres  6586    \_ postgres: postgres postgres 127.0.0.1(41496) idle
```

Nous voyons donc un processus principal: ici le PID (Processus ID) 1995 tournant avec l'utilisateur postgres. Puis nous avons déjà 6 processus fils par défaut, chacun avec un métier qui lui est propre, et sur lesquels nous reviendrons.

Si des utilisateurs avaient des connexions ouvertes vers le service PostgreSQL nous aurions un nouveau processus fils par connexion, c'est le système des **forks de processus** souvent utilisé dans le monde Unix. Il ne s'agit donc pas ici d'une gestion multithreadée et ceci est la principale raison pour laquelle vous obtiendrez toujours des performances inférieures de PostgreSQL sur un système Windows. PostgreSQL est pensé comme un démon Unix et le portage sur Windows souffre de la mauvaise gestion des forks sur Windows. Il ne s'agit pas de limitations du système Windows, les services Windows utilisent beaucoup plus souvent la gestion multithreadé qui est très performante sur ce système, il s'agit simplement d'une différence de concepts. Il n'existe pas de version multi-threadé pour Windows car la gestion « fork » de PostgreSQL va plus loin qu'un simple système de fork, elle fonctionne en parallèle avec une gestion de la mémoire partagée entre ces forks. En faire une version multi-threadée imposerait d'écrire en fait un nouveau programme, qui ne serait plus PostgreSQL. Nous reviendrons sur ces sujets dans les parties sur le tunning de configuration.

8. Gestion des versions de PostgreSQL

8.1. Emplacement des fichiers de configuration

Pour réussir à vous connecter à votre serveur PostgreSQl vous devrez peut-être modifier sa configuration. Il faut donc déjà dans un premier temps retrouver ces fichiers.

Pour PostgreSQl les fichiers de configuration doivent se trouver au niveau du répertoire de stockage des données du cluster. Ce répertoire se nomme « data dir ». Son emplacement dépend des méthodes d'installation.

Vérifier que vous avez toujours un terminal avec une session root.

Puis devenez l'utilisateur Postgres, celui qui est chargé de faire tourner le service/démon PostgreSQL. L'utilisateur root à le droit de devenir l'utilisateur qu'il veut sans mot de passe. L'utilisation du « - » dans la commande permet de bien charger les variables d'environnement de l'utilisateur.

```
# su – postgres
```

Si nous tapons « pwd » pour voir dans quel répertoire nous nous trouvons:

```
> pwd
/var/lib/postgresql
```

Nous obtenons un répertoire qui a de fortes chances d'être proche du data directory. Sur une distribution Linux les data directory sont en fait /var/lib/postgresql/<version> ce qui permet d'avoir plusieurs versions de PostgreSQL qui tournent. L'assistant d'installation sur un serveur Windows propose lui aussi de préfixer le data directory avec le numéro de version majeur.

Les fichiers de configuration sont donc normalement présents dans /var/lib/postgresql/9.0. Mais il pourraient aussi être installés dans /opt/PostgreSQL/9.0/data si j'avais utilisé un autre type de package que ceux fournis par la distribution officielle. La commande **ps** vous montrant les processus qui tournent peut vous aider à identifier le répertoire utiliser par le démon PostgreSQL, ce répertoire étant passé en option au processus PostgreSQL principal.

Sur les distributions de type Debian les packages ont installés des raccourcis forts utiles dans les répertoires classique de configuration:

```
/etc/postgresql/<version>
```

Sur Windows des raccourcis dans le Menu démarrer sont présents. Et nous pourrons aussi retrouver des raccourcis vers ces fichiers dans le client d'administration pgAdmin.

8.2. Compatibilité binaire, version majeure et mineure

Nous avons vu plusieurs fois qu'il est utile de préfixer les répertoires de stockage des données avec le numéro double de version, par exemple **8.4** ou **9.0** ou **9.1**. Ce numéro est le numéro de version **majeure**. Passer d'une version 8 à une version est un changement de version majeur, bien sur, mais passer d'une 8.3 à une 8.4 aussi, ou bien d'une 9.0 à une 9.2.

Si nous regardons à cette adresse: http://developer.postgresql.org/pgdocs/postgres/release.html nous voyons qu'il existe aussi des changements de version mineurs, par exemple la 9.0.4 http://developer.postgresql.org/pgdocs/postgres/release-9-0-4.html qui suit les version 9.0.3 9.0.2 et 9.0.1. Ces versions corrigent des bugs (sur la 9.0.4 on peut notamment voir un bug important de pg_upgrade avec les Blobs) et il est important de les appliquer. PostgreSQL assure la **compatibilité binaire** sur ces changements de versions. **Cela signifie que vous pouvez remplacer les binaires PostgreSQL de la version 9.0.2 avec ceux de la 9.0.5 sans aucun processus de migration de vos données** (à priori, lisez quand même les Changelog au cas où vous seriez impactés par un bug qui nécessite des traitements annexes).

Passer d'une version 9.0 à une version 9.2 sera par contre un changement majeur. Comme le passage d'une 8.3 à une 9.1. Il faudra effectuer un processus de migration et de transformation de vos données. Un utilitaire est prévu à cet effet: **pg_upgrade**.

En terme de développement cela signifie qu'il faut à priori utiliser une version cible de PostgreSQL, dans laquelle vous pourrez identifier les outils que vous avez à votre disposition. A cet effet une page très importante de la documentation est la **matrice des fonctionnalités**: http://www.postgresql.org/about/featurematrix.

Si vous trouvez une solution à un problème sur Internet et qu'elle ne marche pas sur votre base données n'hésitez jamais à aller vérifier qu'il ne s'agit pas d'une fonctionnalité apparue dans une version ultérieure.

De manière plus générale parcourir cette matrice vous permettra de découvrir des fonctionnalités que vous n'auriez peut-être jamais devinées.

9. Se connecter à PostgreSQL

9.1. pg_hba.conf : autoriser les connexions

PostgreSQL va gérer les autorisations de connexion aux bases à plusieurs niveaux. Nous verrons plus loin des moyens de contrôler les droits d'accès des login utilisateurs à l'intérieur de PostgreSQL. Mais ce qu'il faut bien garder à l'esprit c'est que toute la politique de gestion des droits que l'on pourrait appeler « politique des GRANT » ne s'applique qu'après un premier niveau de contrôle d'accès extérieur à PostgreSQL.

Le fichier pg_hba.conf est un fichier qui permet de contrôler de façons très simple ou très complexe quels sont les logins utilisateurs autorisés à se connecter à PostgreSQL, sur quelles bases, depuis quelles machines et avec quelles politiques de mot de passe.

A partir de ce fichier on peut restreindre l'accès à PostgreSQL depuis le localhost uniquement, ou bien à partir de certaines machines, on peut forcer un login à n'être utilisé que par un utilisateur système portant le même nom, ou bien appliquer un mapping entre les noms d'utilisateurs de l'OS et les logins de connexions, on peut utiliser des annuaires, etc.

L'étendue des possibilités est très grande, les deux références en la matière sont:

- les commentaires du fichier pg_hba.conf
- la documentation en ligne : http://www.postgresql.org/docs/9.0/static/auth-pg-hba-conf.html

Dans l'immédiat nous allons examiner les valeurs par défaut puis les étendre afin de rendre ce contrôle d'accès inopérant. Pour un administrateur de bases de données il conviendra d'appliquer des contrôles plus sévères en production.

Le plus souvent la configuration par défaut est:

```
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD

# "local" is for Unix domain socket connections only
local   all         all                               ident
# IPv4 local connections:
host    all         postgres    127.0.0.1/32          ident
host    all         all         127.0.0.1/32          md5
# IPv6 local connections:
host    all         all         ::1/128               md5
```

Ce qui signifie qu'en utilisant la socket Unix chaque utilisateur du système peut se connecter à une base qui porterait son propre nom (sans mot de passe). L'utilisateur postgres (superutilisateur) est autorisé à se connecter en TCP/IP mais en local sur la base postgres, sans aucun mot de passe mais uniquement depuis une session utilisateur postgres. Ensuite PostgreSQL autorise n'importe quel login de connexion en TCP/IP à destination de n'importe quelle base, sans considération sur l'identité réelle de celui qui se connecte, à partir du moment ou il peut donner le md5 du mot de passe (ce que font la plupart des clients de connexion PostgreSQL).

On constate que par défaut aucune machine distante n'est autorisée.

- *En développement, **et pour cette formation on éditera ce fichier** pour remplacer chaque fin de ligne (les ident/md5/etc) par « **trust** ». Cela veut dire « confiance ». Vous demandez donc à PostgreSQL de faire une confiance aveugle, tous les utilisateurs peuvent se connecter sur toutes les bases, avec le login de leur choix, sans même avoir besoin d'indiquer le bon mot de passe.*

Le trust ne devrait surtout pas être utilisé en production bien sûr. Cela permet d'abstraire cette première couche de sécurité, cela n'exclue pas des contrôles d'accès supplémentaires au sein de PostgreSQL en fonction du login utilisé. C'est un peu comme si vous aviez un Windows avec des comptes utilisateurs différents, y compris un compte administrateur (ici postgres), sans mots de passe. Il faut cependant bien se connecter avec le compte administrateur pour avoir les accès administrateur.

☑ N'oubliez pas de **redémarrer** PostgreSQL quand vous modifiez le pg_hba.conf !

9.2. psql : ligne de commande

La ligne de commande `psql`, disponible aussi sous Windows, est l'outil principal de communication avec le serveur. Ce client de connexion à un serveur est le plus simple (un simple écran de ligne de commande interactive) mais il est toujours disponible, y compris à distance sur une session SSH sur un serveur qui n'autorise pas les connexions extérieures. Vous devriez toujours apprendre à vous en servir, pour exécuter des scripts SQL, pour taper quelques commandes simples, ou pour devenir votre principal moyen de communication avec le serveur PostgreSQL.

Il faut disposer d'un client psql d'une version au moins supérieure à la version du serveur (un client 8.4 ne saurait pas parler avec un serveur 9.0).

Cette commande est l'équivalent de la commande `mysql` avec MySQL ou `sqlplus` avec oracle.

Comme toute ligne de commande cette commande accepte un grand nombre d'options à commencer par un «`--help`»

```
$ sudo su -
# su - postgres
$ psql --help
psql est l'interface interactive de PostgreSQL.

Usage :
  psql [OPTIONS]... [NOM_BASE [NOM_UTILISATEUR]]

Options générales :
  -c, --command=COMMANDE
                  exécute une commande unique (SQL ou interne), puis
quitte
  -d, --dbname=NOM_BASE
```

```
                  indique le nom de la base de données à laquelle se
                  connecter (par défaut : « postgres »)
  -f, --file=FICHIER
                  exécute les commandes du fichier, puis quitte
  -l, --list      affiche les bases de données disponibles, puis
quitte
  -v, --set, --variable NOM=VALEUR
                  initialise la variable psql NOM à VALEUR
  -X, --no-psqlrc ne lit pas le fichier de démarrage (~/.psqlrc)
  -1 (« un »), --single-transaction
                  exécute un fichier de commande dans une transaction
unique
  --help          affiche cette aide, puis quitte
  --version       affiche la version, puis quitte

Options d'entrée/sortie :
  -a, --echo-all  affiche les lignes du script
  -e, --echo-queries
                  affiche les commandes envoyées au serveur
  -E, --echo-hidden
                  affiche les requêtes engendrées par les commandes
internes
  -L, --log-file=FICHIER
                  envoie les traces dans le fichier
  -n, --no-readline
                  désactive l'édition avancée de la ligne de commande
                  (readline)
  -o, --output=FICHIER
                  écrit les résultats des requêtes dans un fichier (ou
                  |tube)
  -q, --quiet     s'exécute silencieusement (pas de messages,
uniquement le
                  résultat des requêtes)
  -s, --single-step
                  active le mode étape par étape (confirmation pour
chaque
                  requête)
  -S, --single-line
                  active le mode ligne par ligne (EOL termine la
commande
                  SQL)

Options de formattage de la sortie :
  -A, --no-align  active le mode d'affichage non aligné des tables (-P
                  format=unaligned)
  -F, --field-separator=CHAINE
                  initialise le séparateur de champs (par défaut : « |
»)
                  (-P fieldsep=)
  -H, --html      active le mode d'affichage HTML des tables (-P
format=html)
  -P, --pset=VAR[=ARG]
                  initialise l'option d'impression VAR à ARG (voir la
                  commande \pset)
  -R, --record-separator=CHAINE
                  initialise le séparateur d'enregistrements (par
défaut :
```

```
                  saut de ligne) (-P recordsep=)
  -t, --tuples-only
                  affiche seulement les lignes (-P tuples_only)
  -T, --table-attr=TEXTE
                  initialise les attributs des balises HTML de tableau
                  (largeur, bordure) (-P tableattr=)
  -x, --expanded  active l'affichage étendu des tables (-P expanded)

Options de connexion :
  -h, --host=HOTE nom d'hôte du serveur de la base de données ou
répertoire
                  de la socket (par défaut : /var/run/postgresql/)
  -p, --port=PORT port du serveur de la base de données (par défaut :
                  « 5432 »)
  -U, --username=NOM
                  nom d'utilisateur de la base de données (par
défaut :
                  « postgres »)
  -w, --no-password
                  ne demande jamais un mot de passe
  -W, --password  force la demande du mot de passe (devrait survenir
                  automatiquement)

Pour en savoir davantage, saisissez « \? » (pour les commandes
internes) ou
« \help » (pour les commandes SQL) dans psql, ou consultez la section
psql
de la documentation de PostgreSQL.
base
Rapportez les bogues à <pgsql-bugs@postgresql.org>.
```

Pour nous connecter à notre serveur local avec l'utilisateur postgres nous taperons donc:

```
$ psql -h localhost -p 5432 -U postgres
postgres=#
```

la ligne

```
postgres=#
```

signifie que nous sommes dans la session interactive de `psql`. Les commandes les lus utiles dans un premier temps seront:

```
\h
```

Qui donne une aide sur le SQL. Vous pouvez en effet requêter en SQL depuis cette ligne de commande pour créer des objets ou les interroger.

```
\?
```

Qui donne tous les raccourcis de `psql`. Ces raccourcis permettent de lister les base (`\l`) les tables, les utilisateurs, les droits, etc. Toutes ces commandes sont en fait des raccourcis correspondant à des requêtes très complexes effectuées sur le **catalogue**. Le catalogue est une base de données spéciale utilisée en interne par PostgreSQL et donnant accès à toutes les informations (information_schema). Le catalogue dans PostgreSQL se somme `pg_catalog` et peut varier dans sa structure d'une version à l'autre de PostgreSQL, c'est pourquoi le client ligne de commande doit avoir une version supérieure au serveur, il doit savoir comment interroger le `pg_catalog`.

Au passage nous allons voir une commande fort utile de `psql`. Tout d'abord sortons de cette invite de commande avec:

```
\q
```

Notez que vous pouvez aussi utiliser le CONTROL+D général qui veut dire « **sortir** » à peu près partout et sur tous les systèmes d'exploitation (mais pas dans vi, ceci est un autre problème...)

Nous allons nous reconnecter en utilisant l'option `-E` qui permet de faire un echo des commandes utilisées par `psql`.

```
$ psql -h localhost -p 5432 -U postgres -E
```

Puis nous tapons

```
\d
```

Cette commande signifie : « affiche la liste des tables et vues de la base courante »

Nous n'obtenons à priori aucun résultat, nous n'avons pas créé de tables. Mais nous avons en echo la requête effectuée sur le catalogue. Pour un serveur 8.4 cela donne:

```
SELECT n.nspname as "Schema",
  c.relname as "Name",
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'i'
THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's' THEN 'special' END as
"Type",
  pg_catalog.pg_get_userbyid(c.relowner) as "Owner"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','v','S','')
      AND n.nspname <> 'pg_catalog'
      AND n.nspname <> 'information_schema'
      AND n.nspname !~ '^pg_toast'
  AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
```

Ainsi si vous voulez un jour faire des requêtes dans le catalogue pour obtenir des informations utiles (quel est le code de mon trigger? Quelles sont les tables sur lesquelles l'utilisateur toto à les droits d'écriture?) vous pourrez demander à votre `psql` de vous montrer comment lui interroge le catalogue et vous en inspirer.

En regardant l'aide nous voyons que la commande `\d` peut s'écrire avec un grand S pour obtenir la liste des tables et vues du système, il s'agirait alors des tables et vues de la 'base' `pg_catalog`. Essayez de taper:

```
\dS
```

Vous obtenez un résultat qui est paginé par le programme car le contenu est trop long pour tenir à l'écran (utilisez les touche entrée et espace). Pour sortir de ce programme de pagination du résultat il faut taper « **q** ».

Certaines autres commandes permettent de changer de base de données courante (`\use`) ou d'effectuer des commandes en dehors de la ligne de commande (commandes système) ou de changer les options d'affichage, etc.

Sortons de l'invite de commande et utilisons maintenant `psql` pour effectuer une simple requête SQL et stocker le résultat dans un fichier (dans le dossier tmp):

```
psql -h localhost -p 5432 -U postgres -c "SELECT * from
pg_catalog.pg_database" -o /tmp/resultat.txt
```

Vous pouvez visualiser le résultat avec un simple

```
cat /tmp/resultat.txt
```

Ou bien utiliser un éditeur de texte comme **nano** ou … **vi**. (pour quitter vi faites ECHAP:q ou ECHAP:x pour sauver, évitez le reboot serveur pour quitter vi, évitez vi en fait)

Pour formater le résultat différemment vous pouvez tenter des combinaisons avec `--html`, `--tuples-only`, sans l'option `-o`, etc.

Si vous regardez en détail l'aide fournie avec `\?` dans la session interactive vous pourrez retrouver de nombreux moyens d'effectuer des traitements similaires (exporter des résultats formatés) dans un session interactive.

L'utilisation la plus courante de `psql` est l'intégration de fichier SQL. Ceci se fait avec l'option `-f` qui permet de faire jouer un fichier contenant de multiples commandes. Ce fichier d'entrée pourrait alors contenir des commandes `\*`, elles seraient comprises par `psql` comme des commandes différentes du SQL, spécifiquement destinées à `psql`, et pourraient alors servir à changer de base de donnée, construire des exports formatés, etc.

notez aussi l'option `-1` qui permet de forcer l'intégralité d'un script pouvant contenir plusieurs commandes sous la forme d'une transaction unique (nous reviendrons sur les transactions plus tard).

Dernier petit détail avec `psql`. Nous avons jusque ici utilisé la connexion en, TCP/IP. Nous avons vu dans le fichier pg_hba.conf qu'il existait aussi un mode de connexion par **socket Unix**. Le serveur PostgreSQL écoute sur une socket réseau mais il écoute aussi sur un socket fichier. Le passage par une socket ficher n'est possible qui si l'on est situé sur la même machine (accès au système de fichiers) mais évite toutes les lenteurs dues au passage des informations à travers le protocole TCP/IP. Pour des traitements importants comme des migrations ou des importations (voir des backups) on devrait donc privilégier l'accès par socket. Il suffit donc de connaître le chemin vers cette socket pour réussir cette connexion:

```
psql -h /var/run/postgresql -U postgres
```

fonctionne (si votre fichier pg_hba.conf l'autorise). Si vous regardez dans le répertoire /var/run/postgresql vous retrouvez bien un fichier spécial (la socket):

```
/var/run/postgresql/.s.PGSQL.5432
```

/var/run/postgresql est la valeur par défaut pour l'option `-h` et `-U` postgres est la valeur par défaut pour `-U` (nom de l'utilisateur courant). Un simple

```
psql
```

suffit donc à vous connecter en tant qu'utilisateur postgres via la socket locale. Mais si vous n'êtes pas l'utilisateur postgres vous aurez certainement besoin de spécifier le rôle utilisateur que vous voulez utiliser (postgres) et le moyen de connexion (TCP/IP+port ou socket), sachant que le fichier pg_hba.conf par défaut n'autorise que l'utilisateur postgres sur la socket.

9.3. PgAdmin III : le client graphique

L'administration en ligne de commande a ses fans. Mais un outil graphique s'il est bien fais est certainement un atout indispensable pour le développeur de base de données comme pour l'administrateur. Cet outil existe pour PostgreSQL, il s'agit de **pgAdmin III,** souvent nommé **pgadmin.**

Nous utiliserons principalement cet outil dans cette formation. Nous n'allons donc pas faire très long dans un premier temps. Signalons simplement qu'il est possible à partir de cet outil de se connecter à plusieurs serveurs, sous des identités différentes. Mais comme pour le client psql il faut disposer d'un client ayant la connaissance du schéma du `pg_catalog` du serveur, donc avec un supérieur d'une version supérieure ou égale à celle du serveur. pgAdmin effectue un grand nombre de requêtes sur le catalogue afin de nous présenter tous ces objets de façon graphique; Il n'y a pas de protocole de communication magique entre les clients et le serveur, tout se passe avec le langage SQL.

9.4. PhpPgAdmin: le client web

De la même façon qu'il existe un client d'administration web de MySQL bien connu (phpMyAdmin) Il existe phpPgAdmin, que l'on peut facilement installer sous forme de package.

```
sudo apt-get install phppgadmin
```

Le fichier de configuration apache (phppgadmin) installé se trouve dans

```
/etc/apache2/conf.d
```

Il serait certainement plus utile sous la forme d'un VirtualHost dédié sur une installation de production, il faudrait alors le déplacer dans

```
/etc/apache2/sites-available/100-phppgadmin
```

Puis l'éditer un peu afin de la transformer en VirtualHost. Avec l'installation par défaut tous les VirtualHost disposent de l'application sous l'URL /phppgadmin/.

Nous n'en ferons rien pour ce tutoriel, on laisse ce fichier intact. En tapant http://localhost/phppgadmin/ dans un navigateur on obtient donc cette page (la version de phppgadmin pouvant varier):

Illustration 1: écran d'accueil phppgadmin

En listant les fichiers installés depuis le package :

```
dpkg -L phppgadmin
```

On trouve un fichier

```
/etc/phppgadmin/config.inc.php
```

c'est dans ce fichier que l'on va pouvoir définir des connexions au serveur PostgreSQL en combinant des adresses de serveurs, des logins utilisateur et des mots de passe. Si vous voulez autoriser une connexion super-utilisateur via cette interface vous devrez changer la valeur d'une des options:

```
$conf['extra_login_security'] = false;
```

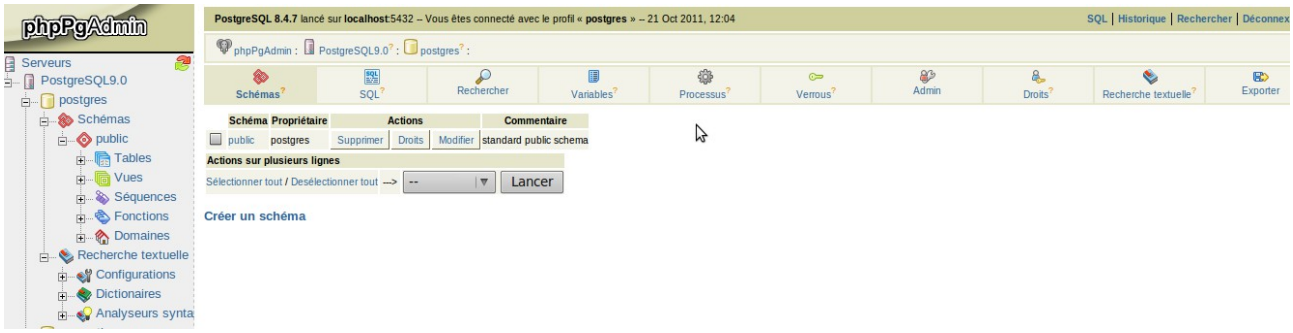


Illustration 2: Un des écrans de phppgadmin

10. Créer une base de donnée

10.1. Le cas d'exemple de la formation

Pour avoir un cas d'espèce à étudier nous imaginerons que nous sommes dans un société. Dans cette société un service de Relation humaine possède des données qui sont stockées dans une base PostgreSQL. Une partie de ces données est sensible (comme les salaires) et seuls quelques utilisateurs devraient pouvoir y accéder.

Cette société fabrique des fournitures du types trombones et punaises et propose en interne à ses employés et employés de passage (des intérimaires) de passer des commandes de fournitures.

Dernièrement il a été décidé qu'un système de points de fidélité serait mis en place afin de récompenser les employés et intérimaires qui passent des commandes de fournitures en interne. Un nombre variable de points seront attribués à la personne en fonction du montant de la commande et un nombre fixe de points sera attribué au service ou à l'agence d'intérim de l'employé. L'usage final de ces points n'est pas dans notre exemple.

La partie qui nous intéresse est que certains employés sont chargés de mettre en place une base de donnée PostgreSQL pour leur application, qui sera appelée sous le code projet APP. Ces employés devront donc accéder aux données, mais pas aux données interdites, et ils devront mettre en place un système de d'enregistrement des commandes et d'attribution des points.

10.2. Utilitaires en ligne de commande

Pour créer des bases de données ou pour créer des utilisateurs dans les bases données il existe des utilitaires en ligne de commande installés avec PostgreSQL.

```
createdb
creatuser
createlang
```

Ces lignes de commandes acceptent l'option `--help` ainsi que bien d'autres options. Leur maniement est simple et nécessite soit d'être l'utilisateur root soit d'être l'utilisateur postgres.

Cependant pour cette formation nous utiliserons les outils graphiques mis à disposition par pgadmin. Ceux-ci présentent le même nombre d'options, avec cependant l'énorme avantage d'être graphiques.

Signalons que l'utilitaire **`createlang`** permet d'ajouter le support de langages différents dans une base de donnée.

10.3. Cluster, Encodage des caractères, Locales, langages et templates

Pour bien comprendre la création d'une base de donnée il faut revenir sur quelques concepts.

Tout d'abord le serveur de bases de données postgreSQL est un **cluster**. Cela signifie que pour un même démon (ou service sous Windows) nous pouvons installer plusieurs bases de données différentes. Ces bases seront très fortement séparées. Il n'existe pas à priori de moyens pour effectuer des requêtes utilisant des tables situées dans des bases séparées. Cette distinction tend à disparaître depuis PostgreSQL 9 avec l'arrivée des connecteurs spéciaux de type **dblink.**

Avant PostgreSQL9 toutes ces bases devaient partager une même **locale (collation)**, qui était installée sur le cluster lui-même, il est dorénavant possible de spécifier des locales différentes pour chacune de ces bases. La locale va indiquer des choses dépendantes d'une langue et d'une culture:

– l'ordre de tri alphabétique

- les séparateur de milliers et de décimales
- le symbole monétaire
- l'ordre des éléments dans une date

Les locales se nomment par exemple fr_FR.UTF_EUR ou fr_FR_ISO8859-1_EUR, ou encore C (la locale de l'ASCII7) ou en_US.

Enfin l'encodage des caractères va déterminer la taille minimale d'un caractère et l'encodage utilisé pour le transformer d'une série de bits à un caractère reconnu. C'est ici que l'on indique si on utilise de l'ASCII7 (pas d'accents), de l'ISO8859-1 (latin1), ou de l'UTF8.

Ces paramètres sont propres à chacune des bases de données hébergées sur le cluster. Enfin on trouvera aussi des langages. PostgreSQL supporte le langage SQL, le C et le pl/PgSQL nativement pour toutes les bases de données. Avant la version 9 le pl/pgSQL devait être ajouté sur une base pour qu'il soit supporté (d'où la présence de la commande « `createlang plpgsql` » dans de nombreuses documentations). D'autres langages peuvent être ajoutés, comme le pl/Perl, ou le pl/Python.

Un cluster nouvellement instancié contient 3 bases:

- **postgres**: la base qui contient toutes les informations système permettant de gérer et d'interroger le cluster. C'est ici que sont stockés les utilisateurs et groupes de connexion ainsi que les tables du catalogue
- **template1**: quand vous aller créer un base de données elle sera en fait une copie de cette base. Toutes les tables, tous les types, langages, et toutes les données déjà installés dans cette base seront donc recopiés dans la nouvelle.
- **template0**: est une copie de template1 faite à l'initialisation, il s'agit donc du modèle « propre et intact », ne modifiez jamais ce modèle, c'est une sauvegarde au cas où vous endommagez template1.

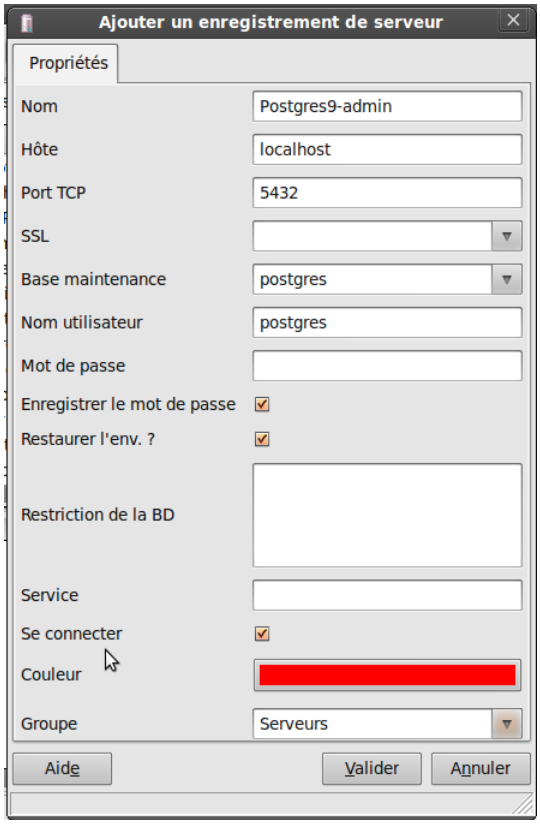


Illustration 3: Création d'une connexion à la base

10.4. Créer une connexion administrateur avec pgAdminIII

Au niveau théorique nous sommes quasi prêts. Nous allons créer une nouvelle base sans utiliser la commande `createdb`, ni en utilisant **PHPpgadmin**. Nous allons plutôt utiliser **pgadmin**. Pour cela il faut que nous commencions par nous créer une connexion super utilisateur dans cet outil.

Cela se fait depuis le menu « Fichier > Ajouter un serveur ». Nous allons entrer des paramètres jusqu'à ce que nous arrivions à nous connecter à PostgreSQL en tant qu'utilisateur postgres. Nous aurons peut-être besoin pour cela de corriger le fichier pg_hba.conf si cette connexion nous est refusée.

Cette connexion est assez dangereuse, nous sommes super-utilisateur et nous pouvons tout casser. Nous retiendrons ce fait en ajoutant une couleur **rouge** à cette connexion.

10.5. Créer une base de donnée formation

En double cliquant sur la connexion nous l'ouvrons. Dans l'arbre des menus qui apparaît nous avons quatre premiers menus, rôles de connexion, groupes de connexion, tablespaces et bases de données. Nous voulons créer une base de données donc:

clic droit base de données > Ajouter une base de données...

Ceci ouvre un assistant que nous remplissons en suivant le copie d'écran (nom formation, en UTF8, collation et type de caractère fr_FR.utf8). Si vous naviguez parmi les onglets présents vous pourrez voir que le dernier onglet, comme sur presque tous les assistants, vous indique la commande SQL que l'assistant génère.

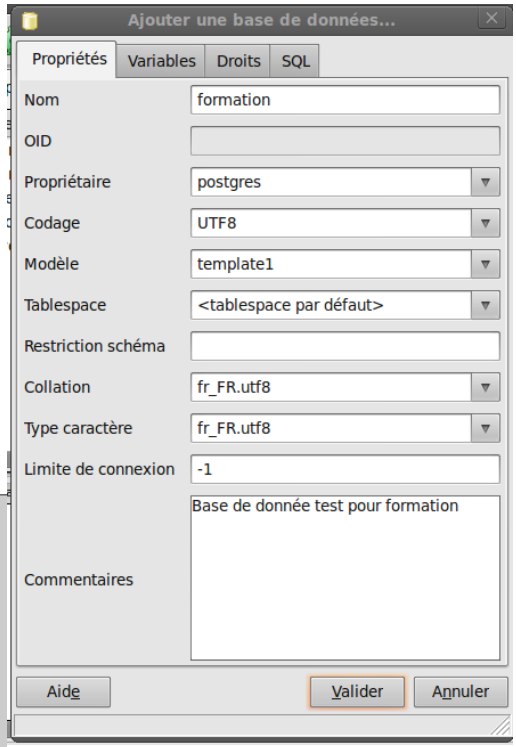


Illustration 4: Assistant de création de base de données

```
CREATE DATABASE formation
  WITH ENCODING='UTF8'
       OWNER=postgres
       TEMPLATE=template1
       LC_COLLATE='fr_FR.utf8'
       LC_CTYPE='fr_FR.utf8'
       CONNECTION LIMIT=-1;
COMMENT ON DATABASE formation IS 'Base de
données test pour formation';
```

☑ Remarquez la case lecture seule sur l'assistant, si vous décochez cette case vous pouvez alors ajouter vous-même du code SQL que pgAdmin ne saurait pas intégrer à la requête. Les écrans de l'assistant ne peuvent du coup plus fonctionner. Ceci vous permet d'avoir un train d'avance sur pgadmin (une nouvelle option qui n'est pas encore intégrée dans l'outil), ne l'oubliez pas. En l'occurrence nous n'avons pas besoin de modifier ce code SQL.

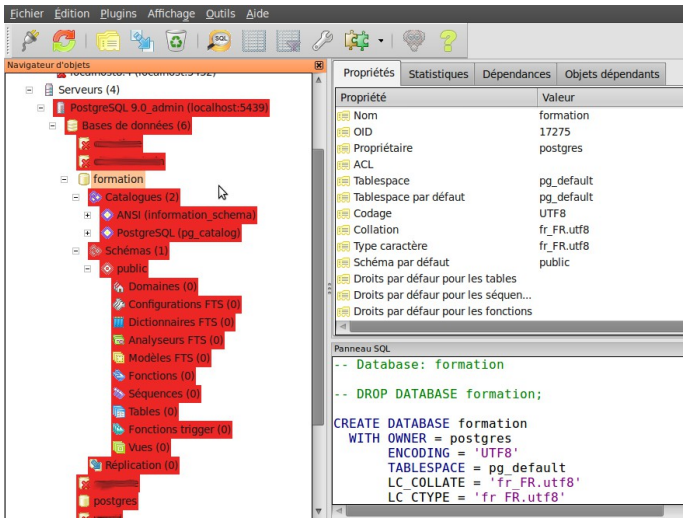


Illustration 5: Objets disponibles dans la base

Une fois la base créée vous pouvez naviguer dans pgadmin et voir que de nombreux objets pourraient exister au sein de cette base, à commencer :

- des catalogues: il y en a deux, le pg_catalog et le catalogue information_schema qui est une obligation de toutes les bases de données se conformant à la norme SQL ANSI. Si vous naviguez dans le catalogue vous y trouverez un grand nombre d'objets qui sont disponibles pour tous en permanence.
- Des schémas, un seul au départ, nommé public. Dépliez le pour y retrouver les objets de la base

- des domaines (il s'agit de définitions de types propres à cette base, avec des contraintes associées (comme une chaîne de caractère devant se conformer à une expression régulière)
- plusieurs objets comportant le mot FTS, signifiant Full Text Search. Il s'agit des différents éléments devant être mis en place pour ajouter des fonctionnalités de recherche plein texte
- des fonctions, si vous regardez l'équivalent dans le pg_catalog vous retrouverez toutes les fonctions que vous pouvez utiliser dans les requêtes (comme substring par exemple)
- des séquences, il s'agit d'objets permettant de générer des auto-incréments (pour faire simple)
- des tables bien sûr
- des vues, nous verrons ce que cela signifie
- des fonctions trigger, là encore nous verrons plus tard ce que cela recouvre.

11. Tablespaces

Nous venons de créer une base. Nous aurions pu avant cela créer un tablespace. Heureusement pour nous un tablespace par défaut existe et a été choisi pour nous.

Le tablespace est la représentation interne pour le serveur d'un espace de stockage sur le disque.

Le tablespace par défaut (pg_default) représente donc l'espace de stockage qui a été créé dans le « data dir ». Nous voyons dès lors qu'il est en fait possible de créer plusieurs répertoires de données.

Sur un serveur qui ne possède qu'un seul contrôleur de disques cela n'est le plus souvent pas du tout utile. Créer des tablespaces différents permet de forcer la répartition physique des données sur des disques physiques différents. Il est tout à fais possible de créer plusieurs tablespaces sur un même disque mais c'est en fait inutile. Le vrai intérêt du tablespace est de forcer l'usage de disques différents. Tous les objets créés dans la base de donnée auront un tablespace affecté. Une des techniques d'optimisation courante avec les tablespaces consiste par exemple à forcer le stockage des index d'une table sur un disque et les données de la table sur un autre (en utilisant deux tablespaces différents). Lorsque le serveur fera des requêtes intensives sur cette table il pourra ainsi paralléliser les lectures d'index et les lectures de données, au niveau du système d'exploitation et de l'électronique des contrôleurs de disques les traitements seront parallélisés.

La gestion des tablespaces est donc un élément de tunning assez avancé que nous pouvons ignorer pour le moment (il est possible de modifier les tablespaces des objets après leur création, mais cette opération aura des impacts si les données sont importantes en taille).

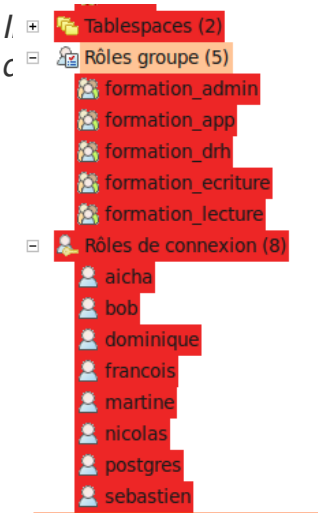
12. Définitions des rôles et droits

Pour le moment nous avons une nouvelle base mais elle appartient à l'utilisateur postgres. Afin de nous placer dans un cas plus réaliste il nous commencer à créer des logins, des rôles. Ceci pourrait se faire à l'aide de la commande `createuser`, mais là encore nous allons plutôt utiliser les assistants de pgadmin.

> ☑ Dans le futur retenez que seul le super utilisateur est capable de gérer les utilisateurs dans pgadmin, avec des connexions de moindre niveau vous n'aurez plus la possibilité de créer ou modifier les utilisateurs et groupes.

12.1. Les rôles de connexions

- **Aicha**: DBA (Administrateur de base de données) responsable de cette application. Nous lui créons un rôle pour éviter d'utiliser l'utilisateur postgres. Nous pourrions aussi créer un rôle portant le nom de la base, histoire de bien séparer les droits d'administrations par base. Aicha à tous les droits.
- **Martine**, **Dominique** et **Sébastien** sont des utilisateurs de la base de données. Ils sont responsable de la mise en place d'une application que nous nommerons 'app'. Sébastien travaille au service DRH, il a le droit de voir le contenu complet des tables de la DRH. Ce n'est pas le cas pour les autres, qui ne devraient voir qu'un sous ensemble des données de la partie DRH. Quand l'application sera prête un nouveau rôle sera créé pour le programme utilisant la base, dans l'immédiat des rôles individuels avec des mots de passe leur sont attribués à chacun.
- **Nicolas** et **François** : deux personnes ayant le droit d'alimenter les données de la base, mais aussi de supprimer ces données. **Nicolas** est de plus DRH et est le seul à pouvoir alimenter les données en rapport avec la gestion du personnel, par contre Nicolas n'a rien à voir avec l'application 'app'.



Classiquement les bases de données s'utilisent avec **un seul login** qui possède **tous les droits** sur la base. C'est le cas par exemple quand on déploie une application LAMP chez un hébergeur mutualisé. Celui-ci mets à disposition une base de données avec un login utilisateur unique pour l'application. Il est cependant possible de **gérer plusieurs logins par base de données**. Une application qui au minimum serait capable d'utiliser **deux** connexions, une en écriture et une en lecture pourrait ainsi beaucoup plus facilement passer à une gestion distribuée sur plusieurs serveurs 'esclaves' des opérations de lecture tout en se concentrant sur le serveur maître pour les parties applicatives nécessitant des lectures et écritures (pour toutes les transactions par exemple).

➢ *Créez les utilisateurs aicha, sebastien, martine, dominique, nicolas et francois*

Mais PostgreSQL peut même gérer un grand nombre d'identifiants de connexions, basés par exemple sur les utilisateurs du système d'exploitation ou les utilisateurs d'un annuaire. En terme de gestion des droits associés il faut donc aussi gérer des profils que l'on retrouvera sous la notion de groupes dans PostgreSQL. Depuis plusieurs versions de ce SGBD la distinction entre rôle de connexion et groupes de rôles de connexions est abstraite, car en fait tout rôle peut contenir d'autres rôles, dit autrement, tout rôle peut hériter d'un autre.

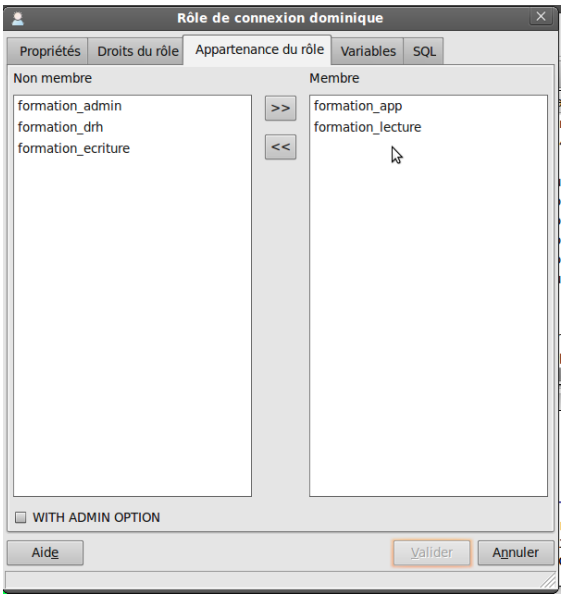


Illustration 7: Affecter des groupes à un rôle

Nous créerons autant de rôles que d'utilisateurs et nous y associerons des « **rôles groupes** » qui marqueront les différentes politiques de droits que nous aurons à gérer (bien sur ces politiques dépendent de choix fonctionnels, il n'y a aucune obligation à créer ces rôles à chaque fois que vous créez une base avec PostgreSQL):

- **formation_admin** : rôle des administrateurs de cette base de donnée au sein du serveur de base de

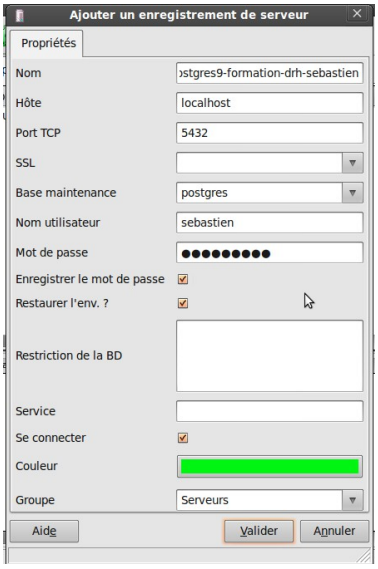
données. Si vous gérer un seul login et un seul niveau de droit il vous faudra sans doute n'utiliser que ce rôle. (aicha)

- **formation_ecriture**: rôle permettant d'ajouter des données dans la base (nicolas et francois)
- **formation_lecture**: rôle permettant de requêter la base (sebastien, martine et dominique)
- **formation_app**: rôle des utilisateurs gérant l'application « app » (sebastien, martine, dominique et francois)
- **formation_drh**: rôle des utilisateurs du service DRH (nicolas et sebastien)

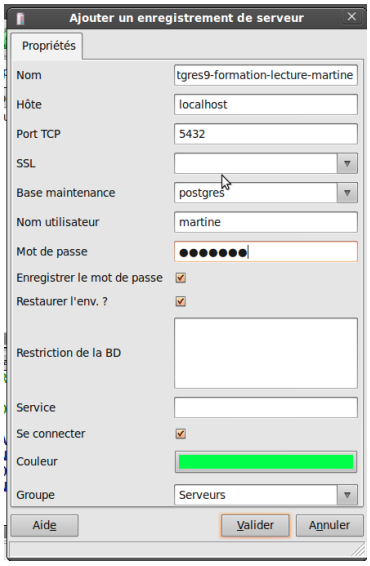
Ce qui donne par rôle:

- **aicha**: formation_admin
- **nicolas**: formation_ecriture & formation_drh
- **francois**: formation_ecriture & formation_app
- **sebastien**: formation_lecture & formation_app & formation_drh
- **martine**: formation_lecture & formation_app
- **dominique**: formation_lecture & formation_app

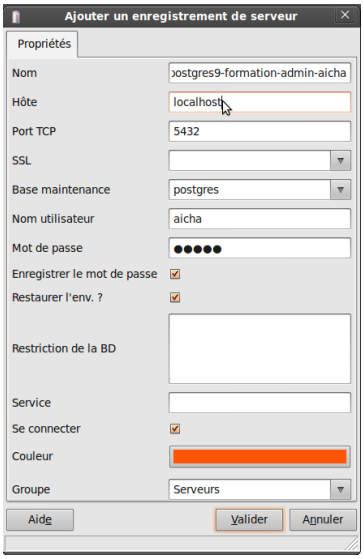
12.2. Créer des connexions utilisateur avec PgAdminIII



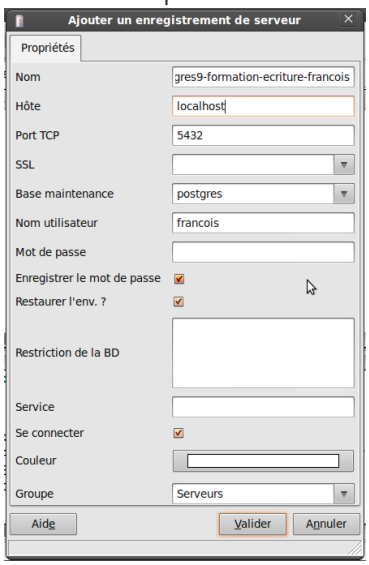
Nous allons créer cinq nouvelles connexions, une pour chaque groupe (formation_admin, formation_ecriture, formation_lecture et formation_drh), en prenant à chaque fois l'un des utilisateurs de ces groupes comme login de connexion, pour le dernier groupe nous créons deux connexions, une avec nicolas, une avec sebastien. En effet nicolas appartient au groupes formation_ecriture et formation_drh alors que sebastien appartient à formation_lecture, formation_app et formation_drh; il est donc intéressant d'étudier la différence entre ces deux profils.



Plus tard nous appliquerons les droits afférants à chacun des groupes, nos cinq connexions (six en comptant celle de postgres) se font sur la même base de données mais avec des utilisateurs différents, elles devraient donc posséder des limitations différentes.



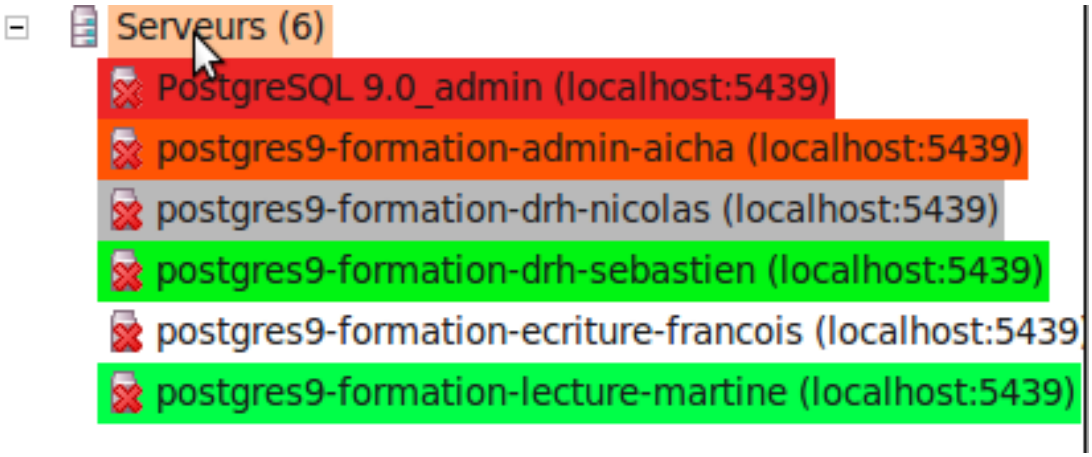
Nous choisirons la couleur orange pour la connexion avec les droits d'administration de la base (formation_admin). Vert pour la connexion limitée en lecture, et aucune couleur pour la connexion avec droits en écriture. Ainsi nous visualiserons mieux dans pgadmin le niveau de **danger** de chacune des ces connexions. Pour les deux dernières nous choisissons gris pour nicolas et encore vert pour sébastien.



Ces connexions se nomment Serveurs dans pgadmin. Il faut donc créer cinq 'serveurs', qui sont en fait à chaque fois le même serveur PostgreSQL 9.0, sur le même port, mais avec un utilisateur de connexion différent:

- postges9-formation-admin-aicha: login aicha : couleur **orange**
- postges9-formation-ecriture-francois: login francois : **pas de couleur**
- postges9-formation-lecture-martine: login martine : couleur **verte**
- postges9-formation-drh-nicolas: login nicolas : couleur **grise**
- postges9-formation-drh-sebastien: login sebastien : couleur **verte**

☑ Attention: après avoir créé les connexions il est très fortement recommandé de quitter pgadmin puis d'y revenir. Les connexions sont sauvegardées lors de la sortie du programme, qui malheureusement est parfois instable et ne quitte pas toujours proprement.



12.2.1. Retour au pg_hba.conf

➢ *Maintenant que nous disposons de plusieurs connections sur la base formation, les utilisateurs les plus en avance pourront aller modifier leur pg_hba.conf pour appliquer des droits ou des restrictions en fonction des logins, ou pour autoriser des connexions depuis des machines extérieures et observer les blocages produits dans les connexions de pgAdmin.*

12.3. Les schémas

– http://docs.postgresql.fr/9.0/ddl-schemas.html

On peut voir de façon simplifiée les schémas comme des bases de données à l'intérieur d'une base de données. Par défaut un seul schéma existe dans une base de données, le schéma public. Mais une base peut contenir plusieurs schémas qui seront autant de sous-bases dans la base.

Une des principales application des schémas est l'application de politiques de droits par schémas. Mais ils permettent aussi de mieux scinder des aspects purement fonctionnels de la base. L'avantage de l'utilisation des schémas par rapport à l'utilisation de plusieurs bases est qu'il est possible d'effectuer des requêtes impactant plusieurs schémas d'une base, ce n'est pas le cas entre plusieurs bases.

12.4. Les droits d'accès dans PostgreSQL et les schémas

http://www.postgresql.org/docs/9.0/static/sql-grant.html

Plusieurs niveaux de droits existent dans PostgreSQL:

– **les droits d'accès à la base**, une couche qui peut faire redondance avec les restrictions situées dans ph_hba.conf (mais la redondance n'est pas un mal en terme de sécurité)

– droits sur les **créations d'objets** dans la base (tables, tables temporaires)

– droits par défaut sur des éléments du langage SQL. Ainsi le `SELECT` donne le droit de visualiser les données, mais pour les commandes permettant d'éditer les données on va retrouver un ensemble de droits distincts. Les classiques `INSERT`, `UPDATE`, `DELETE` et `TRUNCATE` (qui est une variation du `DELETE`). Mais aussi `REFERENCES` (le droit de créer des clefs étrangères) et `TRIGGER` (le droit de créer des triggers).

– Au niveau des schémas on va pouvoir redéfinir les droits par défaut des instructions SQL. Mais aussi directement un droit d'accès ou non aux objets de ce schéma (`USAGE`)

- ces droits du langage SQL (`SELECT`, etc) peuvent ensuite être modifiés au niveau de chaque table.
- Avec PostgreSQL 9 ces droits vont même aujourd'hui jusqu'aux colonnes des tables

Il est donc possible de mettre en place des politiques assez complexes. **Dans la pratique les objets principaux sur lesquels vous devriez appliquer des droits sont les schémas**.

PostgreSQL ne permet pas les requêtes entre plusieurs bases, si vous voulez utiliser plusieurs applications avec des droits très différents vous devriez donc, plutôt que de mettre en place plusieurs bases de données dans le cluster, mettre en place plusieurs schémas dans la base.

> ☑ Vous pouvez voir un schéma comme des « sous-bases ». ces sous-bases ayant la possibilité d'être poreuses, vous allez pouvoir faire transiter ou pas des informations entre les schémas. Chose que vous ne pourrez pas faire entre les bases de données à moins d'utiliser des programmes externes ou des connecteurs particuliers (comme dblink).

12.5. Création des schémas drh et app

Nous allons utiliser la connexion rouge de postgres. Car c'est pour l'instant le propriétaire de la base.

Dans pgadmin nous allons sur l'objet schémas dans l'arborescence et ajoutons un nouveau schéma avec l'assistant de pgadmin. Nous allons créer le schéma « **drh** » avec « **formation_drh** » comme propriétaire.

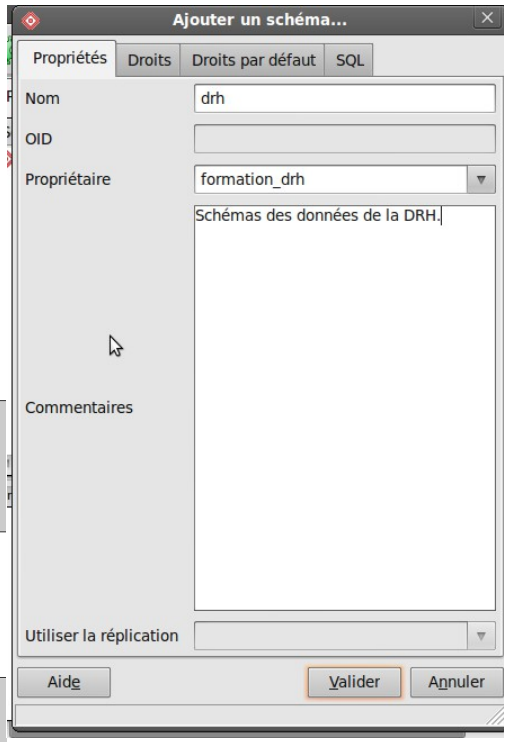


Illustration 8: Création du schéma drh

On constate que le SQL généré est:

```
CREATE SCHEMA drh
       AUTHORIZATION formation_admin;
COMMENT ON SCHEMA drh IS 'Schémas des
données de la DRH.';
```

On peut donc utiliser une commande SQL pour générer le second schéma (app) sur lequel on garde formation_admin en propriétaire (remarquez la double apostrophe dans le commentaire):

```
CREATE SCHEMA app
       AUTHORIZATION formation_admin;
COMMENT ON SCHEMA app IS 'Schémas des
données de l''application app.';
```

Pour taper du SQL utilisez le bouton **SQL** de pgadmin dans la barre d'outil, la connexion en cours sera utilisée.

Rafraichissez les données de pgadmin avec la touche **F5** ou le bouton rafraîchir.

On change ensuite les propriétaires des schémas afin qu'il s'agisse bien de **formation_admin** et non de **aicha** ou **postgres**.

12.6. Création des droits

Nous allons changer le propriétaire de la base formation. La commande SQL qui nous permettrait d'effectuer cette opération est :

```
ALTER DATABASE formation OWNER TO formation_admin;
```

- *Retrouvez le moyen de le faire graphiquement dans pgadmin.*
- *Est-il possible de modifier le nom d'une base de données après sa création?*

Nous allons maintenant appliquer une politique de droits en deux étapes qui correspondent aux deux onglets de droits présentés par pgadmin sur les objets:

- les droits généraux d'accès aux objets
- les droits sur les éléments du langage SQL

- *L'écriture des droits d'accès se fais en choisissant propriétés sur un clic droit sur la base formation.*

L'ergonomie de cet écran n'est pas évidente au premier abord. Il faut choisir un groupe en bas, cocher ses droits puis l'ajouter. Essayez d'obtenir exactement le même état que dans cette copie d'écran puis validez.

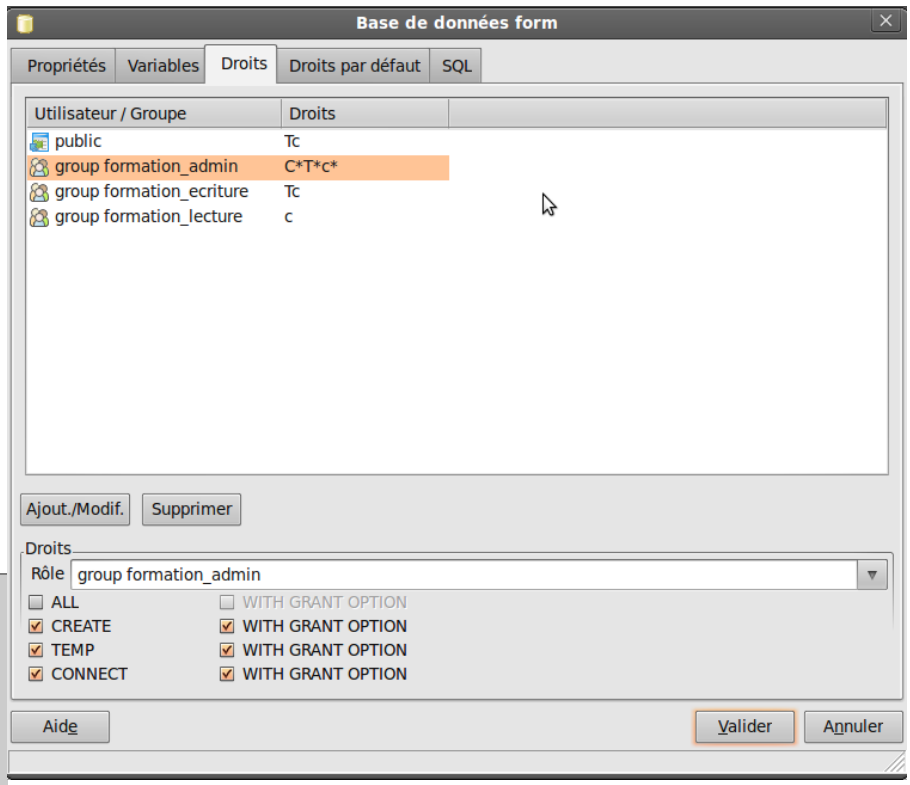


Illustration 9: Droits d'accès à la base par rôle groupe

Quand vous sélectionnez l'objet 'formation' dans la liste des bases de données vous pouvez remarquer que le panneau SQL (en bas à droite) dans pgadmin vous liste les commandes SQL qui permettent de reconstruire cet objet dans l'état où il se situe actuellement. Une fois que vous aurez appliquer les droits sur la base vous devriez retrouver cette liste de commandes SQL sous la commande CREATE DATABASE:

```
GRANT CONNECT, TEMPORARY
ON DATABASE formation TO
public;
GRANT ALL ON DATABASE
formation TO
formation_admin WITH
GRANT OPTION;
GRANT CONNECT, TEMPORARY
ON DATABASE formation TO formation_ecriture;
GRANT CONNECT ON DATABASE formation TO formation_lecture;
```

Ré-ouvrez les propriétés de la base et allez dans l'onglet Droits par défaut.

Attention: cet onglet est constitué de trois sous onglets.

Essayez d'obtenir l'équivalent de ces trois copies d'écrans

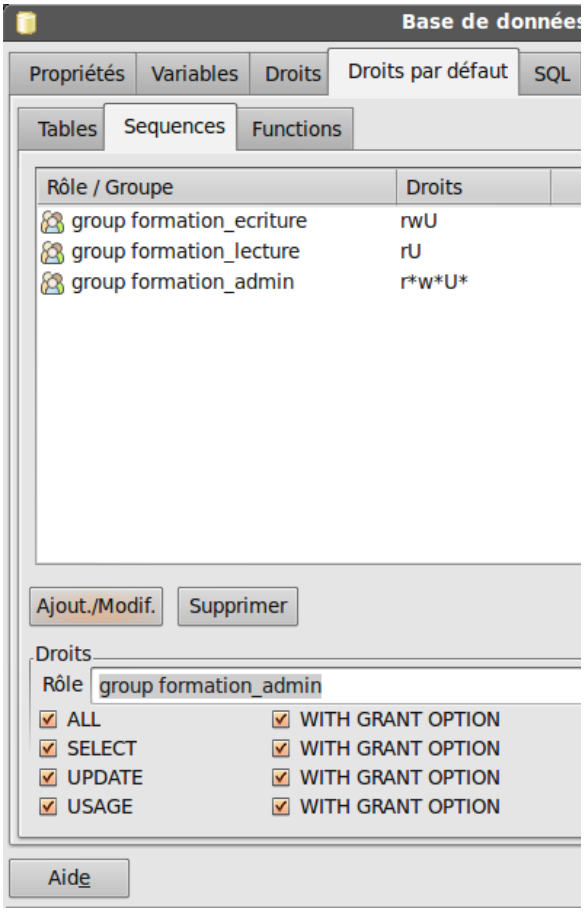


Illustration 10: Droits par défaut onglet Séquences

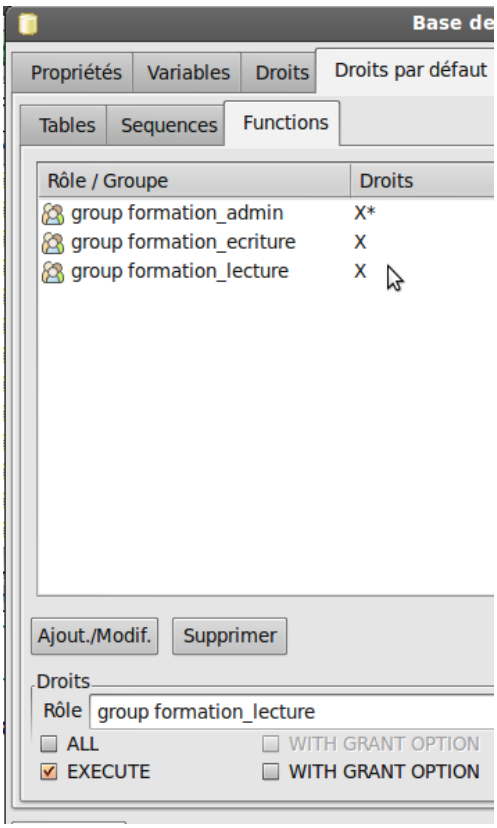


Illustration 11: Droits par défaut onglet fonctions

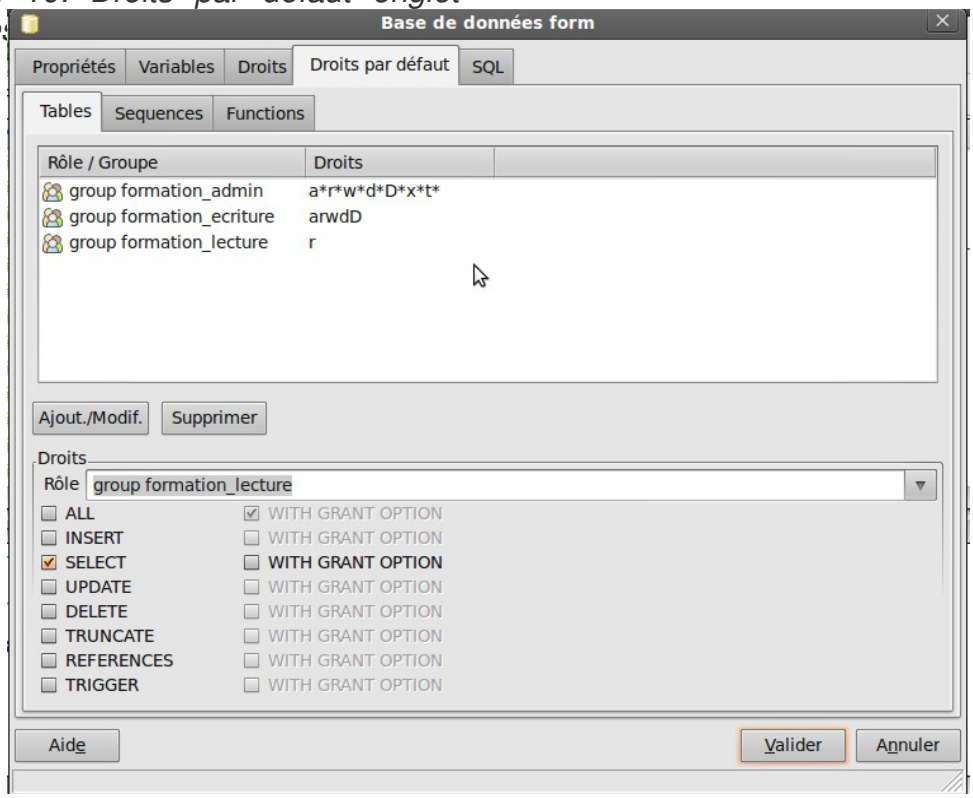


Illustration 12: Droits par défaut onglet Tables

Le résultat en terme de SQL devrait être:

```
ALTER DEFAULT PRIVILEGES
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES,
TRIGGER ON TABLES
    TO postgres;

ALTER DEFAULT PRIVILEGES
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE ON TABLES
    TO formation_ecriture;

ALTER DEFAULT PRIVILEGES
    GRANT SELECT ON TABLES
    TO formation_lecture;

ALTER DEFAULT PRIVILEGES
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES,
TRIGGER ON TABLES
    TO formation_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES
    GRANT SELECT, UPDATE, USAGE ON SEQUENCES
    TO postgres;

ALTER DEFAULT PRIVILEGES
    GRANT SELECT, UPDATE, USAGE ON SEQUENCES
    TO formation_ecriture;

ALTER DEFAULT PRIVILEGES
    GRANT SELECT, USAGE ON SEQUENCES
    TO formation_lecture;

ALTER DEFAULT PRIVILEGES
    GRANT SELECT, UPDATE, USAGE ON SEQUENCES
    TO formation_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES
    GRANT EXECUTE ON FUNCTIONS
    TO public;

ALTER DEFAULT PRIVILEGES
    GRANT EXECUTE ON FUNCTIONS
    TO postgres;

ALTER DEFAULT PRIVILEGES
    GRANT EXECUTE ON FUNCTIONS
    TO formation_ecriture;

ALTER DEFAULT PRIVILEGES
    GRANT EXECUTE ON FUNCTIONS
    TO formation_lecture;

ALTER DEFAULT PRIVILEGES
    GRANT EXECUTE ON FUNCTIONS
    TO formation_admin WITH GRANT OPTION;
```

Il nous faut ensuite appliquer certains droits d'accès au niveau des schémas:

L'équivalent en SQL est:

```
GRANT USAGE ON SCHEMA drh TO GROUP formation_drh;
GRANT USAGE ON SCHEMA app TO GROUP formation_app;
```

12.7. La variable search_path

Plusieurs variables sont associées à la session de connexion à la base. Certaines sont définies lors de l'ouverture de la session (instruction `SET mavariable='foo';`), certaines sont définies au niveau du rôle, ou au niveau de la base. Une des variables les plus importantes à partir du moment où l'on travaille avec les schémas est `search_path`. On peut voir cette variable comme un équivalent des variables d'environnement `PATH` sur les divers système d'exploitation.

La valeur par défaut de `search_path` est `"$user",public`

Cela signifie que lorsque une instruction SQL recherchera un objet qui n'a pas été préfixé par un nom de schéma il recherchera par défaut cet objet dans un schéma portant le nom de l'utilisateur puis dans le schéma public.

Ainsi si l'utilisateur postgres écrit:

```
SELECT * FROM matable;
```

Le moteur SQL va tenter: de trouver une table `matable` dans le schéma `postgres`, et si elle n'existe pas il recherchera cette table dans le schéma public. Toutes les bases de données ont au moins un schéma `public`.

Si cette même requête est lancée par l'utilisateur bob `matable` sera recherchée dans le schéma `bob` puis dans le schéma `public`. Le fait que le schéma bob n'existe pas ne va pas générer d'alerte.

Pour être certain de manipuler le bon objet, ou pour trouver un objet qui est dans un schéma qui n'est pas listé dans `search_path` il faut préfixer le nom de l'objet par le nom du schéma, comme dans cette requête:

```
SELECT sylvie.calcul_differentiel(foo.field1,foo.field2)
FROM marco.tablefoo foo;
```

Dans cet exemple on voit que l'objet recherché peut-être une table mais aussi une fonction ou tout autre objet appartenant à un schéma.

On peut aussi modifier la valeur de `search_path` afin que la recherche puisse tester les bons schémas dans le bon ordre de priorité. Un examen du code SQL généré par un pg_dump de type PLAIN montre par exemple une utilisation intensive des instructions SET `search_path` pour fixer en fait les schéma de travail en cours.

12.8. Tester les droits et schémas

Muni de nos différentes connexions et de nos schémas nous allons pouvoir tester nos droits d'accès Il nous manque cependant encore deux éléments, des tables et des données afin de vérifier que nous pouvons lire et/ou écrire dans ces tables.

Dans un premier temps, avant d'importer des bases réelles nous allons apprendre à créer très vite des jeux de test.

12.8.1. Création table test1 en SQL dans le schéma public

Nous utiliserons la connexion **rouge** de postgres. Nous cliquons sur le schéma public puis sur le bouton SQL:

```
CREATE TABLE test1 (
 id serial,
 val character varying,
```

```
 PRIMARY KEY(id)
);
```

Nous obtenons quelques notices:

```
NOTICE:  CREATE TABLE will create implicit sequence "test1_id_seq" for
serial column "test1.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"test1_pkey" for table "test1"
La requête a été exécutée avec succès en 111 ms, mais ne renvoie aucun
résultat.
```

Un **F5** dans pgadmin nous montre cette table dans la liste des tables du schéma public. Nous voyons aussi la séquence qui a été créée.

☑ Le mot clef **serial** ici est en fait une macro. Il représente un type integer (ou big si on utilise **bigserial**). Il impose la création automatique d'une **séquence**, que nous aurions pu créer manuellement, et impose la valeur par défaut de notre integer à être `nextval(la-sequence)`. Une séquence est un objet que nous pouvons examiner dans pgadmin, son but est de fournir des incréments de façon transactionnelle (pas de doublon) et la fonction `nextval()` sur une séquence fournit cette nouvelle valeur. Serial et bigserial sont donc les mots clefs de PostgreSQL pour les **auto incréments**.

☑ Nous pourrions définir nous-même notre auto-incrément en créant la séquence, un champ de type integer avec une valeur par défaut prenant `nextval()` de notre séquence. Mais pourquoi se fatiguer? Ne faites cela vous-même que si vous voulez partager une séquence entre plusieurs tables

Pour alimenter cette table **test1** nous allons utiliser une requête de type `INSERT`. Mais tout d'abord nous pouvons remarquer qu'il n'y a pas besoin d'une table pour faire une requête `SELECT`:

```
SELECT 42;
```

Nous renvoie le résultat 42.

Plus utile, regardons une fonction PostgreSQL nommée `generate_series`:

```
SELECT generate_series(1,999);
```

Qui nous renvoie 999 lignes de résultat, avec une série de nombre allant de 1 à 999. Si nous voulons générer 999 chaînes de caractères différentes nous pouvons utiliser l'opérateur de concaténation de chaînes: « || »:

```
SELECT 'test' || generate_series(1,999);
```

Voilà qui nous donne 999 valeurs intéressantes à insérer dans notre table de test.

☑ Nous venons de concaténer une chaîne et un nombre, PostgreSQL s'en est sortit poliment sans nous réprimander. Ce ne sera pas toujours le cas, il faut souvent penser à faire des **cast,** des **conversion de types**. Si PostgreSQL nous avait demandé de manipuler des objets de même type nous aurions pu forcer la valeur avec des fonctions de conversion ou avec le raccourci « ::<nom du type ». Ce qui aurait donné : `'test' || generate_series(1,999)::text`

Il faudrait aussi créer les valeurs de la colonne « **id** » mais il s'agit d'un auto-incrément qui va donc se remplir tout seul si nous ne mettons rien (ou si nous mettons NULL):

```
INSERT INTO test1(val) VALUES ('test' || generate_series(1,999));
```

Vous pouvez vérifier avec `SELECT * from test1;` que les valeurs sont bien saisies. Vous pouvez aussi utiliser un double clic ou l'icône de visualisation des tables depuis la liste des tables.

> ☑ Remarquez que depuis la fenêtre de visualisation des tables de pgadmin vous pouvez effectuer des saisies/modifications de données **SI la table contient une clef primaire**. La clef primaire permet en effet à pgadmin l'écriture de requêtes de mises à jour ou d'insertions.

12.8.2. Vérification de l'application des droits par défaut

Si nous regardons le code SQL qui permet de recréer l'objet test1 dans l'écran pgadmin nous voyons:

```
-- DROP TABLE test1;

CREATE TABLE test1
(
  id serial NOT NULL,
  val character varying,
  CONSTRAINT test1_pkey PRIMARY KEY (id)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE test1 OWNER TO postgres;
GRANT ALL ON TABLE test1 TO postgres;
GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE test1 TO
formation_ecriture;
GRANT SELECT ON TABLE test1 TO formation_lecture;
GRANT ALL ON TABLE test1 TO formation_admin WITH GRANT OPTION;
```

> ☑ Nous pouvons remarquer que la première ligne est en vert, en effet « **--** » est la marque du **commentaire** en SQL.

Nous voyons aussi que les **privilèges par défaut** définis au niveau de la base de données ont été redescendus au niveau de la table nouvellement créée.

Si nous utilisons la connexion **orange** de aicha, que nous choisissions la base de donnée et que nous créons une deuxième table de test:

```
CREATE TABLE public.test2 (
 id serial,
 val character varying,
 PRIMARY KEY(id)
 );
```

Nous pouvons aller regarder cette table sur pgadmin, nous voyons alors que seule une ligne de `GRANT` a été ajoutée:

```
ALTER TABLE test2 OWNER TO aicha;
```

☑ Les **privilèges par défaut n'ont pas été redescendus**. Si vous essayez d'aller sur cette table avec une autre connexion (comme celle de martine) vous ne pourrez pas y accéder.

Les privilèges par défaut sont en fait enregistrés sur le rôle postgres!

☑ Si vous relisez le code produit par pgadmin après l'écriture des droits par défaut, il ne s'agit pas de droits par défauts appliqués à la base mais de droits par défaut appliqués à un rôle, il y a une valeur par défaut FOR rôle xxx qui n'est pas montrée, cette commande **s'applique aux nouveaux objets créés par ce rôle et pas globalement à la base elle-même**.

```
ALTER DEFAULT PRIVILEGES
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE ON TABLES
    TO formation_ecriture;
```

Signifie en fait:

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE ON TABLES
    TO formation_ecriture;
```

Pour que ces privilèges par défaut s'appliquent avec notre utilisateur aicha, qui est pourtant dans le groupe propriétaire de la base, **nous devrons corriger le SQL de définition de la base**. Pour cela nous utiliserons un script **SQL**, mais dans l'immédiat nous allons continuer à créer nos objets avec le compte postgres (rouge). Pour le moment nous allons donc supprimer cette table créé par aicha. Soit avec un clic droit sur l'objet soit en tapant directement la commande SQL:

```
DROP table test2;
```

12.8.3. Création table test2 dans le schéma drh avec pgadmin

Toujours avec la connexion de l'utilisateur **postgres** nous allons créer une deuxième table mais en utilisant l'assistant de création de table. Choisissez le schéma **drh**, faites un clic droit, `Ajouter un objet > Ajouter une table...`

Il faut créer une table **test2**, avec une colonne **id** en type **serial** (tout en bas dans la liste des types), une colonne **val** en type character varying et une **primary key** sur la colonne id. Vérifiez avec l'onglet SQL que vous obtenez:

```
CREATE TABLE drh.test2
(
   id serial,
   val character varying,
   CONSTRAINT "PRIMARY KEY" PRIMARY KEY (id)
)
WITH (
  OIDS = FALSE
);
```

Le nom du schéma a été préfixé au nom de la table, ce qui n'était pas fais sur la table test1 puisqu'elle était dans le schéma par défaut (public).

Remplissons cette table avec 10000 valeurs, commençons par:

```
INSERT INTO test2(val) VALUES ('foo' || generate_series(1,10000));
```

nous obtenons:

```
ERROR: relation "test2" does not exist
```

La table test2 n'est pas retrouvée car le schéma **drh** n'appartient pas au **search_path**, il nous faut donc le forcer:

```
INSERT INTO drh.test2(val) VALUES ('foo' || generate_series(1,10000));
```

12.8.4. Création table test1 dans le schéma app en SQL avec le search_path

Revenons à l'éditeur de SQL et tapons:

```
SET search_path=app,public,drh;
CREATE TABLE test1 (id integer, val character varying, PRIMARY KEY
(id));
```

On constate que la table est créée dans le premier schéma défini dans **search_path**, donc dans le schéma « **app** ». Vérifiez que le propriétaire de la table est bien formation_admin.

Maintenant que la variable search_path contient notre schéma app et le schéma drh, si nous tapons

```
INSERT INTO test2(val) VALUES ('bar' || generate_series(10001,20000));
```

Nous devrions réussir à ajouter 10 000 enregistrements supplémentaires dans la table **test2** du schéma **drh**, elle n'a pas été trouvée dans app mais elle existe bien dans le deuxième schéma listé.

Essayons de remplir la table test1 du schéma app. Si nous ne préfixons pas la table nous devrions tomber sur la test1 du schéma app qui est prioritaire sur le schéma public dans notre search path:

```
INSERT INTO test1(val) VALUES ('nii' || generate_series(1,10));
```

Si « tout va bien » nous obtenons alors une erreur:

```
ERROR: null value in column "id" violates not-null constraint
```

Si vous regardez bien la définition de la table test1 du schéma app, nous n'avons pas utilisé le type **serial** mais un simple **integer** pour la clef primaire. Il n'y a donc pas d'auto-incrément. Il nous faut remplir la colonne id lors des insertions. Essayons d'utiliser generate_series:

```
SELECT generate_series(1,10), 'nii' || generate_series(1,10);
```

Nous donne **110 lignes de résultats avec postgreSQL 9.0 (apparement PostgreSQL 9.1 supporte cette syntaxe par contre)**. Ce n'est pas le bon chemin. Essayons d'utiliser notre série de donnée automatique comme une table sur laquelle on fait une requête:

```
SELECT serie
FROM generate_series(1,10) serie;
```

Cela fonctionne. On va donc faire une requête un petit peu plus complexe:

```
SELECT serie, 'nii' || serie
FROM generate_series(1,10) serie;
```

On obtient un résultat de deux colonnes qui ressemble à ce que l'on voudrait insérer dans test1. Il est tout à fais possible d'utiliser le résultat d'un requête comme valeurs à insérer avec INSERT, la syntaxe est `INSERT INTO table (col1,col2...) SELECT ….`; et non `INSERT INTO table (col1,col2...) VALUES ( SELECT ….);`

```
INSERT INTO test1(id,val) SELECT serie, 'nii' || serie FROM
generate_series(1,10) serie;
```

Testons que cela a fonctionné:

```
select count(*) from test1;
```

devrait renvoyer **10**

et nous devrions avoir **999** pour

```
select count(*) from public.test1;
```

car public.test1 est la table test1 dans le schéma public.

12.8.5. Régler search_path, les variables

Les différents utilisateurs auront des besoins différents en terme d'accès aux schémas. On va donc régler leur search_path au niveau des «rôles » de cette façon:

- **aicha**: search_path=public,app,drh
- **nicolas**: search_path=drh,public
- **sebastien** : search_path=app,drh,public
- **martine**: search_path=app,public
- **dominique**: search_path=app,public
- **francois**: search_path=app,public

Pour cela nous utiliserons pgadmin en cliquant (droit) sur les **rôles** et en allant dans l'onglet **variables**:

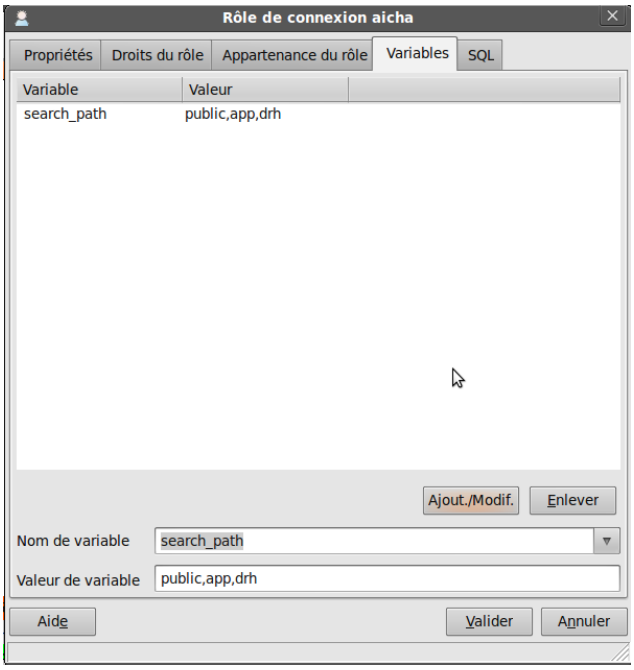


Illustration 13: Changer la variable search_path pour aicha

On voit que le SQL généré est du type:

```
ALTER ROLE dominique SET search_path=app, public;
```

Pour vérifier la valeur de cette variable pour chacune des connexions on peut ouvrir une fenêtre SQL et y taper:

```
SHOW search_path;
```

> ☑ A noter: il n'est pas possible de définir cette valeur au niveau d'un « groupe rôle », seuls les GRANT (droits) sont hérités d'un groupe, pas les variables.

Remarquez aussi qu'il existe de nombreuses variables, comme celles définissant les types utilisables pour la saisie des dates (**date_style**).

> ☑ Les variables peuvent se définir au niveau des rôles, mais elles peuvent aussi être définies au niveau des connexions à la base de données avec les commandes `SET variable=value`. On retrouvera ce type de commandes dans les programmes objet pour initialiser la communication entre le serveur et l'application.
>
> ☑ Depuis PostgreSQL 9 une variable nouvelle est apparue: **application_name**, vous devriez utiliser cette variable dans votre application, ceci permet d'identifier les connexions de votre application parmi les centaines de connexions ouvertes, elle est par exemple utilisée par pgadmin, cliquez sur `Outils>Etat du serveur` et regardez la liste des connexions ouvertes, certaines ont un nom.

12.8.6. Tests d'accès

Si vous aviez des connexions serveurs ouvertes dans pgAdmin il faut les fermer et les rouvrir pour que les variables sont bien initialisées (attention, pas simplement les replier et déplier, il faut choisir « se déconnecter »). Nous allons taper une série de requêtes simples sur les tables test1 et test2 afin de vérifier pour chaque utilisateur:

- les accès en lecture et les priorités de schémas
- les accès en écritures

Pour tester les requêtes ouvrez une des connexions, choisissez la base formation puis le bouton SQL.

➢ *essayez de deviner les raisons avant que le formateur n'explique*

```
SELECT COUNT(*) FROM test1;
```

- aicha : **999**
- nicolas: **999**
- sebastien: **10**
- francois: **10**
- martine: **10**

```
SELECT COUNT(*) FROM test2;
```

- aicha : **20000**
- nicolas: **20000**

- sebastien: **20000**
- francois: **ERROR**: permission denied for relation test2
- martine: **ERROR**: permission denied for relation test2

```
INSERT INTO test1 (val) VALUES ('ins1');
```

- aicha : **OK**
- nicolas: **OK**
- sebastien: **ERROR**: permission denied for relation test1
- francois: **ERROR: null value** in column "id" violates not-null constraint, francois travaille sur app.test1 qui n'a pas de clef primaire. `INSERT INTO test1 (id,val) VALUES (11,'ins1');` devrait fonctionner par contre. Exécuter cette deuxième instruction deux fois devrait lever une nouvelle erreur: **ERROR: duplicate key** value violates unique constraint "test1_pkey"
- martine: **ERROR**: permission denied for relation test1

Certains utilisateurs comme **francois** et **nicolas** ont donc le droit d'insérer des données dans des tables. Vérifions cependant que seul **nicolas** est autorisé à insérer des données dans le schéma **drh**:

```
INSERT INTO drh.test2 (val) VALUES ('ins2');
```

francois obtient le message d'erreur:

```
ERROR: permission denied for schema drh
```

➢ *Vous pouvez tester des instructions DELETE, UPDATE, TRUNCATE, les droits sont normalement bien appliqués. De même si nous tentons de modifier la structure d'une table:*

```
ALTER TABLE public.test1 ADD COLUMN foo integer;
```

Tout le monde, en dehors **d'aicha**, aura ce message d'erreur:

```
ERROR: must be owner of relation test1
```

12.9. DDL DML et DCL : et gestion avancée des droits

Dans le language SQL on distingue en fait trois grandes familles de commandes. Quand vous pensez à régler les droits d'accès il s'agit le plus souvent de gérer qui à accès à ces grandes familles.

- **DDL** Data Definition Language : Tout ce qui permet de définir et de modifier (voir de supprimer) la structure des objets de votre base de donnée. Il s'agit clairement d'un niveau administrateur (`CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `COMMENT`, etc)
- **DML** : Data Manipulation Language : les commandes de manipulation des données au sein de cette structure (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `EXPLAIN`, `LOCK`, etc). Attention dans cette famille il y a des commandes en écriture (beaucoup), et une seule commande nécessaire pour un accès en lecture (`SELECT`)

Il existe en fait théoriquement d'autres familles comme:

- **DCL** : Data Control Language : La gestion des droits (`GRANT`, `REVOKE`)
- **TCL**: Transaction Control : les commandes de gestion des transaction(`BEGIN`, `COMMIT`, `ROLLBACK`, etc)

Dans la réalité d'une politique de droits on retrouvera très souvent:

- un niveau **administrateur**, responsable des créations et de l'application des droits.
- Un niveau avec tous les droits en **écriture** (DML+TCL) mais aucun accès au DDL ou au DCL
- Un niveau en **lecture** seule (uniquement `SELECT` dans le DML)

Mais on peut vouloir aller plus loin, la politique de droits disponibles dans postgreSQL permet d'aller assez loin dans le niveau de finesse. Si vous ouvrez la définition d'un table dans pgadmin, que vous sélectionnez une colonne et cliquez sur « **Modifier** » vous pouvez voir un onglet **droits**, dans lequel vous pouvez restreindre les droits d'accès à cette colonne (`INSERT`, `UPDATE`, `REFERENCES`, `SELECT`) par rôle (ou groupe).

Il existe en théorie un niveau de droit que l'on retrouve par exemple sous l'appellation « Virtual Private Database » et qui consiste à avoir des **politiques d'accès au niveau des lignes des tables**:

Pour masquer partiellement certaines lignes de tables avec PostgreSQL il faudrait utiliser des modules complémentaires comme le projet Veil (http://veil.projects.postgresql.org/curdocs/main.html). Ou des règles faites à la main (nous verrons les règles un peu plus loin)

13. Premières sauvegardes et restaurations

Nous avons pu constater que toutes les commandes effectuées dans PostgreSQL avec pgAdmin correspondaient à des **commandes SQL**. On peut donc à tout moment sauvegarder l'état actuel d'une base de données, autant au niveau de la **structure** que du **contenu**.

Les sauvegardes vont principalement avoir deux rôles:

- **permettre de récupérer l'état de la base** en cas de problème technique ou fonctionnel, la vision classique de la sauvegarde pour un administrateur
- permettre de **partager la base** entre développeurs, de créer des fichiers **d'installation** et de population de la base, vision plus utile au développeur.

Pour l'administrateur la vision que nous allons étudier dans un premier temps, le dump, est une première solution, ce n'est pas la seule et nous y reviendrons plus tard. Pour le développeur le dump devrait devenir un élément prioritaire.

13.1. pg_dump : obtenir un script SQL de recréation de la base

Un dump de base de données est un script SQL qui permet de reconstruire une base et d'y réinsérer toutes les données.

La commande utilisée pour effectuer un dump sous PostgreSQL est **`pg_dump`**. Il s'agit d'une ligne de commande qui accepte un très grand nombre d'options. PgAdmin propose un assistant de sauvegarde utilisant `pg_dump` et `pg_restore` qui va nous permettre d'effectuer très simplement un grand nombre de types de dumps différents.

Nous allons travailler avec l'utilisateur **postgres (rouge)**, qui n'aura aucun problèmes de droits pour sauvegarder la base.

Si nous effectuons un clic droit sur la base de données formation nous voyons un assistant « **sauvegarder** ». Le but de cet assistant est de lancer un commande `pg_dump` avec les options que vous cochez. Cet assistant n'est pas parfait en terme d'ergonomie, certaines options sont incompatibles sans que vous ne puissiez le deviner et la forme finale du dump dépends très fortement des options. Nous allons tester quelques combinaisons.

Nous commençons par ne travailler que sur les formats « **PLAIN** ». Ce format génère du **SQL**. Nous donnons donc l'extension « **.sql** » au fichier de sortie

- **formation_test1.sql** : PLAIN + codage UTF8 + inclure l'instruction CREATE DATABASE: `pg_dump --host localhost --port 5432 --username "postgres" --format plain --create --encoding UTF8 --verbose –file "(...)/formation_test1.sql" "formation"`
- **formation_test2.sql** : PLAIN + codage UTF8 + inclure l'instruction DROP Database + utiliser les colonnes pour les INSERT + utiliser des commandes INSERT: `pg_dump --host localhost --port 5432 --username "postgres" --format plain --clean --encoding UTF8 --inserts --column-inserts --verbose --file "(...)/formation_test2.sql" "formation"`
- **formation_test3.sql:** PLAIN + codage UTF8 + inclure l'instruction DROP Database + schéma uniquement :
  `pg_dump --host localhost --port 5432 --username "postgres" --format plain --schema-only --clean --encoding UTF8 --verbose --file "(...)/formation_test3.sql" "formation"`
- **formation_test4.sql:** PLAIN + codage UTF8 + schéma uniquement : `pg_dump --host localhost --port 5432 --username "postgres" --format plain --schema-only --encoding UTF8 --verbose --file "(...)/formation_tests_4.sql" "formation"`

Si nous comparons le contenu des ces fichiers avec un éditeur de texte on peut remarquer plusieurs choses:

- **formation_test1.sql** :il s'agit d'un script destiné à être lu par la ligne de commande psql, il comporte des instructions spécifiques propres à psql comme toutes celles qui commencent par « \ ». L'insertion des données se fait en utilisant l'option `-f` dans psql plutôt que par des instructions SQL (`commande COPY test1 (id, val) FROM stdin`). Ce script présuppose que la base de donnée formation n'existe pas, il faut donc supprimer une éventuelle base avant de l'exécuter. Nous pourrions l'exécuter très simplement ainsi:

```
psql -U postgres -h localhost -d formation -f
/path/to/formation_test1.sql
```

- **formation_test2.sql** ne contient aucune commande destinées à psql, c'est le format idéal pour être rejoué dans pgadmin (fenêtre SQL). Il ne contient pas de commande de création de base de donnée mais supprime tous les objets et droits puis les recrée dans l'ordre. Notez que l'ordre de suppression et création des objets est très complexe, il est quasi impossible d'effectuer à la main un script qui saurait organiser les suppressions et recréation en tenant compte de toutes les dépendances entre objets (remarquez au passage l'onglet dépendances sur tous les objets dans pgAdmin). Notez que des insertions sous forme de requêtes `INSERT` sont plus lentes qu'avec le premier fichier (`COPY`). Remarquez aussi que l'instruction `inclure DROP DATABASE` dans l'assistant aurait du être nommée `inclure DROP des objets de la base` (il n'y a pas de `DROP DATABASE`!!).
- **formation_test3.sql** est quasi équivalent au deuxième fichier sauf qu'il ne contient pas les données. Notez que l'instruction `ne sauvegarde que le schéma` aurait du être traduite par `ne sauvegardez que la structure` car cela ,n'est pas en rapport avec les objets de type schéma. Si vous voulez ne sauvegarder qu'un schéma, voir un ensemble de tables vous pouvez voir qu'un des onglets de l'assistant permet de choisir ces objets.

– **formation_test4.sql** n'avait ni '`DROP DATABASE`' ni '`CREATE DATABASE`' en options, nous obtenons un script d'initilisation qui peut tourner sur une base vierge et qui créé tous les objets, ici sans les données. C'est une bonne alternative au formation_test2.sql

Vous pouvez facilement imaginer un **formation_test4.sql** ne contenant que les données ou bien une sauvegarde ne contenant qu'un seul schéma ou une seule table.

☑ Si vous avez créé des objets supplémentaires dans vos bases depuis le dump ils ne sont pas pris en compte dans des dumps de suppression/recréation. Ce qui empêchera au passage le dump de se rejouer car les schémas ne seront pas vides. Changez alors les instructions `DROP SCHEMA <nom du schéma>` par `DROP SCHEMA <nom du schema> CASCADE` dans le script SQL, tous les objets supplémentaires inclus dans le schéma seront supprimés. Ou bien utilisez un dump comme le quatrième, sans les suppressions, et supprimez votre base avant import (cf partie suivante sur les droits pour les GRANT d'accès)

☑ N'oubliez pas qu'il n'y a pas que le format PLAIN, nous étudierons dans les parties administration les autres formats de backup, y compris ceux de pg_dump

☑ Les assistants de restauration de pgAdmin sont prévus pour les autres formats (pas PLAIN), car les format PLAIN se rejoue simplement dans les écrans SQL.

Une fois ces fichiers créés rien ne vous interdit de les modifier pour y ajouter quelques commandes SQL. Ainsi nous pourrions ajouter nos trois commandes de remplissage de tables à la fin de formation_test3.sql ou formation_test4.sql.

13.2. Problèmes avec la gestion des droits, élévation de privilèges

Si vous recherchez toutes les instructions GRANT dans ces fichiers de dump vous verrez que certaines sont absentes:

```
GRANT CONNECT, TEMPORARY ON DATABASE formation TO public;
GRANT ALL ON DATABASE formation TO formation_admin WITH GRANT OPTION;
GRANT CONNECT, TEMPORARY ON DATABASE formation TO formation_ecriture;
GRANT CONNECT ON DATABASE formation TO formation_lecture;
```

Il s'agit du contenu du premier onglet `Droits` sur la base de donnée elle-même (les droits sur les schémas font eux partie du dump). **Ceci est du au fait que ces droits qui gère un premier niveau d'accès à la base (globalement le droit de s'y connecter) ne sont pas gérés au sein de la base elle-même mais au sein de la base postgres**. Il existe un deuxième utilitaire `pg_dumpall` qui sauve l'intégralité du cluster de base de donnée, ces droits inclus.

A l'usage il est plutôt conseillé de faire un fichier SQL dédié aux droits d'accès et d'y ajouter ces quelques lignes. La gestion des droits d'accès à la base n'étant pas supposée changer trop souvent. L'autre solution est l'utilisation de `pg_dumpall --globals-only` qui comme son nom l'indique se contente de sauvegarder les « globales » du cluster parmi lesquels on retrouve le rôles et les droits primaires.

Si vous regardez à la fin de ces fichiers vous y verrez toute une série de `ALTER DEFAULT PRIVILEGES` effectués sur le rôle postgres:

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres REVOKE ALL ON SEQUENCES
FROM PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres REVOKE ALL ON SEQUENCES
FROM postgres;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres GRANT ALL ON SEQUENCES  TO
postgres;
```

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres GRANT ALL ON SEQUENCES  TO
formation_ecriture;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres GRANT SELECT,USAGE ON
SEQUENCES  TO formation_lecture;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres GRANT ALL ON SEQUENCES  TO
formation_app WITH GRANT OPTION;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres REVOKE ALL ON FUNCTIONS
FROM PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres REVOKE ALL ON FUNCTIONS
FROM postgres;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres GRANT ALL ON FUNCTIONS  TO
postgres;
(etc.)
```

Si nous avions défini des droits par défaut dans le schéma app nous aurions eu des lignes supplémentaires du type

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres IN SCHEMA app REVOKE ALL ON
TABLES  FROM PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres IN SCHEMA app REVOKE ALL ON
TABLES  FROM postgres;
ALTER DEFAULT PRIVILEGES FOR ROLE postgres IN SCHEMA app GRANT
SELECT,INSERT,DELETE,TRUNCATE,UPDATE ON TABLES  TO formation_ecriture;
(etc.)
```

Il s'agit de la définition des `Droits par défaut` dans pgAdmin.

Quand nous avons créé ces privilèges par défaut **pgadmin a décidé de ne les créer que pour l'utilisateur postgres**.

Quand pgadmin nous montre une ligne de droits ainsi:

```
ALTER DEFAULT PRIVILEGES
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE ON TABLES
    TO formation_ecriture;
```

Cela signifie en fait:

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres
    GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE ON TABLES
    TO formation_ecriture;
```

C'est pourquoi quand nous créons un objet avec l'utilisateur **aicha** celui-ci n'hérite pas des droits par défaut. **Si vous voulez que l'utilisateur aicha créé des objets avec les droits par défaut il faut donc faire une copie de toutes ces lignes ALTER DEFAULT PRIVILEGES, y remplacer le mot 'postgres' par 'aicha' (ou formation_admin).**et ajouter ces lignes soit dans le fichier de dump, soit dans un fichier à part (par exemple un dump dédié à la gestion des droits**.**

Essayons avec `formation_admin` :

```
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin REVOKE ALL ON
SEQUENCES  FROM PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin REVOKE ALL ON
SEQUENCES  FROM formation_admin;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
SEQUENCES  TO postgres;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
SEQUENCES  TO formation_ecriture;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT SELECT,USAGE
ON SEQUENCES  TO formation_lecture;
```

```
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
SEQUENCES  TO formation_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin REVOKE ALL ON
FUNCTIONS  FROM PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin REVOKE ALL ON
FUNCTIONS  FROM formation_admin;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
FUNCTIONS  TO PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
FUNCTIONS  TO postgres;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
FUNCTIONS  TO formation_ecriture;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
FUNCTIONS  TO formation_lecture;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON
FUNCTIONS  TO formation_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin REVOKE ALL ON TABLES
FROM PUBLIC;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin REVOKE ALL ON TABLES
FROM formation_admin;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON TABLES
TO postgres;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT
SELECT,INSERT,DELETE,TRUNCATE,UPDATE ON TABLES  TO formation_ecriture;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT SELECT ON
TABLES  TO formation_lecture;
ALTER DEFAULT PRIVILEGES FOR ROLE formation_admin GRANT ALL ON TABLES
TO formation_admin WITH GRANT OPTION;
```

Si nous tapons cette série de commandes sur la base et que nous testons la création d'une table avec aicha cela ne fonctionne toujours pas. Il faut en fait qu'au moment de la création de la table avec l'utilisateur aicha **celle-ci indique qu'elle va effectuer cette commande en tant que rôle formation_admin** en tapant au préalable:

```
SET ROLE formation_admin;
```

Nous pourrions donc aussi ajouter des altérations de droits par défaut pour le rôle aicha lui-même. Afin d'éviter d'avoir à taper cette **élévation de privilèges**.

> ☑ **Retenez donc que les droits par défaut ne sont pas associés à la base mais à un rôle, et que par défaut ce FOR rôle xxx est attribué à l'utilisateur de la session qui lance ces commandes – donc souvent le rôle super-utilisateur.**
>
> ☑ **Retenez aussi que pour le rôle super-utilisateur sur la base il vaut mieux utiliser un rôle de connexion plutôt qu'un groupe. Sans quoi les rôles administrateurs devront avoir leurs commandes d'altération de droits par défaut individuelles ou utiliser des `SET ROLE` pour changer de niveau de session.**

Mais si demain nous ajoutons 5 utilisateurs dans le groupe `formation_admin` nous ne voudrions certainement pas avoir à retaper les altérations de privilèges par défaut pour chacun de ces rôles. Dans notre cas la meilleure solution est **de forcer la session ouverte avec l'utilisateur aicha à faire une élévation de rôle**. Nous avons

vue que cela se fait par une commande `SET`. Cela veut dire que **le rôle est une variable comme les autres**. Nous pouvons donc forcer la valeur de la variable rôle soit au **niveau de la session**, soit dans les **variables de l'utilisateur**, soit dans les **variables de l'utilisateur pour une base**. Nous avons déjà vu les deux premières méthodes, testons la troisième.

Pgadmin ne gère pas très bien les altérations de la variable `role` dans les onglets de variables. Nous utiliserons donc pgadmin pour nous montrer le début de la commande puis nous modifierons la requête pour l'adapter à nos besoins.

- *On commence par aller sur l'onglet `Variables` sur l'écran de propriétés de la base (avec l'utilisateur postgres).*
- *On utilise ensuite l'onglet SQL pour voir la requête générée et on décoche « Lecture seule »afin de pouvoir modifier la requête*

Illustration 15: on décoche lecture seule

- *On modifie la requête pour utiliser la variable rôle que pgadmin ne semble pas connaître. (Sur la gestion des utilisateurs pgadmin connait cette variable mais n'autorise que des numéros pour la valeur, on serait donc aussi obligé d'utiliser la modification manuelle de la requête)*

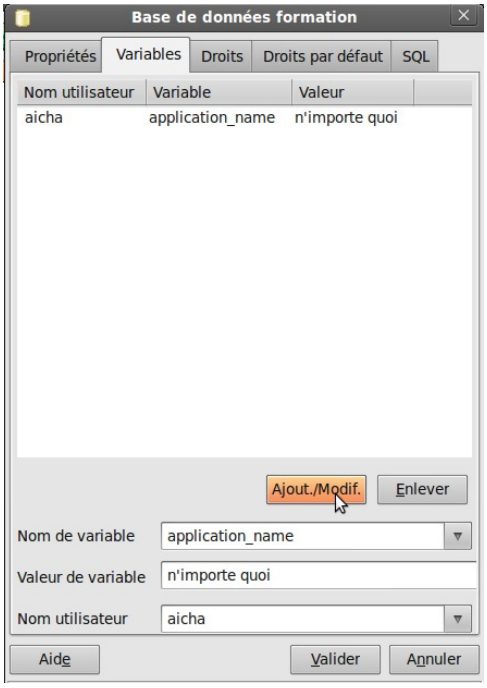


Illustration 14: Ajout d'une variable bidon sur la base formation pour aicha

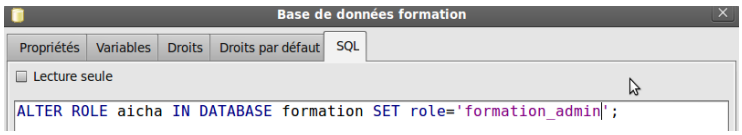


Illustration 16: modification manuelle de la requête

On obtient cette requête qui va normalement se situer avec les requêtes de GRANT de haut niveau dans le panneau SQL lorsque l'on sélectionne la base.

```
ALTER ROLE aicha IN DATABASE formation SET role='formation_admin';
```

Pour tester que cette altération de rôle fonctionne on peut se déconnecter puis se reconnecter avec `aicha` et taper cette commande SQL:

```
SHOW ROLE;
```

☑ Cette commande `ALTER ROLE` fais partie des `GRANT` qui ne feront pas partie d'un simple `pg_dump` de la table et qui devraient être sauvegardées en effectuant un `pg_dumpall --globals-only`.

- *Une fois le script corrigé en ajoutant une copie de toutes ces lignes pour l'utilisateur formation_admin nous allons le rejouer dans pgadmin en, ouvrant une connexion de l'utilisateur postgres, on se met sur la base formation, puis on ouvre l'éditeur SQL, Fichier, Ouvrir, on choisit notre dump corrigé puis on l'exécute.*

☑ Rassurez-vous à la prochaine sauvegarde les privilèges par défaut que nous avons ajoutés au rôle `formation_admin` n'auront pas disparu. Mais si les privilèges par défaut changent il ne faudra pas oublier de corriger le dump pour mettre `formation_admin` en correspondance avec `postgres`.

13.3. importation de la base de formation

Notre base formation avec ses trois schémas et ses trois tables est un peu légère. Nous allons la supprimer puis importer plusieurs fichiers de dumps qui contiennent une base plus avancée dans le développement de l'application.

> ☑ Nous aurions pu travailler avec un fichier unique de sauvegarde/import mais il est plus réaliste de commencer à travailler avec des fichiers multiples, tels qu'ils pourraient être gérés dans un vrai projet.

Les fichiers à disposition sont:

- **formation_creation.sql:** Ce fichier a été fais à la main et contient la commande de création de la base et les `GRANT` initiaux minimums, ceux qui ne peuvent être dumpés (recopiés depuis pgAdmin):

```
-- Adding default database creation and Database Grants
CREATE DATABASE formation
  WITH OWNER = formation_admin
       ENCODING = 'UTF8'
       TABLESPACE = pg_default
       LC_COLLATE = 'fr_FR.utf8'
       LC_CTYPE = 'fr_FR.utf8'
       CONNECTION LIMIT = -1;
ALTER ROLE aicha IN DATABASE formation SET role='formation_admin';
GRANT CONNECT, TEMPORARY ON DATABASE formation TO public;
GRANT ALL ON DATABASE formation TO formation_admin WITH GRANT OPTION;
GRANT CONNECT, TEMPORARY ON DATABASE formation TO formation_ecriture;
GRANT CONNECT ON DATABASE formation TO formation_lecture;
```

- **formation_initialisation.sql**: création des droits d'accès minimaux et des schémas Créé en faisant un dump en UTF8 + PLAIN + avec « Schéma uniquement », sans décocher aucun objet. Le dump obtenu a ensuite été vidé de 90% de son contenu, pour ne garder que les créations de schémas, et les grants sur ces schémas ainsi que les grants sur les rôles (les pg_dump partiels ne sont pas toujours capable de créer les schémas et de sauver les GRANTS par défaut)
- **formation_drh_public.sql** : tout le contenu du schéma drh, avec les données DRH et le schéma public (qui ne contient qu'une fonction). Créé avec un dump en UTF8+PLAIN+colonnes et commandes INSERT+ le schéma app décoché.
- **formation_schema_app_dev1.backup**: une première version de l'application, pour le schéma app, que nous installerons plus tard.
- **formation_schema_app_dev2.backup**: une deuxième version de l'application, pour le schéma app avec des fonctions et triggers, que nous étudieront plus tard aussi

Dans un premier temps nous n'allons étudier que le schéma `drh`. Il nous faut donc:

- *Utiliser une connexion administrateur, par `aicha` ou `postgres`*
- *Supprimer notre base formation actuelle (clic droit dans pgAdmin pour faire un `DROP DATABASE`)*
- *Ouvrir la fenêtre SQL en cliquant d'abord sur la base Postgres. Importer le contenu du fichier **formation_creation.sql** dans cette fenêtre. Si on tente de l'executer on obtient: `ERROR: CREATE DATABASE cannot be executed from a function or multi-command string`, pour exécuter de tells commandes dans*

*l'interpréteur SQL il y a une petite subtilité, il faut cliquer sur le `Menu Requêtes>Exécuter pgScript`. Remarquez aussi que si vous voulez **changer le nom de la base** il n'y aura qu'à le modifier à chaque ligne de ce fichier.*

- *Passons à l'initialisation des schémas et droits par défaut, sélectionnez la base formation qui a du être recréée (F5) puis ouvrez un interpréteur SQL.*
- *Ouvrez le fichier **formation_initialisation.sql** et exécutez le (normalement celui là).*
- *Nous pouvons alors passer au contenu du schéma drh. Ouvrez le fichier **formation_drh_public.sql** et exécutez-le. Vous devriez obtenir une erreur `ERROR: schema "drh" already exists` commentez cette ligne `CREATE SCHEMA drh;` avec deux tirets.*

13.4. Examen la base drh

Cette base est assez complexe.

Il y a de nombreuses choses à observer dans le fichier **formation_drh_public.sql** ou depuis l'interface de pgAdmin.

13.4.1. Types de données

Pour commencer vous pouvez observer les types utilisés sur les colonnes. De nombreux types sont disponibles http://docs.postgresqlfr.org/9.0/datatype.html et une base de données bien conçue devrait toujours utiliser le bon type pour la bonne donnée.

- *Parcourez les différents types de données proposés par postgreSQL*

☑ Pour les montants utilisez les types `numeric` et pas les `float`. Le type `money` existe aussi, mais il est assez pauvres en conversions de type et vous devriez éviter de l'utiliser

☑ Dans les types textes, remarquez en plus du `character varying` le type `text`.

☑ Pour les types binaires utilisez binary. La gestion des binaires de taille important sera faite automatiquement avec les **TOAST** http://docs.postgresqlfr.org/9.0/storage-toast.html. The Oversized-Attribute Storage Technique. Les binaires de taille importante (>8Ko s on simplifie) seront stockés dans un espace physique secondaire de la table. Ceci permet à un champ sur une ligne de table, d'atteindre une taille de 1Go.

☑ Attention avec les booléens, si vous ne mettez pas `NON NULL` ceux-ci pourront prendre 3 valeurs, `true`, `false` ou `Null`. N'utilisez **JAMAIS** une chaîne de caractère pour stocker un booléen! (réfléchissez à la taille en bit d'un caractère UTF8)

☑ Remarquez le type **`ENUM`**

☑ Remarquez qu'il est possible de créer ses propre `types`, d'y associer des contraintes (`DOMAINES`), de créer des types `composés` (comme les structures en C), de gérer des `arrays`.

13.4.2. Héritage de tables

La table personnel sert de table modèle à la table `employes` et la table `interimaires`. Il y a de l'**héritage de tables**. Cela veut dire que les tables `employes` et `interimaires` héritent des toutes les colonnes de `employes`.

Testez ces commandes SQL:

```
SELECT * from drh.personnel;
SELECT * from ONLY drh.personnel ;
```

On voit que deux personnes ont été créées qui ne sont ni des employés ni des intérimaires (peut-être des stagiaires).

- ☑ L'héritage de table est utile pour modéliser des relations d'héritage ou pour optimiser le stockage de grosses tables de données, le **partitionnement** (en utilisant des tablespaces différents pour les tables héritées).
- ☑ Le principal défaut de l'héritage de table est **la gestion des index**, les index sont propres à chaque table, la définition de contraintes **d'unicité** ou de **clef étrangère** est complexifiée.

13.4.3. Clefs étrangères

Dans cette base schéma on trouve des relations de **clef étrangère** entre tables:

- classiquement entre employés et services ou entre intérimaires et agences
- mais aussi en passant par une table d'association n-aire comme pour la relation entre employés et projets

Remarquez que ces contraintes ne sont pas toujours exprimées de la même façon:

```
ALTER TABLE ONLY employes ADD CONSTRAINT "EMPLOYE_POUR_UN_SERVICE_FK"
FOREIGN KEY (ser_id) REFERENCES services(ser_id) ON UPDATE CASCADE ON
DELETE SET DEFAULT DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE ONLY services ADD CONSTRAINT "SERVICE_RELATION_PARENT"
FOREIGN KEY (ser_parent) REFERENCES services(ser_id) ON UPDATE CASCADE
ON DELETE SET DEFAULT DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE ONLY employes_projet
    ADD CONSTRAINT "employes_projet_EMPLOYES_FK" FOREIGN KEY (emp_id)
REFERENCES employes(per_id) ON UPDATE CASCADE ON DELETE CASCADE;
```

Les valeurs par défaut pour les cascades sont `RESTRICT` qui empêchent les modifications si celles-ci pourraient avoir des impacts. Remarquez qu'il existe d'autres possibilités.

13.4.4. Triggers

Sur le schéma drh des **triggers** existent, nous verrons plus en détail l'écriture et le fonctionnement des triggers, mais sachez que ce schéma se sert des ces triggers pour mettre à jour la date de modification de certaines tables, ou pour calculer le code d'un employé.

13.4.5. Contraintes

Plus important des **contraintes supplémentaires** ont été ajoutées. Examinez par exemple toutes les contraintes exprimées sur la table employes. Ces contraintes que l'on nomme des `CHECKS` sont importants pour **l'administrateur de données**. Elles permettent de s'assurer qu'une base de donnée alimentée par plusieurs programmes possède ses propres règles de validation des données et que ces règles seront toujours respectées, quelque soit le niveau de qualité des programmes utilisant la base. Ceci fais partie de la définition de la **cohérence des données** qui est un élément très important des bases de données. Les contraintes d'unicité ne sont pas les seules à pouvoir être exprimées au niveau de la base.

13.4.6. Vues

Les vues peuvent servir à de nombreuses choses:

- assurer la **compatibilité** des anciens programmes connaissant un ancien modèle de tables en leur proposant des « fausses » tables qui portent les anciens noms de colonnes
- Ajouter des **vues simplifiées sur des objets répartis** dans différentes tables (c'est le cas dans le modèle drh), il y a de nombreux exemples de telles vues dans le **pg_catalog** aussi.
- Appliquer des politiques de droits en autorisant l'accès à une vue tout en interdisant l'accès aux tables qui composent cette vue

Une vue est en fait une **requête** déjà écrite et qui se présente comme une table pour tous les utilisateurs. Si vous regardez en détail dans pgAdmin comment un vue est construite vous verrez qu'elle utilise le système des **règles** (`RULES`). Le système des règles est un système qui permet de détourner toute sortes de requêtes sut un objet pour les réécrire. Ici toutes les requêtes en lecture touchant la vue seront réécrites pour intégrer les éléments de la vraie requête sous-jacente.

➢ *Pour bien voir ce système tapez:*

```
SELECT *  FROM drh.vue_tableau_personnel;
```

Exécutez la requête. Ensuite cherchez le bouton EXPLAIN situé à droite des boutons d'exécution de requêtes. Cliquez dessus et observez la vraie requête que votre requête toute simple demande.

13.4.7. Jouons avec les triggers et les cascades

➢ *Q1: Modifiez l'identifiant du service Finances actuel de 2 à 200. Que s'est-il passé sur la table des employés? Que s'est-il passé sur la table services? Pourquoi?*

➢ *Q2 Modifiez maintenant le code du service Finance qui a l'identifiant 200 en « FXXX1 ». Que s'est-il passé sur la table des employés (faites des rafraichissements si vous affichiez déjà la table)?*

➢ *Q3: Supprimez le service Finance avec identifiant 200 et rafraîchissez la table. Que s'est-il passé sur la table service et dans la table « employes ».*

➢ *Pour rétablir la situation indiquez l'identifiant du service Finance nouvellement créé aux utilisateurs qui étaient dans ce service sur la table « employes ». Puis indiquez cet identifiant aux services dépendants du service Finance (Comptabilité et Trading).*

Solutions:

Q1 : La modification de l'identifiant du service de 2 à 200 a été répercutée dans la table des employés, ceux qui avaient un `ser_id` à `2` on maintenant un `ser_id` à `200`. Il y a aussi une clef étrangère définissant la relation de parenté des services entre eux, quand le code du service Financier à changé la `CASCADE` a répercuté le changement sur ses fils (Comptabilité et Trading)

Q2 : Lorsque le code est modifié un trigger est exécuté (`ser_update_alter_emp_code`), dans ce trigger on voit que si le code a changé une requête update est effectuée sur la table employés pour recalculer le code employé de tous les employés de ce service.

Q3 : La clef étrangère définie le comportement en cas de DELETE à SET DEFAULT, la valeur par défaut de la colonne `ser_id` pour les employés est `1`, ils sont donc passés au service d'identifiant `1`. Un trigger est déclenché sur la table employes en cas de création d'employé ou en cas de modification d'identifiant employé ou de code employé. Nous sommes dans cette troisième situation, le trigger se lance donc et recalcule le code employé. Il y a aussi une clef étrangère définissant la relation parent sur la table services, lorsque le parent « Financier » a disparu les deux services fils sont passés à la valeur par défaut (service inconnu).

---

14. Requêtes

Nous allons faire des requêtes sur le schéma drh et essayer de faire un tour d'horizon de l'étendue des possibilités offertes par la commande `SELECT`.

14.1. Sélection de colonnes, SELECT *, Distinct

L'étoile dans le `SELECT` est un caractère spécial qui signifie « toutes les colonnes de toutes les tables de mon `SELECT` ».

➢ *Testez les résultats obtenus avec ces différentes requêtes:*

```
SELECT * FROM drh.EMPLOYES;
SELECT per_nom FROM drh.employes;
SELECT distinct(per_nom) FROM drh.employes;
```

14.2. ORDER BY

Pour trier les résultats d'un requêtes on doit utiliser `ORDER BY`. Testez ces différentes requêtes

```
SELECT per_nom,per_prenom,emp_code_pays FROM drh.employes
  ORDER BY 1;
SELECT per_nom,per_prenom,emp_code_pays FROM drh.employes
  ORDER BY 1,3,2;
SELECT per_nom,per_prenom,emp_code_pays FROM drh.employes
  ORDER BY emp_code_pays, per_nom,per_prenom;
SELECT per_nom,per_prenom,emp_code_pays FROM drh.employes
  ORDER BY emp_code_pays DESC, per_nom DESC,per_prenom ASC;
SELECT per_nom,per_prenom,emp_code_pays FROM drh.employes
  ORDER BY emp_code_pays DESC NULLS LAST, per_nom DESC,per_prenom ASC;
```

14.3. Le problème du NULL

NULL est une valeur à laquelle vous devrez **TOUJOURS** penser. Partout, dans les tris, les fonctions, les regroupements, nous allons le recroiser partout et il aura parfois des effets dévastateurs. Prenez dès à présent l'habitude de penser aux valeurs nulles, un peu comme le « si x!=0 » en mathématique quand vous passiez votre Bac.

NULL n'est pas la chaîne vide, il n'est pas zéro, il est l'absence d'information, le vide. Et si par exemple vous appliquez un opérateur de concaténation de chaîne ou d'addition vous allez avoir des surprises:

```
SELECT 42 + NULL;
SELECT 'toto' || NULL;
```

14.4. Fonctions et opérateurs utiles

Il existe énormément de fonctions qui peuvent vous être utiles dans les requêtes. Vous pouvez en voir la liste dans le pg_catalog `Catalogue > PostgreSQL > Fonctions` en liste plus de deux mille. La documentation en ligne est plus utile, les fonctions y sont classées: http://docs.postgresqlfr.org/9.1/functions.html

Nous allons regarder un tout petit sous-ensemble des ces fonctions utiles.

14.4.1. Travailler sur les chaînes de caractères

Nous avons vu l'opérateur de concaténation `||`. Il existe aussi à partir de postgresql 9.1 des fonctions `concat` et `concat_ws`. Ces fonction concatènent plusieurs chaînes, **en ignorant les NULL** et en ajoutant un séparateur pour la deuxième.

Les autres fonctions les plus utiles sont:

- `character_length(chaîne)`: longueur de la chaîne
- `lower(chaîne)` : passage en minuscules (`upper()` pour l'inverse)
- `substring(chaine,début,fin)`: extraction d'une sous chaîne
- `trim(chaîne)`: suppression des espaces inutiles, mais elle peut faire beaucoup plus (regardez la documentation en ligne)

Pour rechercher un sous chaîne dans un chaîne il existe l'opérateur **`LIKE`** ou **`ILIKE`** (insensible à la casse), qui prend en argument une chaîne de caractère où **`?`** est remplacé par un caractère et **`%`** par 0 ou n caractères.

Les chaînes de caractères sont séparées par des guillemets simples `'` (guillemets doubles pour MySQL). Pour mettre une apostrophe ou un guillemet simple on doublera le guillemet `''`, non pas `"` où on utilisera un caractère d'échappement.

Les chaînes qui possèdent des caractères d'échappement (comme `\n` pour le retour à la ligne) devraient être préfixées par le caractère `E`: `E'Ceci est \n un chaîne sur deux lignes.'`

```
select 'test d''échappement';
select E'test d\'échappement';
```

Les noms de tables ou d'alias de colonnes sont eux séparés par des guillemets doubles `"` (quote anglaise sous MySQL `` ` ``)

14.4.2. Travailler avec les nombres

Il y a de très nombreux opérateurs mathématiques et beaucoup de fonctions aussi, notons:

- `round()` : arrondi mathématique (52.65 donne 53)
- `ceil`(): arrondi à l'entier inférieur (52.65 donne 52), voir `floor`() pour le contraire.
- `random():` un nombre réel aléatoire entre 0 et 1 (non compris)

14.4.3. Somme, Moyenne, Minimum, maximum

De nombreuses fonctions peuvent travailler simplement sur un table sans nécessiter d'opérations d'agrégation (`GROUP BY`, que nous verrons plus loin). Retenez donc que ces fonctions peuvent fonctionner sans `GROUP BY`:

- `sum()`: somme
- `max()`: maximum
- `min()`: minimum
- `avg()`: moyenne
- `count()`: compteur, `count(*)` compte les lignes de résultat. `count(champ)` compte les occurrences non nulles du champ, `count(distinct champ)` compte les valeurs distinctes du champ.

14.4.4. Travailler avec les dates

Je ne saurais trop vous conseiller la lecture complète de http://docs.postgresqlfr.org/9.1/functions-datetime.html où vous trouverez de nombreuses fonctions très utiles comme `age()`, ainsi que des opérateurs miraculeux comme `OVERLAP`.

Signalons ici quelques fonctions de base pour le travail avec les dates, et plus qu'un long discours utilisons de bons exemples:

```
SELECT current_timestamp, now(),current_date,current_time;

SELECT date_part('hour',current_timestamp),
  date_trunc('hour',current_timestamp),
  extract(hour from current_timestamp);

SELECT current_timestamp + INTERVAl '2 months 3 days 3 hours';

SELECT current_timestamp - '2001-10-19',
  justify_interval(current_timestamp - '2001-10-19');
```

14.4.5. Autres fonctions utiles

Il y a un grand nombres de fonctions utiles. Nous avons déjà vu `generate_series()`, en voici quelques autres:

- `coalesce(a;b;c;d)` : renvoie la première valeur non Nulle de la liste.
- `string_agg()` : l'équivalent de `group_concat` sous MySQL

14.4.6. Exercices

- *R1: Quel est le salaire annuel minimum des employés?*
- *R2: Quel est l'âge du plus jeune employé?*
- *R3: Quelle est la date d'embauche de l'employé le plus récemment embauché?*
- *R4: Quel est le salaire moyen des employés?*
- *R4bis: Arrondissez le résultat à 2 chiffres après la virgule*
- *R5: combien d'employés?*
- *R6: combien d'employés avec un code pays?*
- *R7: combien de code pays différents?*
- *R8:Combien d'employés ont la chaîne 'im' dans leur prénom (majuscules ou minuscules)?*

14.4.7. Solutions

R1:

```
SELECT MIN(emp_salaire_annuel) as "salaire minimum"
FROM drh.employes;
```

R2:

```
SELECT MIN(age(emp_naissance))
FROM drh.employes;
```

R3:

```
SELECT MAX(emp_date_entree)
FROM drh.employes;
```

R4:

```
SELECT AVG(emp_salaire_annuel) as "salaire moyen"
FROM drh.employes;
```

R4 bis:

```
SELECT ROUND(AVG(emp_salaire_annuel),2) as "salaire moyen"
FROM drh.employes;
```

R5

```
SELECT count(*)
FROM drh.employes;
```

R6

```
SELECT count(emp_code_pays)
FROM drh.employes;
```

Pas besoin de spécifier un filtre « `IS NOT NULL` », le `count` ne compte pas les valeurs nulles.

R7

```
SELECT count(distinct emp_code_pays)
FROM drh.employes;
```

R8

```
SELECT COUNT(per_prenom)
FROM drh.employes
WHERE per_prenom ILIKE '%im%';
```

14.5. Filtrage avec WHERE

La dernière question introduisait le filtrage des requêtes avec L'opérateur `WHERE`. La seule difficulté avec l'opérateur `WHERE` est qu'il n'accepte qu'une seule condition, si celle-ci doit contenir de nombreux filtres ils doivent être combinés avec des `AND` et des `OR`, au besoin en ajoutant des parenthèses pour bien spécifier les groupes et les priorités.

> ☑ Le NULL se gère de façon spéciale dans les filtres `WHERE foo<>null` ne fonctionne pas et doit s'écrire `WHERE foo IS NULL` ou `WHERE foo IS NOT NULL`. On voit donc souvent des expressions du type `(WHERE foo < 12 OR foo IS NULL)`.

➢ *Que fais cette requête?*

```
SELECT per_nom,per_prenom,emp_code_pays, extract(year from
age(emp_naissance)) as age
FROM drh.employes
WHERE
(emp_code_pays IN ('US','CA')
  AND age(emp_naissance) >= '50 year'::interval )
OR (emp_code_pays IN ('FR','UK')
```

```
  AND age(emp_naissance) >= '40 year'::interval );
```

Réponse:

elle liste les noms,prénoms et ages des employés qui sont aux états-unis ou au canada et qui ont plus de 50 ans, ou bien qui sont en France ou au Royaume Unis et qui ont plus de 40 ans. Remarquez qu'on utilise pas `extract` dans les clauses `WHERE`, ce traitement n'est utile que pour l'affichage final.

14.6. LIMIT et OFFSET

- http://docs.postgresqlfr.org/9.0/queries-limit.html

Le dernier mot clef d'une requête SQL peut être LIMIT, Cette instruction permetde limiter le nombre de résultats renvoyés par la requête. Si votre requête ne nécessite pas trop d'instruction complexes (comme un tri non prévu par un index) le fait de limiter le nombre de résultat va très fortement réduire le temps d'exécution de la requête. Cela va aussi permettre de limiter le transfert d'informations vers l'application au nombre de lignes réellement utile (ne pas par exemple rapatrier les 45 256 dernières news pour afficher uniquement les 10 dernières!).

Les mots clefs LIMIT et OFFSET vont en fait être la base de la pagination d'une requête. Ainsi pour afficher les 25 premiers résultats on peut écrire:

```
SELECT *
FROM matable
LIMIT 25
```

ou encore:

```
SELECT *
FROM matable
LIMIT 25 OFFSET 0
```

Et pour afficher les résultats de la troisième page :

```
SELECT *
FROM matable
LIMIT 25 OFFSET 50
```

`OFFSET 50` signifie que la première ligne de résultat sera la **51**ème.

14.7. Sous Requêtes

Une sous-requête est une requête à l'intérieur de notre requête principale.

14.7.1. ANY, ALL et EXISTS

Nous avons vu dans la requête précédente l'opérateur `IN`, l'opérateur inverse `NOT IN` existe aussi. Il existe d'autres opérateurs du même type:

- `ANY` : renverra vrai si au moins un élément est concordant (`IN` est équivalent à `=ANY`)
- `ALL` : ne renverra vrai que si tous les éléments concordent (`NOT IN` est équivalent à `<>ALL`)
- `EXISTS` : est un équivalent de `ANY`, avec ces deux opérateurs il n'y a pas de garantie que la sous-requête sera complètement exécutée, dès qu'une ligne est bonne la sous-requête peut s'arrêter.

☑ **N'oubliez jamais qu'`EXISTS` existe.** Si vous voulez vérifier qu'un article appartient à une catégorie avez vous réellement besoin de rapatrier l'ensemble des catégories liées à l'article pour les compter et vérifier que ce comptage est supérieur à zéro? `EXISTS` est une optimisation de requête simple et peu coûteuse qu'un développeur SQL ne devrait jamais oublier.

- *Exécutez ces requêtes et essayez de comprendre le résultat*

```
SELECT 42 IN (42,43,44, NULL);
```

```
SELECT 'US' = ANY (SELECT emp_code_pays FROM drh.employes);
```

```
SELECT 'ZZ' = ANY (SELECT emp_code_pays FROM drh.employes);
```

```
SELECT 'ZZ' = ANY (
SELECT emp_code_pays FROM drh.employes WHERE emp_code_pays IS NOT NULL
);
```

```
SELECT 'US' = ANY (
SELECT emp_code_pays FROM drh.employes WHERE emp_code_pays IS NULL);
```

- *Q1: La troisième et la dernière requête renvoient NULL et non FAUX. Pourquoi?*
- *Q2: Que fais cette requête?*

```
SELECT per_nom,per_prenom, emp_code_pays,emp_salaire_annuel
FROM drh.employes
WHERE emp_salaire_annuel >= ALL
( SELECT emp_salaire_annuel
  FROM drh.employes
  WHERE emp_code_pays IN ('US','UK')
);
```

- *Q3: Que se passerait-il si on ajoute la fonction `max(emp_salaire_annuel)` dans la sous-requête*
- *Q4: Que fais cette version?*

```
SELECT per_nom,per_prenom, emp_code_pays,emp_salaire_annuel
FROM drh.employes
WHERE emp_salaire_annuel >= ALL
( SELECT max(emp_salaire_annuel)
  FROM drh.employes
  WHERE emp_code_pays NOT IN ('FR') OR emp_code_pays IS NULL
);
```

- *Q5: Pourquoi cette requête renvoie NULL?*

```
SELECT per_nom,per_prenom, emp_code_pays,emp_salaire_annuel
FROM drh.employes
```

```
WHERE emp_salaire_annuel >= ALL
( SELECT emp_salaire_annuel
  FROM drh.employes
  WHERE emp_code_pays NOT IN ('FR') OR emp_code_pays IS NULL
);
```

- *Q6: Que se passerait-il si nous changions `ALL` en `ANY` comme dans le requête si dessous?*

```
SELECT per_nom,per_prenom, emp_code_pays,emp_salaire_annuel
FROM drh.employes
WHERE emp_salaire_annuel >= ANY
( SELECT emp_salaire_annuel
  FROM drh.employes
  WHERE emp_code_pays NOT IN ('FR') OR emp_code_pays IS NULL
  AND emp_salaire_annuel IS NOT NULL
);
```

***Réponses**:*

Q1: `ANY` renvoi NULL si il rencontre un NULL dans la liste des valeurs comparées, il ne renvoi plus faux. La requête correcte est donc la quatrième.

Q2: Cette requête renvoie les noms, prénoms, codes pays et salaires des employés ayant un salaire annuel supérieur ou égal au salaire maximum des employés des USA ou du Royaume Unis. Par chance aucun salaire dans ces pays n'est à NULL. Sans cela la requête serait en échec.

Q3: le résultat serait le même. Le risque de rencontrer un NULL serait diminué (mais pas absent, si tous les salaires sont vides). Par contre la sous requête ne renvoie qu'une seule ligne (et une seule colonne) , on peut alors écrire la requête sans le `ALL`:

```
SELECT per_nom,per_prenom, emp_code_pays,emp_salaire_annuel
FROM drh.employes
WHERE emp_salaire_annuel >=
( SELECT max(emp_salaire_annuel)
  FROM drh.employes
  WHERE emp_code_pays NOT IN ('FR') OR emp_code_pays IS NULL
);
```

Q4: Cette version renvoie la liste des personnes qui ont un salaire supérieur ou égal au salaire maximum des gens qui ne sont pas en France.

Q5: nous avons enlevé le `max`, et une des lignes au moins de la sous requête à un salaire NULL, `ALL` renvoie NULL s'il trouve un NULL. La vraie solution serait d'ajouter un `AND emp_salaire_annuel IS NOT NULL` dans la sous-requête mais aussi dans toutes les précédentes.

Q6: si nous changeons pour `ANY` nous obtenons beaucoup plus de résultats, puisqu'il suffit alors d'avoir un salaire supérieur à n'importe quel salaire d'un employé ne travaillant pas en France.

14.7.2. Emplacement d'une sous-requête

Une sous-requête peut se situer dans la liste des champs sélectionnés:

```
SELECT per_nom,
  per_prenom,
  emp_code_pays,
  emp_salaire_annuel,
 ( SELECT COALESCE(max(emp_salaire_annuel),0.0)
    FROM drh.employes
    WHERE emp_code_pays='FR'
  )  as "salaire max FR"
FROM drh.employes;
```

Ou dans la clause `FROM` (elle devient alors une des tables sources, ce qui est assez dangereux, on en reparlera):

```
SELECT per_nom,
 per_prenom,
 emp_code_pays,
 emp_salaire_annuel,
 ssrq1."maxi fr"
FROM drh.employes,( SELECT COALESCE(max(emp_salaire_annuel),0.0) as
"maxi fr"
  FROM drh.employes e2
  WHERE e2.emp_code_pays='FR'
) ssrq1;
```

Elle peut enfin, et c'est là qu'on les trouve le plus souvent, se situer dans les clauses de filtrage comme dans les exercices précédents.

Le plus souvent ces requêtes ne seront exécutés qu'une seule fois par le moteur SQL, ce fait combiné à une clause `EXISTS` peut rendre les sous-requêtes très utiles.

> ☑ **Attention**, les sous-requêtes sont malheureusement très souvent utilisée à la place des jointures, notamment en tant que sous-requête corrélées, elles deviennent alors beaucoup plus coûteuses et sont très souvent la marque d'une méconnaissance du langage SQL. Si vous débutez essayez de penser d'abord aux jointures, puis aux sous-requêtes en dernier recours, et non l'inverse.

14.7.3. Sous requêtes Corrélées

Une sous-requête corrélée est une sous-requête qui comporte une référence à une des tables de sa requête parente. Schématiquement

```
Select a.*
FROM tableA a
WHERE a.foo = (
  SELECT foo
  FROM tableB b
  WHERE b.bar =a.bar  — ici la corrélation avec la requête parente
);
```

A partir du moment où une requête est corrélée le résultat de son exécution dépend d'une valeur de la ligne de la table parente qui est en train d'être évaluée.

On passe d'un mode où le résultat de la sous-requête était calculé une fois et une seule puis comparé à chaque ligne de la requête parente à un mode ou à chaque ligne parente on va devoir recalculer la sous-requête. Cela peut avoir un impact non négligeable en terme de performances de la requête. Vous devrez apprendre à reconnaître ces requêtes, la

marque principale est donc qu'il y est fais référence à un table qui ne figure pas dans la clause `FROM` de la sous-requête.

Ces sous-requêtes ont leur utilité, mais il s'agit très souvent de modes de requêtes avancés (comme les requêtes récursives).

☑ On retrouvera cette problématique du **calcul pour chaque ligne ou du calcul effectué une fois** pour toute la requête dans de nombreuses optimisations du SQL. Ainsi observez les différences entre ces deux requêtes:

```
SELECT * FROM drh.employes WHERE extract(year from age(current_date,emp_naissance))>=40;
```

```
SELECT * FROM drh.employes WHERE emp_naissance < current_date-INTERVAL '40 years';
```

☑ Retenez une règle, **dans les filtrages de requêtes les fonctions sont mieux à droite**

14.8. Les Jointures

14.8.1. Produit Cartésien

Un produit cartésien est l'ensemble des combinaisons entre deux tables. Vous pouvez en obtenir un très simplement en ajoutant plusieurs tables dans la clause `from`.

```
SELECT *
FROM drh.employes, drh.services
```

Le produit cartésien de deux tables est souvent inintéressant. Nous allons filtrer le croisement de deux tables (où nous avons pour chaque ligne de la première l'ensemble des lignes de la seconde), pour ne retenir que quelques lignes de la seconde – souvent une seule-- pour chaque ligne de la première. C'est ce qu'on appelle une jointure.

Une des façons de l'exprimer est d'ajouter des clauses de filtrage mettant en relation les deux tables:

```
SELECT *
FROM drh.employes, drh.services
WHERE employes.ser_id=services.ser_id
```

Si nous utilisons des alias de nom pour les tables:

```
SELECT *
FROM drh.employes em, drh.services se
WHERE em.ser_id=se.ser_id
```

On imagine alors que la requête représente le produit cartésien des deux tables, sur lequel un filtrage est appliqué. Heureusement l'analyseur de requêtes est sans doute beaucoup plus fin que cela et aura repéré que nous voulions une jointure à partir d'une colonne indexée, une clef étrangère. Vous pouvez regarder le EXPLAIN de la requête pour vous en assurer.

Il faut retenir cependant qu'exprimer les jointures de cette façon n'est pas une très bonne idée, surtout si la requête commence à travailler avec un nombre important de tables.

- on prends le risque d'oublier les conditions de jointures (une liste de tables et seulement 9 conditions de jointure?)

- On écrit une requête qui demande un filtrage de produit cartésien tout en espérant que l'analyseur de requête soit plus intelligent que nous
- Si on veut ajouter des vrais filtrages du résultat, ils seront mélangés avec nos conditions de jointure

Bref c'est assez peu maintenable.

14.8.2. Jointure Complète, Droite, Gauche, Naturelle

Les jointures s'expriment normalement avec des vraies conditions de jointure, qui peuvent être assez complexes. Voici un exemple théorique complet:

```
SELECT f.id,f.name,a.i, b.name, b2.name, c.foo, c.bar.d.amount, g.*
FROM tableA a,
  NATURAL JOIN tableC c,
  INNER JOIN tableB b ON a.bid=b.id,
    LEFT JOIN tableB b2 ON b.parent=b2.id,
    LEFT JOIN tableD d ON b.did=d.id,
      LEFT JOIN tableE e ON (d.eid=e.id AND e.active),
  RIGHT JOIN tableF f ON a.fid = f.id,
    LEFT JOIN tableG g ON f.code>g.code
WHERE f.name LIKE 'toto%'
AND f.id NOT IN (12,42,34)
```

C'est un type de requête qui est assez réaliste. Une librairie d'un gestionnaire de persistance comme Hibernate pourrait très bien générer ce type de requête, on peut imaginer un écran qui permet à l'utilisateur de filtrer sur un nom de la tableF, tout en excluant certaines lignes. Les jointures sont construites d'après des relations connues entre les objets.

Depuis PostgreSQL9 le moteur d'analyse est assez intelligent pour repérer où la ligne de jointure avec la tableE ne sers à rien (aucun champ utilisé dans le filtrage de la requête, pas de champ affiché), il ne le fera donc pas.

La jointure sur cette tableE était pourtant intéressante car on y voit qu'un filtrage basé sur plusieurs champs est autorisé dès la jointure (ici on ajoute le fait que le booléen active est à true sur la table tableE.

On pourra remarquer aussi dans la jointure de la table G qu'une jointure ne se fait pas forcément sur une égalité, on risque sur cette requête d'obtenir beaucoup de lignes de la table G pour chaque ligne de la table F.

Le premier `JOIN`, avec la table C est un **`NATURAL JOIN`**, ils sont assez dangereux à utiliser et dur à maintenir, ils signifient « fais la jointure entre la table A et la table C en te servant des champs qui ont le même nom dans les deux tables ». Si vous avez utilisé une nomenclature de champs comme celle utilisé dans les bases d'exemple cela peut fonctionner, mais on voit que la requête elle-même n'exprime pas clairement la relation de jointure. C'est dangereux si l'on reprend un des principes de programmation qui est « Makes Wrong Code Looks Wrong ». Le code de la requête ici est dur à comprendre sans disposer du schéma des tables, ce qui n'est pas le cas pour les autres jointures.

La seconde jointure est un **`INNER JOIN`**, il signifie qu'il ne faut garder que les lignes qui ont un enregistrement en commun entre la table A et la table B. Si des lignes de B n'ont pas de correspondant dans A elles disparaissent, si des lignes de A n'ont pas de lien vers B elles disparaissent aussi

Viennent ensuite des `LEFT JOIN` (très courants) et des `RIGHT JOIN` (beaucoup moins). Nous allons les expliquer ci-dessous.

> ☑ Si vous voulez un produit cartésien vous pouvez l'exprimer comme une jointure avec `CROSS JOIN`.

Les noms des jointures sont parfois exprimés différemment:

- `JOIN`: `INNER JOIN` : **Jointure interne** (on aura les champs des deux tables qui concordent, ceux qui n'ont pas de concordance, quelque soit le côté ne seront plus là)
- **Auto-jointure**: une jointure de la table vers elle-même (le cas des relations parent souvent)
- `LEFT JOIN`: `LEFT OUTER JOIN`: **jointure externe gauche** (les champs de la nouvelle table, donc celle de gauche, seront affichés avec des valeurs NULL si un champ de la table d'origine n'a pas de concordance dans cette table)
- `RIGHT JOIN`: `RIGHT OUTER JOIN`: **jointure externe droite** (toutes les valeurs de la nouvelle table seront gardées, les champs de la table d'origine qui n'ont pas de concordance avec cette nouvelle table seront affichés avec des valeurs NULL)
- `FULL JOIN`: `FULL OUTER JOIN`: **jointure externe bilatérale**, il s'agit d'un `LEFT JOIN` plus un `RIGHT JOIN` (contrairement au `INNER JOIN` où on a aucun NULL on a donc aussi les champs NULL à droite et à gauche)

14.8.3. Quelques exercices

- *Q1: Affichez tous les employés avec éventuellement le nom de leur service*
- *Q2: Affichez tous les employés avec éventuellement la liste des projets auxquels ils ont participé*
- *Q3: Affichez tous les employés ayant participé au projet « Séminaire 2010 »*
- *Q4: Affichez tous les employés ayant participé à un séminaire, donnez aussi le nom du séminaire, quitte à avoir plusieurs lignes par employé, triez par nom de projet*
- *Q5: que se passe-t-il si on utilise que des `LEFT JOIN` pour la requête précédente?*

14.8.4. Solutions

Q1:

```
SELECT per_nom,per_prenom,emp_code, ser_nom
FROM drh.employes em
LEFT JOIN drh.services se ON em.ser_id = se.ser_id
ORDER BY ser_nom ASC;
```

Q2:

```
SELECT per_nom,per_prenom,emp_code,pro_nom
FROM drh.employes em
LEFT JOIN drh.employes_projet empr ON em.per_id = empr.emp_id
LEFT JOIN drh.projet pr ON empr.pro_id = pr.pro_id
ORDER BY 1,2,4;
```

Q3:

```
SELECT pro_nom,per_nom,per_prenom,emp_code
FROM drh.projet pr
INNER JOIN drh.employes_projet empr ON empr.pro_id = pr.pro_id
INNER JOIN drh.employes em ON em.per_id = empr.emp_id
WHERE pr.pro_nom='Séminaire 2010';
```

Q4:

```
SELECT pro_nom,per_nom,per_prenom,emp_code
FROM drh.projet pr
INNER JOIN drh.employes_projet empr ON empr.pro_id = pr.pro_id
INNER JOIN drh.employes em ON em.per_id = empr.emp_id
WHERE pr.pro_nom LIKE 'Séminaire %'
ORDER BY 1,2,4;
```

Q5:

```
SELECT pro_nom,per_nom,per_prenom,emp_code
FROM drh.projet pr
LEFT JOIN drh.employes_projet empr ON empr.pro_id = pr.pro_id
LEFT JOIN drh.employes em ON em.per_id = empr.emp_id
WHERE pr.pro_nom LIKE 'Séminaire %'
ORDER BY 1,2,4;
```

Sous cette forme le projet Séminaire 2011 apparaît dans la liste, avec des NULL pour les champs employes

14.9. Requêtes avancées

14.9.1. GROUP BY

Notre requête se compose pour l'instant d'une ligne de sélection des champs à afficher, d'une liste de tables sources, avec éventuellement des jointures, de conditions de filtrages, qui pourraient contenir des sous-requêtes. En bas de la requête nous connaissons les lignes d'ordre et les conditions de limitation du nombre de lignes.

Après les conditions de filtrage et avant les conditions d'ordre et de limite un nouvel élément peut s'insérer, le `GROUP BY`, les conditions **d'agrégation**.

L'idée de l'agrégation est en fait assez simple. La requête n'est pas encore ordonnée, elle n'est pas encore non plus limitée en nombre de lignes de résultats. Les conditions d'agrégation vont regrouper les lignes sur des critères communs, dans les parties qui ne seront pas gardées (car sur des critères non listés) des opérations d'agrégation vont pouvoir être effectuées (des sommes, moyennes, min, max).

Cela permet d'obtenir des `min()`, `max()`, `avg()`, `sum()`, `count()` qui renverront plusieurs lignes de résultats. Voyons un exemple:

Nous avions un comptage des employés

```
SELECT count(distinct(per_nom))
FROM drh.employes;
```

Nous allons maintenant pouvoir faire ce comptage par pays:

```
SELECT coalesce(emp_code_pays,'inconnu'), count(distinct(per_nom))
FROM drh.employes
GROUP BY emp_code_pays;
```

> ☑ La règle la plus important est que la ligne d'agrégation devrait contenir **AU MOINS** tous les champs affichés dans la ligne `SELECT` qui ne sont pas des opérations d'agrégation

Prenons là encore un exemple, je veux afficher à la fois le nombre d'employés et le salaire moyen, pour chaque pays, et dans chacun de ces pays les services concernés, cette requête sera invalidée:

```
SELECT
  coalesce(emp_code_pays,'inconnu'),
  substring(upper(coalesce(ser_nom,'inc')),1,4),
```

```
  count(distinct(per_nom)) as nb,
  round(avg(emp_salaire_annuel),2) as "moyenne salaire"
FROM drh.employes em
LEFT JOIN drh.services se ON em.ser_id=se.ser_id
GROUP BY emp_code_pays;
```

Ceci fonctionnera:

```
SELECT
  coalesce(emp_code_pays,'inconnu'),
  substring(upper(coalesce(ser_nom,'inc')),1,4),
  count(distinct(per_nom)) as nb,
  round(avg(emp_salaire_annuel),2) as "moyenne salaire"
FROM drh.employes em
LEFT JOIN drh.services se ON em.ser_id=se.ser_id
GROUP BY emp_code_pays,ser_nom
ORDER BY emp_code_pays,ser_nom,nb;
```

Ma ligne d'agrégation `GROUP BY emp_code_pays,ser_nom` se doit de contenir au moins tous les champs qui ne sont pas des agrégats. Les agrégats sont le `count(nom)` et le `avg(salaire)`. Remarquez que le `GROUP BY` n'est pas obligé d'appliquer toutes les transformations qui sont effectuées en sortie, s'il les utilisaient cela aurait un sens au moment du regroupement (deux services qui auraient les 4 mêmes premières lettres seraient regroupés).

Nous avions parlé de la fonction `string_agg`, c'est ici avec les agrégations qu'elle prend tout son sens, testez cette requête:

```
SELECT emp_code_pays, string_agg(distinct(per_nom),', ')
FROM drh.employes
GROUP BY emp_code_pays;
```

14.9.2. HAVING

`HAVING` est un filtrage, comme le `WHERE` mais qui est effectué après les opérations d'agrégation, vous remarquerez que le SQL est assez logique et que l'ordre des commandes correspond à un vrai ordre d'exécution de la requête en interne. `HAVING` se place donc après les `GROUP BY`. Cela permet d'appliquer des filtrages sur les résultats de l'opération d'agrégation. Reprenons un exemple précédent:

```
SELECT
  coalesce(emp_code_pays,'inconnu'),
  substring(upper(coalesce(ser_nom,'inc')),1,4),
  count(distinct(per_nom)) as nb,
  round(avg(emp_salaire_annuel),2) as "moyenne salaire"
FROM drh.employes em
LEFT JOIN drh.services se ON em.ser_id=se.ser_id
GROUP BY emp_code_pays,ser_nom
HAVING avg(emp_salaire_annuel)>=28000.0
 AND count(distinct(per_nom))>=2
ORDER BY emp_code_pays,ser_nom,"moyenne salaire",nb;
```

14.9.3. UNION et autres ensembles

`UNION` permet de regrouper le résultat de deux requêtes qui comportent le même nombre de colonnes (et des types compatibles pour chaque colonne).

☑ `UNION` **supprime les doublons** du résultat, utilisez `UNION ALL` pour garder les doublons

➢ *la vue présentée dans le schéma drh est une requête `UNION` vous pouvez l'examiner.*

☑ Il existe d'autres opérateurs ensemblistes, `INTERSECT` pour l'intersection des deux résultats, `EXCEPT` pour la soustraction d'ensembles.

14.9.4. Quelques exercices

➢ *Q1: Affichez la moyenne des salaires des participants aux séminaires, listez le résultat par séminaire.*

➢ *Affichez la moyenne des salaires mensuels des intérimaires (en basant un mois à 30 jours), par agence d'intérim, ajoutez le nombre d'intérimaires recensés pour ces agences*

➢ *Affichez le nom, prénom et code employé du ou des employés ayant le salaire minimum*

Solutions:

Q1:

```
SELECT ser_nom, round(coalesce(avg(emp_salaire_annuel),0),2) as
"moyenne salaires"
FROM drh.services ser
  LEFT JOIN drh.employes emp ON ser.ser_id=emp.ser_id
GROUP BY ser_nom
ORDER BY 2 DESC;
```

Q2:

```
SELECT
  age_nom,
  round(30*coalesce(avg(int_salaire_quotidien),0),2) as "moyenne
salaire mensuel",
  count(int.int_id) as "nb intérimaires"
FROM drh.agences ag
  LEFT JOIN drh.interimaires int ON ag.age_id = int.age_id
GROUP BY age_nom
ORDER BY 2 DESC;
```

Q3:

```
SELECT per_prenom, per_nom, emp_code
FROM drh.employes emp
WHERE emp_salaire_annuel = (
  SELECT MIN(emp_salaire_annuel)
  FROM drh.employes em2
);
```

Effectivement, il y a un piège, il n'y a pas de `GROUP BY`. Remarquez que nous obtenons deux personnes.

14.9.5. Curseurs et Table temporaires

- http://docs.postgresql.fr/9.0/plpgsql-cursors.html
- http://docs.postgresql.fr/9.0/sql-createtable.html

Les tables temporaires et les curseurs sont des objets qui sont utilisées par les programmes utilisant la base. Ces objets permettent de stocker temporairement le résultat d'une requête pour travailler avec.

On utilisera les curseurs pour par exemple paginer la lecture des résultats d'une requête importante. Le curseur sera détruit à la fin de la transaction (que nous verrons un peu plus tard).

La table temporaire est par contre une table, visible uniquement depuis la session qui créé cette table, et qui disparaîtra avec la session ou la transaction. Si elle porte le même nom qu'une vraie table elle masquera cette vraie table. L'avantage d'une table temporaire est que l'on peut s'en servir pour y ajouter des index (qui seront eux aussi temporaires), ou pour faire tourner l'analyseur de table dessus (`vacuum analyze`) afin d'optimiser les futures requêtes sur cette table. On peut par exemple utiliser une table temporaire pour créer une table et la manipuler puis la destiner à un export CSV (http://www.postgresql.org/docs/9.0/static/sql-copy.html).

14.9.6. WINDOW

Les fonctions window sont des opérations avancées sur les agrégats.

La documentation en ligne fournis de bonnes explications http://docs.postgresqlfr.org/9.0/tutorial-window.html, en la matière un bon exemple vaut de très longues explications:

```
SELECT per_nom,
  per_prenom,
  emp_code,
  emp_code_pays,
  emp_salaire_annuel,
  avg(emp_salaire_annuel) OVER (PARTITION BY emp_code_pays) as
"moyenne salaire du pays"
FROM drh.employes em
ORDER BY 4;
```

ici la fonction window est sur la 6ème colonne affichée:

```
avg(emp_salaire_annuel) OVER (PARTITION BY emp_code_pays) as "moyenne
salaire du pays"
```

Va nous permettre d'afficher pour chaque ligne de la table employes la moyenne des salaires correspondant au code pays de cette ligne. N'hésitez pas à consulter la documentation en ligne, de nombreuses choses sont possibles avec les fonctions window, mais demandent des tests rigoureux.

```
SELECT per_nom,
  per_prenom,
  emp_code,
  emp_code_pays,
  emp_salaire_annuel,
  rank() OVER (
    PARTITION BY emp_code_pays ORDER BY emp_salaire_annuel DESC
  )  as "ordre dans le pays"
FROM drh.employes em
ORDER BY 4;
```

14.9.7. Requêtes récursives

Si vous avez déjà essayé de gérer dans une application le chargement de données d'un arbre (une arborescence de fichiers, une arborescence de taxonomie, de catégories), surtout en partant d'un ensemble de cet arbre, vous aurez pu mesurer une certaine complexité algorithmique.

Hors le moteur de base de donnée est capable de vous fournir les données arborescentes. En tous cas les moteurs de base de donnée puissants (PostgreSQl v>8.4, Oracle –`CONNECT BY`--, mais pas MySQL).

A titre personnel je pense qu'essayer de comprendre puis d'améliorer une requête récursive est un très bon moyen d'appréhender les possibilités offertes par le SQL dans PostgreSQL.

L'algorithme de la requête se présente ainsi

```
WITH RECURSIVE t(n) AS (
   -- first term
   SELECT 1
 UNION ALL
   -- recursive term
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

- *Nous avons une table récursive sous la main, la table des services, avec une relation parent en auto-jointure. Essayons de lister les services sous forme arborescente. Je donne ici une requête qui fonctionne, l'exercice consiste à la comprendre.*

```
WITH RECURSIVE rectable(
  ser_id,
  ser_nom,
  ser_code,
  ser_parent,
  rlevel,
  rpath,
  rflatpath,
  rcycle
) AS (
    SELECT
     ser_id,
     ser_nom,
     ser_code,
     ser_parent,
     1 as rlevel,
     ARRAY[ser_id],
     ser_nom::character varying(255) as rflatpath,
     false
    FROM drh.services
    WHERE ser_parent=1
    AND ser_id<>1
  UNION ALL
    SELECT
     orig.ser_id,
     orig.ser_nom,
     orig.ser_code,
     orig.ser_parent,
     rec.rlevel+1 as rlevel,
```

```
     rec.rpath||orig.ser_id,
     (rec.rflatpath||'/' ||orig.ser_nom)::character varying(255) as
rflapath,
     orig.ser_id=ANY(rec.rpath)
    FROM drh.services orig,rectable rec
    WHERE orig.ser_parent=rec.ser_id
)
SELECT * FROM rectable
ORDER BY rectable.rflatpath
LIMIT 100;
```

- *Que se passe-t-il si on enlève le `AND ser_id<>1`, que se passe-t-il si on inverse `WHERE orig.ser_parent=rec.ser_id` en `WHERE orig.ser_id=rec.ser_parent`?*

> ☑ n'enlevez jamais le `LIMIT 100` tant que vous êtes en test sur une requête récursive.

- *Essayez de faire partir la requête depuis un sous arbre*
- *Essayez de créer un cycle, que peut on faire pour arrêter la requête en cas de détection de cycle?*

> ☑ Regardez le EXPLAIN de la requête. Un requête récursive peut être coûteuse à l'exécution. Si votre arborescence est simple il est peut-être possible de stocker une version 'à plat' de l'arborescence dans la base, peut-être grâce à des triggers, ou bien utilisez les modèles documentés sur Internet « **Adjacency List Model** » ou « **Nested Set Model** »

15. Opérations en écriture

15.1. Importation de la base de développement app

L'équipe de développement a créé une première version de l'application app, dans le schéma app. Une sauvegarde a été faite en utilisant l'option `COMPRESS` sur l'assistant de sauvegarde. Le fichier **formation_schema_app_dev1.backup** contient cette sauvegarde. Si vous essayez de l'ouvrir avec un éditeur de texte vous pouvez voir que ce fichier ne peut être lu. C'est un format de sauvegarde inter à pg_dump et pg_restore.

Nous allons donc importer cette sauvegarde avec pg_restore, en utilisant l'assistant de pgAdmin. Cliquez (droit) sur le schéma app, Restaurer, puis aller chercher le fichier. On voit que la commande tapée est:

```
pg_restore --host localhost --port 5432 --username « aicha » --dbname
« formation2 » --schema app --verbose
« (...)/formation_schema_app_dev1.backup
```

Si vous lancez cette restauration une deuxième fois elle sera en échec sauf si vous cliquez l'option Nettoyer avant la restauration:

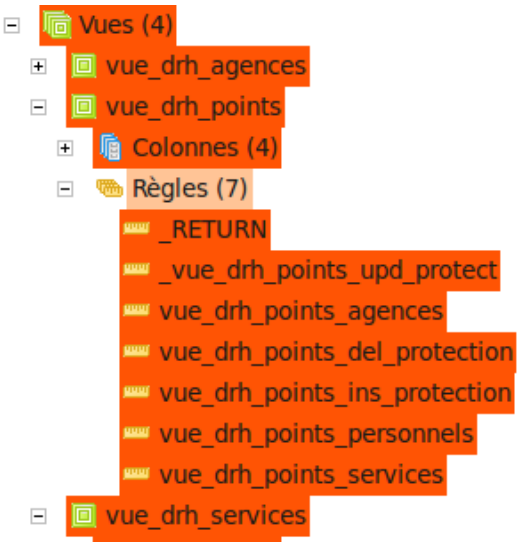
```
pg_restore --host localhost --port 5432 --username « aicha » --dbname
« formation2 » --clean --schema app --verbose
« (..)/formation_schema_app_dev1.backup
```

faites un rafraîchissement du schéma app, on y voit maintenant:

- 3 tables et séquences
- 4 vues

Les 4 vues permettent un accès aux données du schéma drh, en tous cas celles que les créateurs des vues nous autorisent à voir. Les tables sont là pour gérer les commandes de produits qui vont produire des points qu'il faudra impacter sur les tables du schéma drh.

15.2. Règles avancées sur les vues



Parmi les quatre vues 3 ont une règle simple qui redéfinit l'opération de lecture. Leur but est d'autoriser l'accès à certaines données du schéma drh aux utilisateurs du schéma app. La vue `vue_drh_points` possède par contre 7 règles.

Le but de cette vue est double:

- donner un aperçu global du nombre de points affecté à chaque employé et service ou agence à travers une requête d'`UNION` (ces points sont enregistrés dans le schéma `drh`)
- autoriser la **modification** de la colonne **points** depuis le schéma `app` pour répercuter ces points sur la bonne table dans le schéma `drh`.

Examinons chacune des ces règles:

- `vue_drh_points_agences` : si le nombre de points à changé dans une règle `update` et que l'entité est une agence alors mettre à jour le nombre de points dans la table agences du schéma drh. Les autres colonnes modifiées sont ignorées puisque non répercutées.
- `vue_drh_points_service`s: même chose que la précédente mais pour les services
- `vue_drh_points_personnels` : toujours la même chose mais pour les personnels, heureusement la colonne de points est bien sur la table `personnels` et pas sur `employes` ou `interimaires`.
- `_vue_drh_points_upd_protect` : empêche l'exécution de toute requête UPDATE, cette règle commence par un underscore, en ordre alphabétique elle est donc exécutée après toutes les autres, c'est un « `catch-all` » pour rejeter toutes les autres conditions d'`update`.
- `vue_drh_points_ins_protection` : empêche toute requête d'insertion (règle optionnelle)
- `vue_drh_points_del_protection` : empêche toute suppression de ligne (règle optionnelle)

> ☑ La manipulation des règles n'est pas toujours aisée, beaucoup de développeurs préfèrent l'utilisation de **triggers**.

➢ *Observez ce qui se passe quand vous tapez `UPDATE vue_drh_points SET points=0 WHERE entity='agences';`, essayez avec des utilisateurs différents, en faisant varier le nombre de points ou les lignes impactées. Essayez de mettre à jour d'autres colonnes.*

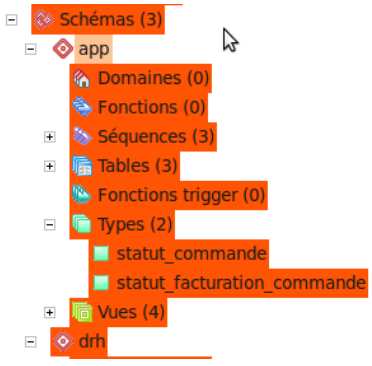
➢ *Essayez (en tant qu'utilisateur aicha) d'ajouter dans la vue `vue_drh_service`s la colonne `ser_code` **avant** la colonne `ser_parent` en utilisant la commande `CREATE OR REPLACE VIEW` qui est listée dans le Panneau SQL. Essayez ensuite d'ajouter cette colonne après la colonne `ser_parent`.*

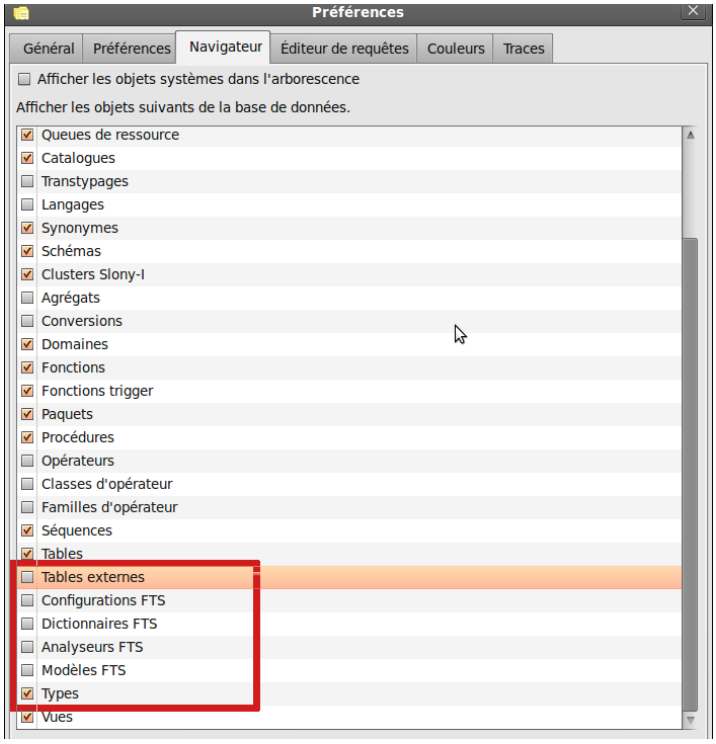
➢ *Lancez un explain graphique dans pgadmin sur `SELECT * from vue_drh_points;` et admirez le résultat.*

15.3. Modifier les objets affichés par pgadmin

Le schéma `app` contient des objets particuliers qui sont des types personnalisés (`ENUM` en l'occurence) mais pgadmin ne nous les montre pas. Par contre nous n'utilisons pas du tout la recherche plein texte mais les écrans de pgadmin nous montrent les objets `FTS`.

Nous allons donc modifier les affichages de pgadmin en allant dans le menu `Fichier > Préférences`.





15.4. Les Transactions

15.4.1. Opérations d'écriture en SQL

Les requêtes qui permettent de modifier la base en SQL sont assez peu nombreuses. Il y a :

- `INSERT` : http://docs.postgresqlfr.org/9.0/dml.html, insertion d'une ligne dans une table. Notez que le mots clef `RETURNING` permet de spécifier que vous voulez avoir en retour de la commande l'identifiant créé ou tout autre information. http://docs.postgresqlfr.org/9.0/sql-insert.html
- `UPDATE` : http://docs.postgresqlfr.org/9.0/dml-update.html, mise à jour d'une ou plusieurs lignes, notez que cette commande aussi supporte le mot clef `RETURNING`. http://docs.postgresqlfr.org/9.0/sql-update.html
- `DELETE` : http://docs.postgresqlfr.org/9.0/dml-delete.html, suppression d'une ou plusieurs lignes. http://docs.postgresqlfr.org/9.0/sql-delete.html
- `TRUNCATE` : un `delete` « amélioré » qui vide toute la table en une seule opération

Par défaut toutes ces opérations se font en mode `AUTO-COMMIT`, ce qui veut dire qu'elles sont des **transactions** qui ne contiennent qu'une seule commande.

> ☑ Vous n'avez tapé qu'une seule commande, mais des règles ou des triggers pourraient aussi générer des commandes en arrière plan comme nous avons pu l'apercevoir précédemment. Ces commandes sont alors dans votre transaction, c'est pourquoi il est important de voir que toute opération d'écriture est une transaction

15.4.2. FillFactor, Vacuum, HOT

Au niveau du stockage physique des tables, certains concepts peuvent nous aider à mieux comprendre ce qu'il se passe lors des opérations d'écriture.

Chaque ligne enregistrée dans la table peut être vue comme une ligne dans un tableau Excel. Quand vous modifiez une ligne une copie de cette ligne est ajoutée, par défaut en bas de la table. Quand votre modification sera terminée l'ancienne sera invalidée et la nouvelle deviendra la ligne officielle. Nous verrons qu'avec les transactions il est possible que plusieurs versions de la ligne existent pour différentes transactions en cours.

Ceci entraîne une répartition des lignes dans la table physique qui peut devenir très éloignée de leur ordre d'insertion initial. Après chaque mise à jour la ligne se retrouve en bas. La conséquence de ceci pourrait se faire sentir si vous effectuez des requêtes pour lesquelles plusieurs pages mémoire différentes de la table devront être chargées alors qu'avec un ordre plus proche de l'ordre d'insertion une seule page mémoire aurait suffit.

C'est ici qu'intervient le `FillFactor`. Donner un `FillFactor` à 100% pour un index ou une table signifie que vous ne laissez aucune ligne vide entre chaque ligne. Si vous diminuez ce fillfactor (facteur de remplissage) à 50% vous indiquez à PostgreSQL de laisser 50% d'espace vide dans la page mémoire (des lignes vides dans le tableau). Lors des mises à jour les lignes vides de la page mémoire de la ligne originale seront utilisées en priorité, plutôt que d'ajouter ces nouvelles lignes à la fin de la table.

> ☑ Le `Fillfactor` est par défaut à 100% (pas de perte d'espace). Si vous savez qu'une table subira un grand nombre de mises à jour n'hésitez pas à modifier son fillfactor ainsi que celui de ses index.

Le système `HOT` (Heap Overflow Tuple) intervient lui pour chaîner l'historique des lignes qui ont été annulées, quand vous accédez à une ligne vous accédez en fait à la ligne originale, vous êtes ensuite entraîné jusqu'à la bonne ligne par rebonds successifs car chaque version garde l'adresse de la version qui a invalidée la ligne. Cela permet d'éviter de toucher aux index (que nous étudierons bientôt) quand on modifie une ligne. Sans cela toute opération d'écriture devrait mettre à jour les index qui pointaient vers l'adresse de la ligne (Ce qui ralentit fortement les écritures).

Enfin les opérations de `VACUUM` sont des opérations de maintenance, qui aujourd'hui sont gérées par un processus de PostgreSQL (l'auto-vacuum). Lors de ces opérations plusieurs choses peuvent intervenir. Les données sont réorganisées dans la table, les index mis à jour, les espaces libres remis en ordre, etc. Des statistiques sur l'organisation des données dans la table seront peut-être aussi remis à jour. Dans un premier temps retenez qu'il est important que le Vacuum s'exécute régulièrement, et plus votre table subit de **mises à jour** et **suppressions** plus le vacuum devient important.

Signalons aussi l'existence de la commande SQL `CLUSTER` http://docs.postgresqlfr.org/9.0/sql-cluster.html qui réordonne les données d'une table sur un index de cette table. Ceci peut éviter des opérations de pages sur des grosses tables souvent utilisées sur un même index. Attention rapellez-vous qu'une requête sans `ORDER BY` renvoie les lignes dans un ordre non déterminé, utiliser `CLUSTER` pour forcer cet ordre par défaut n'est pas une garantie d'ordre.

15.4.3. ACID, MVCC et les transactions

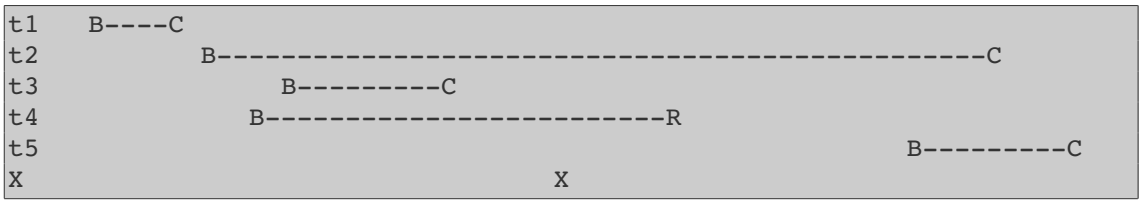
Une Transaction peut contenir plusieurs commandes. Pour effectuer plusieurs commandes dans une même transaction on utilise les mots clefs `BEGIN`, `COMMIT` et `ROLLBACK`. Peut-être connaissez vous le raccourci mnémotechnique ACID, il représente tout ce qu'une transaction gère:

- **A**tomicité : tout ce qui est compris entre `BEGIN` et `COMMIT` aura lieu ou n'aura pas lieu (`ROLLBACK` au lieu de `COMMIT`), il n'y aura pas d'exécution partielle, ce n'est pas découpable, c'est donc atomique.
- **C**ohérence : La base donnée était dans un état cohérent avant, elle le sera aussi après. Pensez par exemple au contraintes d'intégrité, au cours de la transaction elle peuvent être suspendues, à la fin de la transaction elles devront être vérifiées.
- **I**solation : nous y reviendrons, c'est la **le vrai problème.** L'isolation de la transaction par rapport aux autres transactions concurrentes
- **D**urabilité : Une fois le `COMMIT` effectué et validé par la base de donnée vos opérations d'écritures sont réputées validées, elles ne peuvent plus être annulées, le SGBD doit assurer leur pérennité.

> ☑ Si vous vous amusez à modifier des bits dans les fichiers binaires de stockage de la base, vous pourrez modifier une donnée enregistrée par PostgreSQL. Une fois les données stockées sur disque PostgreSQL ne vérifie plus l'intégrité de ces données physiques comme pourrait le faire un , `DBCC CHECKDB` sur SQL Server.

Le principal problème est donc l'isolation des transactions. La référence en la matière est la page de documentation du modèle MVCC de PostgreSQL: http://docs.postgresqlfr.org/9.0/mvcc.html MVCC signifie **M**utli**V**ersion **C**oncurrency **C**ontrol.

Cette page explique bien le problème et les différents « anomalies » que l'on peut rencontrer. Nous ajouterons donc simplement quelques compléments. Le problème de l'isolation concerne les transactions concurrentes, il s'agit de transactions qui tournent en parallèle, en même temps. La première transaction n'est pas du tout obligée d'être la première à se terminer. Représentons par exemple quelques transactions sur un frise chronologique :



B signifie `Begin`, R `Rollback` et C `Commit`. X représente un moment dans le temps à partir desquels des problèmes d'isolation peuvent survenir pour les transactions qui tournent encore (pour les niveaux d'isolations supérieurs à `Uncommited Read` qui n'existe pas dans PostgreSQL).

La transaction `t1` n'a pas de problème d'isolation, elle est seule. Au moment ou `t3` se termine `t2` et `t4` pourraient avoir des problèmes. Au moment ou `t4` fais un Rollback cela ne pose de problèmes à personne. Au moment où `t1` se termine `t5` qui a déjà commencée pourrait avoir un problème.

Par défaut nous sommes dans le deuxième niveau parmi les 4 définis sur la page http://docs.postgresqlfr.org/9.0/transaction-iso.html, ce niveau `Commited Read` autorise, après les X indiqués sur le schémas, deux types de problèmes: les lectures non reproductibles et les lectures fantômes.

> ☑ A partir du moment ou on tape `BEGIN` on fais « sauter » le mode auto-commit. Il faudra donc à un moment ou un autre taper `ROLLBACK` ou `COMMIT` dans ces fenêtres, ou atteindre un `ROLLBACK` par un timeout de la transaction ou la fermeture de la fenêtre (donc de la connexion)

☑ **Règle #1** : refaites toutes vos opérations de lecture après le début de la transaction, tout ce que votre programme a pu lire dans la base AVANT le `BEGIN` a peut-être déjà été modifié.

☑ **Règle #2** : Une transaction sans **locks** est quasi assurée de provoquer des erreurs fonctionnelles. C'est une règle beaucoup trop ignorée (Au point que depuis MySQL 5.1 les locks sont automatiques sur les sélections dans les transactions MySQL-- ne cherchez pas ce qui se passe dans MySQL 5.0, c'est instable).

☑ **Règles #3**: Attendez vous à ce qu'une transaction puisse être annulée par le moteur. Pour des problèmes d'isolation ou de deadlock. Votre programme devrait peut-être essayer de jouer les transactions plusieurs fois

☑ **Règles #4**: Même avec le niveau d'isolation maximum des problèmes peuvent survenir, réfléchissez bien à ces problèmes, faites des tests

Pour régler les problèmes d'isolation le principal outil est la pose de verrous (`LOCKS`). Les transactions concurrents se retrouvent alors mises en attente quand elles demande l'accès à des ressources qui sont utilisées par d'autres transactions.

La pose de verrous pose quelques problèmes, le principal est le `DEADLOCK`.

Si nos deux transactions posent chacune de leur côté des verrous, puis qu'ensuite elles tentent de poser des verrous sur les lignes précédemment verrouillées par la transaction concurrente, elles se retrouvent chacune à attendre la libération de verrous de l'autre. Dans ce cas le SGBD devrait annuler une des transactions (`rollback`) en émettant un message. Au niveau des langages de programmation on retrouvera des `Deadlock Exception` par exemple.

Notez qu'il existe en interne de nombreux niveaux de verrouillages correspondants à diverses opérations, il est par exemple impossible de modifier une table qu'une opération de maintenance est en train de réorganiser sur le disque. La référence complète en la matière est ici: http://docs.postgresqlfr.org/9.0/explicit-locking.html. Remarquez le **pg_advisory_lock(id)** qui permet de poser un verrou applicatif. PostgreSQL devient alors un moyen de stockage de sémaphores pour votre application.

Pour en revenir aux transactions nous pouvons aussi changer le niveau d'isolation en passant au niveau le plus élevé, `Serializable`. Schématiquement si nous reprenons la frise de départ le SGBD va tenter d'émuler Une sérialisation dans le temps des transactions:

```
t1   B----C
t2          B----C
t3                 B---------C
t4                           B---------C
```

Il ne le fais pas aussi schématiquement en repoussant les transactions mais plutôt en détectant une opération au sein d'une des transactions qui rendrait ce schéma impossible.

```
T1   B----C
t2          B-----------------------------------------------C
t3               B---------C
t4             B------------------------R
t5                                                  B-----R
X
```

Ici par exemple `t3` et `t4` n'ont pas touché à des données impactant `t2,` le moteur n'a rien dit. `t4` avait peut être touché des éléments impactant `t2` mais elle a été annulée par l'utilisateur. Par contre au commit de t2 le moteur a repéré que des éléments touchés par `t5` étaient impactés, `t5` a donc été automatiquement annulée par le moteur.

- *Essayons de visualiser réellement ces problèmes, ouvrez deux fenêtres SQL sur pgAdmin, cela représente deux sessions différentes, si vous tapez `BEGIN` dans les deux vous avez **deux transactions concurrentes**.*

Nous listons ici les commandes dans l'ordre chronologique

Fenêtre 1

```
BEGIN;
```

la transaction est démarrée. Videz le texte (ou commentez le) et écrivez maintenant:

```
SELECT
com_id,per_id,com_points
FROM app.commandes
WHERE
com_statut='expédiée'
AND
com_statut_facturation='no
n facturée'
AND com_points=100
ORDER BY com_id LIMIT 3;
```

ici on sélectionne 3 factures, sans doute pour les traiter, j'obtiens les numéros 3082 et 3089 et 3099 avec 100 points.

```
UPDATE app.commandes
 SET com_points =
com_points+1
 WHERE com_id=3089;
UPDATE app.commandes
 SET com_points =
com_points-1
 WHERE com_id = 3082;
```

Retapons le select original

```
SELECT
com_id,per_id,com_points
FROM app.commandes
WHERE
com_statut='expédiée'
AND
com_statut_facturation='no
n facturée'
AND com_points=100
ORDER BY com_id LIMIT 3;
```

on obtient des nouvelles lignes (3102 et 3112) en plus de la 3099, 3082 et 3089 n'y sont plus. Et là on termine:`COMMIT;`

Fenêtre 2

```
BEGIN;
```

Deuxième transaction démarrée. Videz le texte

on tape la même commande:

```
SELECT com_id,per_id,com_points
FROM app.commandes
WHERE com_statut='expédiée'
AND com_statut_facturation='non
facturée'
AND com_points=100
ORDER BY com_id LIMIT 3;
```

et on obtient les mêmes résultats

Ici on ne fais rien, pendant que les 2 requêtes `update` se font à côté ce programme de haut niveau ayant commencé une transaction est peut-être en attente dans les cycles cpu.

(...)

Ici aussi on refais le select original

```
SELECT com_id,per_id,com_points
FROM app.commandes
WHERE com_statut='expédiée'
AND com_statut_facturation='non
facturée'
AND com_points=100
ORDER BY com_id LIMIT 3;
```

on a toujours les lignes 3082,3089 et 3099. Normal on est isolé.

----- A partir de maintenant tout change:

Lecture fantôme : **Si on refais la sélection** on voit les 3099,3082 et 3089 : une même requête donne un résultat différent:

Lecture de données validées! Si je fais une requête

```
SELECT com_points
FROM app.commandes
WHERE com_id in (3082,3089)
```

on voit les scores à 99 alors qu'ils étaient à 100 au début de cette transaction et que je n'ai fais aucun `update`!

```
COMMIT; ou ROLLBACK;
```

Testons la mise en places de LOCKS grâce par exemple au mot clef `FOR UPDATE` (http://docs.postgresqlfr.org/9.0/sql-select.html#sql-for-update-share) ajoutés dans mes sélections:

Fenêtre 1

```
BEGIN;
```

la transaction est démarrée. Videz le texte (ou commentez le) et écrivez maintenant:

```
SELECT
com_id,per_id,com_points
FROM app.commandes
WHERE
com_statut='expédiée'
AND
com_statut_facturation='no
n facturée'
AND com_points=100
ORDER BY com_id LIMIT 3
 FOR UPDATE;
```

ici on sélectionne 3 factures, sans doute pour les traiter, j'obtiens les numéros 3099, 3102 et 3112 avec 100 points.

```
UPDATE app.commandes
 SET com_points =
com_points+1
 WHERE com_id=3115;
```

Et là on termine:

```
COMMIT;
```

Fenêtre 2

```
BEGIN;
```

Deuxième transaction démarrée. Videz le texte

on tape la même commande:

```
SELECT com_id,per_id,com_points
FROM app.commandes
WHERE com_statut='expédiée'
AND com_statut_facturation='non
facturée'
AND com_points=100
ORDER BY com_id LIMIT 3
 FOR UPDATE;
```

et on obtient (…)

une longue attente. Cette requête est en attente de la libération du verrou sur ces lignes qui vient d'être posé par la requête de gauche.

Et là enfin j'ai le résultat de ma requête.

Je peux continuer sans problèmes de lectures fantômes **je lirai les données validées, j'ai attendu pour ça**.

(…) `COMMIT` ou `ROLLBACK`.

Testons maintenant le niveau SERIALIZABLE

Fenêtre 1

```
SET TRANSACTION ISOLATION
LEVEL SERIALIZABLE;
BEGIN;
```

la transaction est démarrée. Videz le texte (ou commentez le) et écrivez maintenant:

```
SELECT
com_id,per_id,com_points
FROM app.commandes
WHERE
com_statut='expédiée'
AND
com_statut_facturation='n
on facturée'
AND com_points=100
ORDER BY com_id LIMIT 3
```

ici on sélectionne 3 factures, sans doute pour les traiter, j'obtiens les numéros 3099, 3102 et 3112 avec 100 points.

```
UPDATE app.commandes
 SET com_points =
com_points+1
 WHERE com_id=3112;
```

On termine:

```
COMMIT;
```

Fenêtre 2

```
SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
BEGIN;
```

Deuxième transaction démarrée. Videz le texte

```
SELECT com_id,per_id,com_points
FROM app.commandes
WHERE com_statut='expédiée'
AND com_statut_facturation='non
facturée'
AND com_points=100
ORDER BY com_id LIMIT 3
```

on tape la même commande:

```
UPDATE app.commandes
 SET com_points = com_points+1
 WHERE com_id=3112;
```

On obtient un **ERROR: could not serialize access due to concurrent update**

Ce qui nous fait un `Rollback` automatique.

Vous pouvez remarquer l'absence de **`FOR UPDATE`** dans les requêtes de sélection.

Cependant si vous re-testez ce scénario sans faire de requête `SELECT` vous remarquerez que la transaction n'échoue plus. Utilisez des requêtes en lecture sur les éléments qui devraient rester intacts pour forcer la pose de verrous par PostgreSQl sur les lignes impactées.

16. Fonctions et Déclencheurs (triggers)

Sébastien François et Martine on bien travaillé sur leur base app, mais ils se rendent compte qu'il y a beaucoup de requêtes de mises à jour à faire dès qu'ils insèrent une ligne de commande. Il faut recalculer le total de la commande, attribuer des points en fonction de ce total (on leur a fournit le calcul de points, il se fait par tranches de montants de commande, et on doit ajouter un nombre fixe de points aux services et agences en fonction du nombre de commandes), il faut donc aussi faire des requêtes de mise à jour de nombre de points sur la vue vue_drh_points.

Sébastien ayant déjà aperçu le détail du schéma drh il a pu y observer des mises à jour automatiques grâces à des triggers. Il y a par exemple une fonction dans le schéma public qui mets à jour les dates de modification:

```
CREATE OR REPLACE FUNCTION public.update_datemodif_column()
  RETURNS trigger AS
$BODY$
  BEGIN
    NEW.date_modification = NOW();
    RETURN NEW;
  END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

Sachant que pour les dates de création la valeur par défaut du champ à `now()` suffit, il n'y a pas besoin d'un déclencheur pour cela. Ce déclencheur de modification de date de modification est branché sur la table `employes` de cette façon:

```
CREATE TRIGGER emp_update_date_modification
  BEFORE UPDATE
  ON drh.employes
  FOR EACH ROW
  EXECUTE PROCEDURE public.update_datemodif_column();
```

On voit que les fonctions utilisées par les déclencheurs sont définis dans un langage (ici plpgsql) et doivent renvoyer un '`trigger`'. Ici la fonction manipule un objet spécial nommé **NEW** et qui contient la ligne en cours de modification.

Il y a deux autres branchements de triggers sur la table `employes`:

```
CREATE TRIGGER emp_insert_code
  BEFORE INSERT
  ON drh.employes
  FOR EACH ROW
  EXECUTE PROCEDURE drh.handle_employe_code();
CREATE TRIGGER emp_update_code
  BEFORE UPDATE
  ON drh.employes
  FOR EACH ROW
  WHEN (new.per_id <> old.per_id) OR (new.ser_id <> old.ser_id)
  EXECUTE PROCEDURE drh.handle_employe_code();
```

Ces deux déclencheurs sont branchés sur la même fonction. La fonction est donc un peu plus complexe, elle doit prendre en compte le fait qu'elle peut être lancée à des moments différents:

```
CREATE OR REPLACE FUNCTION drh.handle_employe_code()
  RETURNS trigger AS
$BODY$
```

```
  DECLARE
    service_code  character varying;
  BEGIN
    RAISE NOTICE E'\n    Operation: %\n    Schema: %\n    Table: %',
        TG_OP,
        TG_TABLE_SCHEMA,
        TG_TABLE_NAME;
    -- this function can be called from services in UPDATE (only if id
or ser_id altered)
    -- or INSERT mode
    -- or from an updated ser_code on services
    -- deleted service will be handled by CASCADE SET ser_id=1
launching an employes update
    -- updated service id will be handled by CASCADE UPDATE launching
an employes update
    IF (TG_OP = 'UPDATE') THEN
      IF (TG_TABLE_NAME = 'employes') THEN
          SELECT ser_code INTO service_code FROM drh.services WHERE
ser_id=NEW.ser_id;
          NEW.emp_code := service_code || '-' ||
trim(to_char(NEW.per_id,'0000'));
          RETURN NEW;
      ELSIF (TG_TABLE_NAME = 'services') THEN
          service_code = NEW.ser_code;
          UPDATE drh.employes
            SET emp_code=service_code || '-' ||
trim(to_char(per_id,'0000'))
            WHERE ser_id=NEW.ser_id;
          RETURN NEW;
      END IF;
    ELSIF (TG_OP = 'INSERT') THEN
      IF (TG_TABLE_NAME = 'employes') THEN
          SELECT ser_code INTO service_code FROM drh.services WHERE
ser_id=NEW.ser_id;
          NEW.emp_code := service_code || '-' ||
trim(to_char(NEW.per_id,'0000'));
          RETURN NEW;
      END IF;
    END IF;
    RETURN NULL;
  END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

Cette fonction met à jour le code employes en fonction du service auquel est affecté l'employé. On voit qu'en plus de **NEW** un trigger peut être amené à jouer avec **OLD** qui contient la ligne telle qu'elle était avant la modification en cours.

Les déclencheurs sont documentés ici: http://docs.postgresql.fr/9.0/triggers.html, http://docs.postgresqlfr.org/9.0/sql-createtrigger.html et ici http://docs.postgresql.fr/9.0/plpgsql-trigger.html pour les points les plus importants. Quelques points à noter:

- `OLD` et `NEW` ne sont pas toujours présent, sur un `INSERT` on aura que `NEW`, sur un `UPDATE` on a `NEW` et `OLD`, sur un `DELETE` on aura `OLD` uniquement
- les déclencheurs se lancent soit avant (`BEFORE`) soit après (`AFTER`) un événement

- on peut associer plusieurs déclencheurs à un même événement (ce qui n'est pas le cas sur MySQL par exemple), ils sont exécutés en ordre alphabétique
- les déclencheurs peuvent être de niveau ligne `FOR EACH ROW` ou de niveau requête `FOR EACH STATEMENT`.
- Renvoyer `FALSE` peut permettre d'annuler la requête dans certain type de triggers.

> ☑ Si vous utilisez un trigger pour recalculer des résultats (comme une requête `GROUP BY`) et les stocker dans une table statistique, travaillez au niveau requête et pas au niveau ROW. Sinon une requête qui mets à jour 10 000 lignes lancera 10 000 fois le calcul au lieu de le faire une seule fois à la fin.

Le premier lien de documentation fournit un exemple de trigger en C. Le langage le plus utilisé est le pl/pgSQL. Mais vous pouvez aussi utiliser d'autres langages, comme le pl/Perl http://docs.postgresqlfr.org/9.0/plperl.html, le pl/Python http://docs.postgresqlfr.org/9.0/plpython.html, le pl/Tcl http://docs.postgresqlfr.org/9.0/pltcl.html.

> ☑ Un déclencheur ne fait qu'associer une fonction particulière (qui doit renvoyer des valeurs particulières). Mais on peut aussi écrire des fonctions qui ne soient pas des triggers. On peut écrire ces fonctions dans ces mêmes langages, mais on peut aussi écrire des fonctions SQL.

Fort de tous ces renseignements nous développeurs de l'application app ont ajoutés quelques fonctions qu'ils ont dumpés dans un fichier.

16.1. Importer les fonctions et déclencheurs pour app

Nous allons utiliser le fichier formation_schema_app_dev2.backup qui a été crée avec l'assistant de sauvegarde, c'est un dump COMPRESS+UTF8 sur les objets du schéma app uniquement.

Clic droit sur la base formation `> Restaurer...` :

- on prend le fichier formation_schema_app_dev2.backup
- on coche « `Nettoyer avant la restauration` »
- on coche aussi sur le troisième onglet `Désactiver > trigger`. Prenez l'habitude dès maintenant de désactiver les triggers pendant les restaurations, imaginez ce qu'il se passera quand des triggers seront actifs sur la base et que vous lancerez une restauration...
- Remarquez l'option « `Nombre de processus/threads` » cette option peut vous sauver la vie sur une restauration d'un gros fichier de backup en passant la procédure de 10 heures à 1 heures – ce qui peut vous ramener dans un fenêtre de tir raisonnable. C'est une option nouvelle qui va permettre de paralléliser les restaurations qui peuvent l'être sur plusieurs processeurs, il faut donc déjà disposer d'une machine à processeurs multiples. Vous pouvez tenter des modifications de valeurs pour voir si cela impacte le temps de restauration.

On constate l'apparition de 7 fonctions et de 4 fonctions trigger. Si vous voulez retrouver la table qui lance ces fonctions trigger regardez l'onglet Objets dépendants de ces fonctions. Vous remarquerez l'importance d'utiliser une nomenclature parlante dans le nom des triggers. On peut voir la définition de l'association des triggers à une table dans le menu triggers accessible en dépliant (+) les tables dans le navigateur d'objets.

☑ Remarquez que certains triggers sont conditionnés dès l'association du trigger à un événement de la table. Ceci afin d'éviter le lancement d'une fonction si elle n'est pas nécessaire.

- *Q1: A l'aide de l'éditeur de table de pgAdmin essayez d'insérer une ligne de commande pour une commande existante, quelles sont les informations minimales que vous avez besoin d'entrer.*
- *Q2: Examinez la fonction `points_from_amount(numeric)`. Quel est son langage? Que fait elle? Essayer de l'utiliser dans une requête. Essayez un nombre négatif. Corrigez la fonction pour qu'un montant négatif ne renvoie pas de points.*
- *Q3: Regardez la fonction `app.update_commande_amounts(numeric,integer)`, Essayez de l'utiliser dans une requête.*
- *Q4: Modifiez la fonction SQL `app.update_commande_amounts(numeric,integer)` pour que son utilisation renvoie un résultat si on l'utilise avec un `SELECT`.*
- *Q5: Écrivez une fonction SQL qui prends deux chaînes en paramètre et qui renvoie la concaténation des trois premiers caractères de ces chaînes*
- *Q6: Question subsidiaire pour ceux qui avancent très vite. Créez une table statistiques_commandes dans laquelle vous stockerez par personnel le nombre commandes et la somme totale des commandes. Écrivez un déclencheur (trigger) qui maintiendra ces statistiques en cas d'`INSERT`, d'`UPDATE`, de `DELETE` ou de `TRUNCATE` sur la table commandes*

Solutions:

Q1: il faut entrer au minimum `lic_quantite`, `lic_prix-unitaire`, `com_id` et `pro_id`. Donc la quantité, le prix et l'identifiant du produit plus la commande à laquelle on appartient. Tout le reste est calculé par des triggers. Faites `F5` pour rafraîchir la ligne. Vous pouvez remonter à la commande, et aux tables de la personne (employé ou intérimaire) et de son service (ou agence) et voir que le nombre de points a été répercuté partout.

Q2: La fonction est écrite en SQL (et non en pl/pgsql). Elle calcule simplement le nombre de points à affecter à une commande en fonction d'un montant reçu en paramètre.

```
SELECT app.points_from_amount(125.0);
```

Les nombres négatifs donnent 5 points, l'erreur est en fait simple à corriger. On recopie le code SQL depuis le panneau SQL quand on sélectionne la fonction et on corrige **$1=0** en **$1<=0**:

```
CREATE OR REPLACE FUNCTION app.points_from_amount(numeric)
  RETURNS integer AS
$BODY$
    SELECT CASE
      WHEN ($1 IS NULL OR $1<=0) THEN 0
      WHEN ($1<50) THEN 5
      WHEN ($1>=50 AND $1<100) THEN 15
      WHEN ($1>=100 AND $1<200) THEN 50
      WHEN ($1>=200 AND $1<500) THEN 100
      ELSE 1000 END;
$BODY$
  LANGUAGE sql IMMUTABLE
  COST 100;
```

☑ remarquez le mot `IMMUTABLE`: cela signifie que pour un même nombre en entrée cette fonction renverra **TOUJOURS** le même résultat. La valeur par défaut est `VOLATILE` http://docs.postgresqlfr.org/9.0/sql-createfunction.html Ceci peut grandement améliorer la vitesse d'exécution des fonctions qui ne modifie pas les données et qui ne dépendent pas des données autres que leurs paramètres (cache).

Q3 :

```
select app.update_commande_amounts(42,3079);
```

La fonction ne renvoie rien mais modifie la total de la commande 3079 à 42.

Q4:

Pour que cette fonction nous renvoie quelque chose nous pouvons utiliser la clause `RETURNING` de la commande `UPDATE`. Mais il faut alors définir un type retour pour la fonction. La commande `CREATE OR REPLACE FUNCTION` ne suffit alors, plus, il faut utiliser un `DROP` avant.

```
DROP FUNCTION app.update_commande_amounts(numeric, integer);
CREATE OR REPLACE FUNCTION app.update_commande_amounts(numeric,
integer)
  RETURNS integer AS
$BODY$
  UPDATE app.commandes SET
    com_total_ht = $1,
    com_total_tva = $1 * com_taux_tva,
    com_total_ttc = $1 * (1+com_taux_tva)
  WHERE com_id=$2
  RETURNING com_id;
$BODY$
  LANGUAGE sql VOLATILE
  COST 100;
select app.update_commande_amounts(43,3079);
```

Mais nous pouvons ajouter quelques variations dans le type de résultats renvoyés

```
DROP FUNCTION app.update_commande_amounts(numeric, integer);
CREATE OR REPLACE FUNCTION app.update_commande_amounts(numeric,
integer)
  RETURNS RECORD AS
$BODY$
  UPDATE app.commandes SET
    com_total_ht = $1,
    com_total_tva = $1 * com_taux_tva,
    com_total_ttc = $1 * (1+com_taux_tva)
  WHERE com_id=$2
  RETURNING com_id,com_total_ht,com_total_tva,com_total_ttc;
$BODY$
  LANGUAGE sql VOLATILE
  COST 100;
select app.update_commande_amounts(44,3079);
```

Ou encore

```
DROP FUNCTION app.update_commande_amounts(numeric, integer);
CREATE OR REPLACE FUNCTION app.update_commande_amounts(numeric,
integer)
```

```
  RETURNS commandes AS
$BODY$
  UPDATE app.commandes SET
    com_total_ht = $1,
    com_total_tva = $1 * com_taux_tva,
    com_total_ttc = $1 * (1+com_taux_tva)
  WHERE com_id=$2;
  SELECT * FROM commandes WHERE com_id=$2;
$BODY$
  LANGUAGE sql VOLATILE
  COST 100;
select app.update_commande_amounts(45,3079);
```

Q5:

```
CREATE OR REPLACE FUNCTION app.monsubstrconcat(character varying,
character varying)
  RETURNS character varying AS
$BODY$
  SELECT coalesce(substring($1,1,3),'')
    || coalesce(substring($2,1,3),'');
$BODY$
  LANGUAGE sql IMMUTABLE;
select app.monsubstrconcat('abcdef','ghijk') as t1,
 app.monsubstrconcat('ab','ghijk') as t2,
 app.monsubstrconcat('','ghijk') as t3,
 app.monsubstrconcat(NULL,'ghijk') as t4,
 app.monsubstrconcat('abcde',NULL) as t5;
```

Q6: pas de solutions, essayez vraiment tout seul.

17. Indexation

17.1. Pourquoi indexer?

La principale raison d'indexer une base est que c'est grâce aux index que vous pourrez résoudre l'immense majorité des problèmes de requêtes lentes (en dehors de l'écriture de requêtes grossièrement fausses).

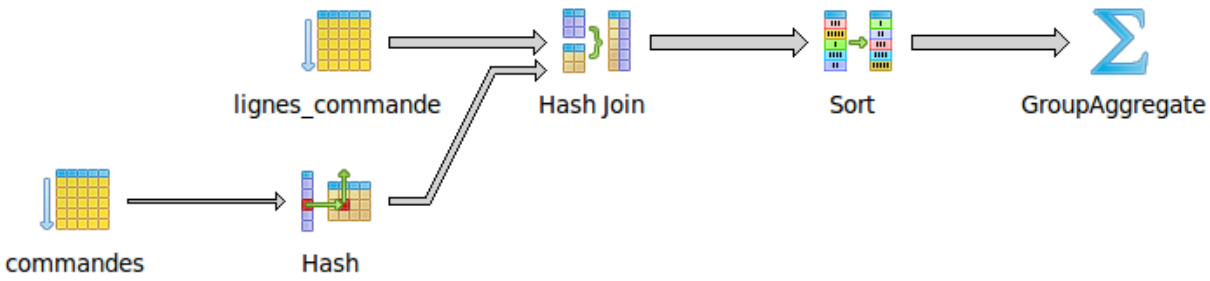
17.1.1. Visualiser les effets de l'indexation et des ANALYZE

Prenons une requête assez simple:

```
SELECT co.com_id,sum(lc.lic_total) as "sommes induites"
FROM app.lignes_commande lc
INNER JOIN app.commandes co ON lc.com_id = co.com_id
WHERE lc.lic_quantite >= 10
GROUP BY co.com_id
ORDER BY 1 DESC;
```

Le but de cette requête est de lister les commandes qui comportent des lignes de commande avec des quantités supérieures à 10, tout en affichant le somme totale des de ces lignes dans la commande.

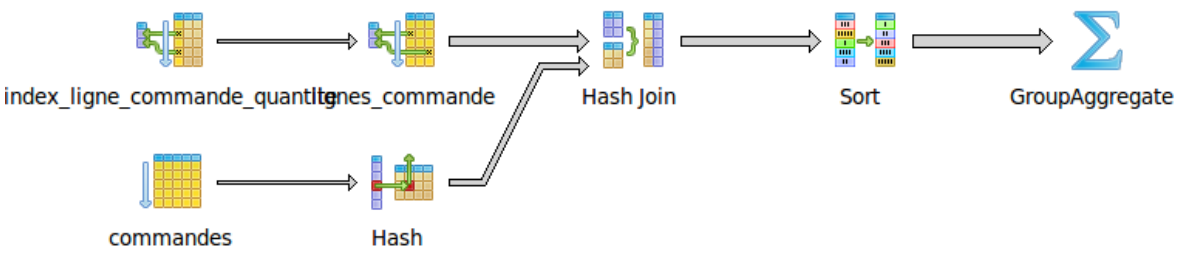
Tapons cette requête avec un EXPLAIN devant, ou bien choisissons tout simplement d'utiliser le bouton explain de pgadmin. Nous obtenons ce schéma:



Créons maintenant deux index sur les quantités de la table lignes_commande en espérant que l'un des deux soit pris en compte:

```
CREATE INDEX index_ligne_commande_quantite
  ON app.lignes_commande (lic_quantite);
CREATE INDEX index_ligne_commande_commande_quantite
  ON app.lignes_commande (com_id, lic_quantite);
```

Relançons la requête et son `EXPLAIN`:



Nous allons supprimer la plus grande partie des lignes de la table lignes_commande en tapant:

```
DELETE FROM app.lignes_commande WHERE lic_quantite<10;
```

Cette commande peut prendre un peu de temps 3680 lignes sont supprimées et tous les triggers ont été pensés au niveau ligne et pas au niveau requête.

La physionomie de la table lignes_commandes est complètement différente maintenant. Nous allons forcer PostgreSQL à se rendre compte de ce fait en tapant ces deux commandes (une par une):

```
VACUUM ANALYZE app.lignes_commande;
VACUUM ANALYZE app.commandes;
```

Relançons la requête; Nous ré-obtenons le premier schéma d'explain. PostgreSQL grâce à l'analyse de la table sait qu'il perdra plus de temps à filtrer la table lignes_commande.

☑ D'une façon plus générale l'utilisation ou non des index dépends non seulement de leur existence mais aussi et surtout des **estimations de coût du moteur SQL** en fonctions des statistiques qu'il collecte sur les tables.

☑ Pour travailler l'indexation de votre base **il vous faut donc une base alimentée en données**, sous un forme réaliste.

17.1.2. Génération de données

Il nous faut une base avec des données réalistes. Les développeurs de app nous on fournit un petit script php à exécuter sur la ligne de commande qui peut alimenter les commandes et lignes de commandes avec des données plus ou moins réalistes. Nous allons l'utiliser pour « charger » notre base.

```
# installtion du php en mode ligne de commande
# le package peut aussi se nommer php-cli
sudo apt-get install php5-cli
cd /repertoire/de/la/formation
```

On peut éditer le fichier pour modifier les constantes en tête du fichier et examiner ce que fais le script. Essentiellement il choisit au hasard une personne, lui créé une commande, lui mets des produits dedans. En boucle. Il y a des requêtes intéressantes sur la façon de choisir des personnes aléatoirement, de travailler avec des valeurs aléatoires, et de générer des dates aléatoires.

Pour éditer le fichier utilisez nano (console) ou gedit (graphique)

```
gedit populate_app.php
```

Exécutons le

```
php populate_app.php
```

A titre d'exemple il m'aura fallu 20minutes pour générer 10 000 commandes sur une machine bien chargée et sans optimisations du serveur PostgreSQL.

17.1.3. Comment fonctionne un index?

Un index peut être vu comme un fichier distinct de la table qui prend le ou les champs qui vous lui demandez d'indexer, qui les stocke dans l'ordre, puis qui stocke le numéro de ligne (l'adresse réelle) de la ligne correspondante de la table. Un index de type unique s'assurera qu'il n'y a qu'une ligne correspondante. Un autre type d'index pourra stocker l'adresse de plusieurs lignes correspondant au critère.

La **cardinalité** d'un index est le nombre d'éléments distincts qu'il doit indexer (par exemple un index des pays des employés devrait stocker 10 valeurs).

Un index peut être vu comme un annuaire, vous y entrez avec vos critères (le nom de la ville et le nom et prénom de la personne), vous recherchez d'abord le nom de la ville (ordre alphabétique), puis dans cette ville, toujours en ordre alphabétique le nom de la personne. Enfin vous recherchez le prénom. Et vous trouvez en face l'adresse (ou le numéro de téléphone).

Un index peut donc aussi être construit sur plusieurs champs

```
CREATE INDEX idx_annuaire
ON adresses (ville,nom,prenom);
```

Ce qui serait différent de ce second annuaire:

```
CREATE INDEX idx_annuaire_nom
ON adresses (nom,prenom,ville);
```

Dans lequel vous devriez commencer par rechercher le nom et le prénom de la personne puis filtrer par ville.

Pour le moteur SQL c'est la même chose. Si vous luis demandez de lister toutes les personnes de la ville NANTES ayant le nom DURAND il va utiliser idx_annuaire, filtrer sur la ville NANTES, puis sur le nom DURAND, **ignorer** le prenom et sortir l'ensemble des lignes communes à cette ville et ce prénom.

> ☑ Un index sur `champ1,champ2,champ3` rend donc inutile un index sur `champ1,champ2` et un index sur `champ1`.

Par contre si vous lui demandez Toutes les personnes dont le prénom est Pierre il n'utilisera ni `idx_annuaire` ni `idx_annuaire_nom`, aucun de ces deux index ne lui permet de commencer par filtrer sur les prénoms

> ☑ Un index sur `champ1,champ2` est très différent d'un index sur `champ2,champ1`, le moteur SQL ne pourra pas utiliser le premier s'il a besoin de filtrer sur champ2 uniquement

En terme de stockage physique PostgreSQL utilise des `B-TREE` par défaut. Le support des index de type `hash` est autorisé, mais souvent inutile à cause des performances de l'implémentation des b-tree, il y a aussi des problèmes de reconstruction des index de type hash, après une restauration il faut lancer des commandes de reconstruction d'index pour les hash. On les voit donc très rarement. Enfin des types d'index avancés `GIST` et `GIN` existent, souvent pour des types de données complexes comme des coordonnées géographiques par exemple.

17.1.4. Taille des index

Effectuons un requête dans le catalogue pour regarder la taille de nos tables et index:

- http://www.postgresql.org/docs/9.0/interactive/disk-usage.html

- http://www.postgresql.org/docs/9.0/interactive/functions-admin.html#FUNCTIONS-ADMIN-DBSIZE

```
SELECT relname, relpages,
pg_relation_filepath(oid),
pg_size_pretty(pg_table_size(oid)) as "table size",
pg_size_pretty(pg_indexes_size(oid)) as "indexes size",
pg_size_pretty(pg_total_relation_size(oid))as "total"
FROM pg_class
ORDER BY relpages DESC;
```

Vous remarquerez qu'une partie de ces informations sont disponibles dans l'onglet statistiques des objets tables sur pgAdmin.

Le point le plus important à noter est que notre table lignes_commandes avec seulement trois index occupe plus de place pour stocker ses index que pour stocker les données de la table.

> ☑ Les index sont utiles pour améliorer la vitesse d'exécution des requêtes mais ils ont un coût non négligeable en terme d'espace disque (et de temps d'insertion/modification)

17.1.5. Trouver le bon index

Côté PostgreSQL trouver le bon index est le métier de l'analyseur de requêtes, aidé en cela par le sous-service de statistiques qui lui renvoie des informations sur les cardinalités de la table et la répartition des différents éléments.

Du côté du développeur et plus encore de l'administrateur de la base de donnée il faut trouver les index qui serviront réellement et les créer.

Les index vont intervenir dans une requête:

- sur les conditions de **Tri** du résultat `ORDER BY`. (Si je vous demande de me donner les habitants de Nantes dans l'ordre alphabétique et que je vous donne l'annuaire ce sera plus simple que sans l'annuaire avec juste la liste des habitants non ordonnée)
- sur les jointures entre tables (choix des algorithmes nested loop, merge joins, sequential scan, etc). Notez que les clefs étrangères génèrent la création automatique d'index.
- Sur les filtrages divers et les conditions de regroupement effectués dans la requête

Notons tout de suite que l'enfer en terme d'indexation est un programme qui affiche un tableau de données filtrées et muni de nombreuses colonnes que l'on peut toutes choisir pour ordonner le résultat.

Maintenant examinons cette requête:

```
SELECT
  date_part('month',com_date) as "mois",
  round(sum(com_total_ht),2) as "total mensuel"
FROM app.commandes
WHERE date_part('year',com_date)=2011
  AND date_part('month',com_date) BETWEEN 1 AND 5
GROUP BY date_part('month',com_date)
ORDER BY date_part('month',com_date) ASC;
```

Il s'agit d'afficher pour chaque mois de l'année 2010 le total commandé de janvier à mai.


```
"GroupAggregate  (cost=428.96..428.99 rows=1 width=16)"
"  ->  Sort  (cost=428.96..428.97 rows=1 width=16)"
"        Sort Key: (date_part('month'::text, com_date))"
"        ->  Seq Scan on commandes  (cost=0.00..428.95 rows=1
width=16)"
"              Filter: ((date_part('month'::text, com_date) >=
1::double precision) AND (date_part('month'::text, com_date) <=
5::double precision) AND (date_part('year'::text, com_date) =
2011::double precision))"
```

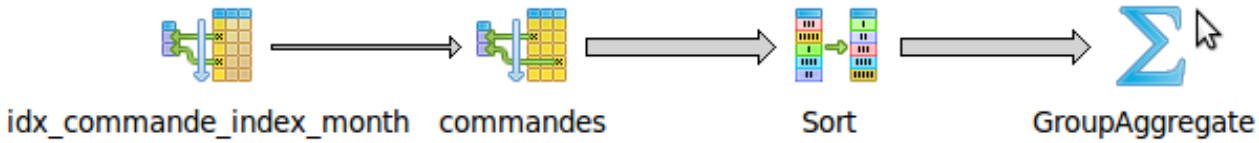
On, imagine que le moteur doit effectuer des calculs à partir de la date pour en ressortir l'année et le mois, pour chaque commande. Si on lui donne un nouvel index sur la colonne date cela ne changera rien.

> ☑ On peut créer un index sur **une fonction appliquée à la colonne** upper(nom), substring(code,1,4), etc!

Nous allons donc créer un index sur le mois, vu que très souvent nous utiliserons l'extraction du mois dans nos requêtes

```
CREATE INDEX idx_commande_index_month
  ON app.commandes
  USING btree
  (date_part('month'::text, com_date));
```

On refait le explain.



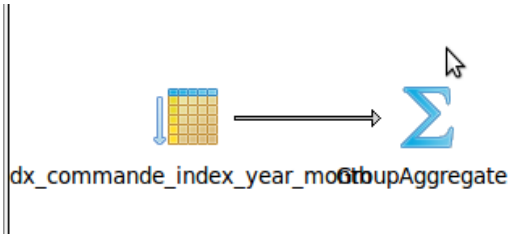
Le explain est déjà meilleur (cost).

```
"GroupAggregate  (cost=116.21..116.23 rows=1 width=16)"
"  ->  Sort  (cost=116.21..116.21 rows=1 width=16)"
"        Sort Key: (date_part('month'::text, com_date))"
"        ->  Bitmap Heap Scan on commandes  (cost=4.76..116.20 rows=1
width=16)"
"              Recheck Cond: ((date_part('month'::text, com_date) >=
1::double precision) AND (date_part('month'::text, com_date) <=
5::double precision))"
"              Filter: (date_part('year'::text, com_date) =
2011::double precision)"
```

```
"              ->  Bitmap Index Scan on idx_commande_index_month
(cost=0.00..4.76 rows=50 width=0)"
"                    Index Cond: ((date_part('month'::text, com_date)
>= 1::double precision) AND (date_part('month'::text, com_date) <=
5::double precision))"
```

Créons maintenant un double index, sur l'année et le mois, car nous savons qu eces requêtes seront toujours filtrées par années, ou bien classées par années, mixer les mois de plusieurs années n'aurait pas de sens fonctionnement (on voit ici que l'indexation est en rapport avec le fonctionnel)

```
CREATE INDEX idx_commande_index_year_month
  ON app.commandes
  USING btree
  (date_part('year'::text, com_date),date_part('month'::text,
com_date));
```



On obtint alors un simple parcours d'index. Notre requête sera certainement plus rapide.

```
"GroupAggregate  (cost=0.01..8.31 rows=1 width=16)"
"  ->  Index Scan using idx_commande_index_year_month on commandes
(cost=0.01..8.28 rows=1 width=16)"
"        Index Cond: ((date_part('year'::text, com_date) =
2011::double precision) AND (date_part('month'::text, com_date) >=
1::double precision) AND (date_part('month'::text, com_date) <=
5::double precision))"
```

17.1.6. Trouver les requêtes à indexer

Ce titre est un abus de langage, ce sont des tables qu'on indexe.

Par contre le point de départ devrait être la requête. L'indexation dépends de l'usage des tables par les applications. Sur des tables importantes les choix d'indexation devraient être remontés dans l'application pour que les requêtes aussi s'adaptent à l'indexation, mais dans l'immense majorité des cas il faudra partir des requêtes réelles.

Le premier outil à disposition est le log des requêtes lentes. Il faut éditer le fichier postgresql.conf et rechercher cette ligne:

```
#log_min_duration_statement = -1
```

et mettre à la place:

```
log_min_duration_statement = 2000
```

Ce qui indique de loger les requêtes qui mettent plus de deux secondes à s'exécuter. Ce sera une bonne piste de départ.

Deux modules complémentaires (contrib) peuvent s'avérer utiles auto_explain http://www.postgresql.org/docs/9.0/interactive/auto-explain.html et pg_stat_statements http://www.postgresql.org/docs/9.0/interactive/pgstatstatements.html.

17.1.7. Contrôler l'usage réel des index

Le catalogue fournit deux **vues** qu'on ne manquera pas de consulter sur une base utilisée en production:

- `pg_stat_user_indexes`: si certains index ne sont jamais utilisés doit-on les garder?
- `pg_stat_user_tables`: si certaines tables on trop de seq_scan il leut manque sans doute des index

18. Éléments Complémentaires

Nous ne pouvons pas couvrir l'ensemble des outils mis à disposition dans une base PostgreSQL. Voici cependant quelques liens vers des objets qui pourraient vous être utiles:

- `LISTEN`/`NOTIFY` : ces instructions permettent la mise en place de programmes persistants en écoute de certains messages (imaginez par exemple un programme externe qui doit effectuer des tâches quand une nouvelle transaction commerciale est enregistrée) http://docs.postgresqlfr.org/9.0/sql-listen.html http://docs.postgresqlfr.org/9.0/sql-notify.html
- **Recherche Plein Texte**: http://docs.postgresqlfr.org/9.0/textsearch.html
- **postgis** : http://www.postgis.org (extension de **S**ystème d'**I**nformation **G**éographique)
- **types composites** : http://docs.postgresqlfr.org/9.0/rowtypes.html
- `DO` : http://docs.postgresqlfr.org/9.0/sql-do.html envie de tester une fonction sans l'enregistrer? Do permet de définir à la volée une fonction pour l'utiliser immédiatement.
- `Create Aggregate` http://docs.postgresqlfr.org/9.0/sql-createaggregate.html
- etc.

19. Questions subsidiaires?

Si vous voulez pousser plus loin ce TP il est à présent temps de se poser des questions sur le travail effectué sur app.

- *Quels sont les index manquants?*
- *Est-ce que les personnels qui ne sont ni employés ni intérimaires sont bien gérés dans app?*
- *La mise à jour du nombre de points en déclencheurs de niveau ligne est-elle la meilleure solution pour contrer les problèmes d'isolation? Est-ce que toutes les problématiques de concurrence de transactions et de LOCKS ont été prises en comptes dans les triggers? Comment pourrais on gérer ce calcul et report de points en déclencheurs de niveau requête et poser les locks ou niveaux d'isolations nécessaires?*

Une bonne lecture en anglais complémentaire du manuel sur les problèmes de locks : http://www.postgresql.org/files/developer/concurrency.pdf

Notez aussi qu'il est impossible de changer le niveau d'isolation d'une transaction au sein d'un délcencheur (il est déjà lui-même integré à une transaction et ne peut pas définir le niveau d'une transaction déjà démarrée). Tout ce qu'il est possible de faire est de mesurer le niveau actuel avec

```
SELECT current_setting('transaction_isolation')
```

Et de par exemple lever une exception si ce niveau ne vous convient pas.

Pour tester les problèmes de cette dernière question voici quelques pistes. Commencez par ajouter des délais d'attente assez longs et variables dans les fonctions de triggers à l'aide de boucles de ce type:

```
PERFORM pg_sleep( ceil(random()*10) );
```

Ce qui peut donner par exemple dans une des fonctions principales (app.total_commande_triggers):

```
CREATE OR REPLACE FUNCTION app.total_commande_triggers()
  RETURNS trigger AS
$BODY$
  DECLARE
    identifiant_commande  integer;
    total_ht              numeric(12,4);
  BEGIN
    RAISE NOTICE E'\ntotal_commande_triggers()\n    Operation: %\n
Schema: %\n    Table: %',
        TG_OP,
        TG_TABLE_SCHEMA,
        TG_TABLE_NAME;
    IF ((TG_OP = 'UPDATE') OR (TG_OP = 'INSERT')) THEN
      identifiant_commande = NEW.com_id;
      total_ht = app.sum_commande(identifiant_commande);
      PERFORM pg_sleep( ceil(random()*10) );
      RAISE NOTICE E'\ncalling app.update_commande_amounts( % ,
% )',total_ht,identifiant_commande;
      PERFORM
app.update_commande_amounts(total_ht,identifiant_commande);
      RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
      identifiant_commande = OLD.com_id;
      total_ht = app.sum_commande(identifiant_commande);
      RAISE NOTICE E'\ncalling app.update_commande_amounts( % ,
% )',total_ht,identifiant_commande;
      PERFORM
app.update_commande_amounts(total_ht,identifiant_commande);
      RETURN NEW;
    END IF;
    RETURN NULL;
  END;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
```

On pourra ensuite utiliser le programme **test_concurrence.php** qui effectue 50 modifications de quantitées sur les lignes de commande de la commande ayant le **com_id** minimal. En utilisant plusieurs terminal (entre 4 et 10 par exemple) vous pouvez faire tourner plusieurs occurrences de **test_concurrence.php.**

Avant de lancer ces programmes lancez ces quelques requêtes pour vérifier l'état des points, puis relancez les après les tests:

```
SELECT co.com_id,co.com_total_ht,sum(lc.lic_total)
FROM app.commandes co
INNER JOIN app.lignes_commande lc ON lc.com_id=co.com_id
WHERE co.com_id = (
SELECT min(com_id) FROM app.commandes
)
GROUP BY co.com_id,co.com_total_ht;
```

Les deux dernières colonnes doivent donner le même montant.

Lançons plusieurs programmes test_concurrence.php en parallèle à l'aide de:

```
php test_concurrence.php
```

En parallèle vous pouvez tester ce que fais le programme en tapant:

```
UPDATE app.lignes_commande SET lic_quantite=lic_quantite+12
 WHERE com_id=3079
 AND lic_id IN (
  SELECT lic_id
  FROM app.lignes_commande
  WHERE com_id=3079
  AND NOT lic_est_reduction
  ORDER BY RANDOM()
  LIMIT 1
);
```

Mais surtout tapez des commandes:

```
SELECT co.com_id,co.com_total_ht,sum(lc.lic_total)
FROM app.commandes co
INNER JOIN app.lignes_commande lc ON lc.com_id=co.com_id
WHERE co.com_id = (
SELECT min(com_id) FROM app.commandes
)
GROUP BY co.com_id,co.com_total_ht;
```

Pendant qu'ils tournent il y a des fortes chances pour que les deux sommes ne soient pas égales. Si vous retapez la requête update alors qu'aucun programme ne tourne ou si l'un des programmes s'arrête longtemps après les autres il y a des chances pour que vous retombiez sur une égalité. Mais au moins vous aurez pu apercevoir un état **incohérent** de la base. Vous pourriez modifier le programme php pour que lui même

- ☑ Essayez de faires les mises à jour avec des opérations atomiques (modifier le total de la commande et calculer ce total dans la même requête)
- ☑ Même en mode sérialisation PostgreSQL ne fais pas de locking prédicitif (deviner toutes les lignes impactées dans vos modifications). Méfiez-vous donc des opérations d'aggrégation (comme le total commande) et placez vous même des locks prédictifs avec des `select for update` sur toutes les lignes commande de la commande (mode revolver) ou des lock de table avec `LOCK TABLE lignes_commande IN SHARE ROW EXCLUSIVE MODE;` (mode lance roquette)
- ☑ En bref sur toutes les opérations critiques et dangereuses essayez de replacer le problème en mode **mono-utilisateur sans concurrence** (lock de table) car c'est souvent le seul mode que votre cerveau arrivera à appréhender sans risques. Si vous avez besoin de meilleures performances en cas de concurrence accrue sur votre application vous devrez peut-être réfléchir et tester des politiques de lock plus fines.

20. Administration PostgreSQL

20.1. Pré-requis

Les chapitres précédents contenaient des informations utiles aux administrateurs. Ainsi on n'oubliera pas de consulter dans les chapitres précédents:

- la gestion du pg_hba.conf
- la gestion des rôles et des droits
- les backups en dumps SQL et COMPRESS ainsi que leurs restaurations
- l'indexation

20.2. 32bits vs 64bits

PostgreSQL existe en version 32 ou 64 bits. Vous pouvez très bien installer une version 32bits sur un OS 64 bits.

Les principaux gains d'une version 64 bits sont:

- une meilleure gestion des types longs, qui peuvent être utilisés dans des registres au lieu de passer par des pointeurs (entiers longs, types date)
- la possibilité d'utiliser plus de 2Go pour le paramètre `shared_buffers` dont on verra qu'il s'agit d'un des paramètres très important pour les performances

Mais des coûts supplémentaires apparaissent aussi en parallèle sur tous les types de base (la taille d'un pointeur en RAM augmente). Sur un Linux 64 bits on devrait toujours installer une version 64bits. Sur un serveur Windows la version 64bits est beaucoup moins intéressante car les serveurs Windows supportent assez mal une valeur supérieure à 500Mo pour `shared_buffers` (on perd donc le principal gain). Certains utilisateurs ont rapporté des installations sur Windows avec des très fortes valeures de `work_mem`, pour lesquelles une version 64bits étaient plus performantes. Mais comme nous le verrons en étudiant ces deux paramètres (`shared_buffers` et `work_mem`) il s'agit là d'installations atypiques

20.3. Analysez l'usage de la base

Les applications qui utilisent la base peuvent avoir des formes et des usages divers. On identifie par exemple certaines grandes familles ainsi:

- Type Web: Taille des données tenant en RAM, beaucoup de requêtes simples, beaucoup de lectures.
- OLTP (Online Transaction Processing): Taille des données très importante (supérieure à la RAM), un nombre important d'opérations d'écritures (plus de 20% des requêtes). Des transactions importantes (beaucoup d'écritures au sein d'une même transaction)
- Data Warehouse, Business Intelligence: taille des données très importante, requêtes d'agrégation complexes (BI), requêtes d'import/export de grandes quantités de données

Les différents conseils sur les performances attendues des données dépendront du profil de la base. Si vos usages sont très différents et que vous hébergez plusieurs bases peut-être devrez-vous songer à utiliser différents serveurs de base de données.

20.4. Autovacuum, vacuum et analyze

http://docs.postgresql.fr/9.0/maintenance.html

http://docs.postgresql.fr/9.0/runtime-config-autovacuum.html

La page de documentation de PostgreSQl sur les opérations de maintenance est assez complète.

Parmi les choses importantes il faut identifier le service autovacuum. Il s'agit d'un des processus fils de PostgreSQL dont le travail est de détecter les maintenance à effectuer et de les faire au fil de l'eau.

Par mi toutes les tâches de maintenance les plus importantes sont donc les `VACUUM`. Le but du vacuum est triple:

- optimiser l'espace disque occupé par la base, le fichier physique stocke plusieurs versions des lignes, ce qui permet d'assurer le MVCC dans les transactions. Lors des `vacuum` les lignes qui ne sont plus valides seront supprimées.
- Mettre à jour les statistiques sur le nombre de lignes de stables ou les cardinalités des index. Ceci afin d'optimiser les choix fais par l'analyseur de requête (vaut-il mieux un seqscan ou utiliser un index?)
- A long terme éviter d'avoir un problème de cycle d'identifiant de transaction (qui n'est pas un nombre infini)

Il y a une forme extrême du `VACUUM` qui est `VACUUM FULL`. Cette commande SQL provoque une sorte de réécriture complète de la table, un réorganisation de toutes ces lignes. C'est une opération longue et bloquante. Depuis la version 9.0 de PostgreSQL cette opération est plus rapide que sur les versions précédentes mais elle impose de disposer de deux fois la taille physique de la table, Si vous avez une table 2Go sur laquelle vous effectuez un `VACUUM FULL` une nouvelle table sera créé, il faut donc environ 2Go d'espace disque disponible. **Cette opération pose un LOCK exclusif** sur la table, elle est inaccessible pour PostgreSQL.

Heureusement il n'est pas nécessaire à priori d'effectuer des `VACUUM FULL`. **Les opérations `VACUUM` standard suffisent à obtenir des maintenances efficaces** et ne sont pas bloquantes, ni en lecture ni en écriture (le DDL est bloqué par contre). On n'utilisera le `VACUUM FULL` que sur un table qui après des imports/exports massifs occupe visiblement une place trop importante. Sur un fonctionnement normal de la base l'exécution régulière de `VACUUM` classiques permet de ne pas avoir de gaspillage de place.

Le but d'autovacuumm est donc de tourner suffisamment souvent pour maintenir la taille physique des tables et pour garantir que le planificateur de requête dispose d'informations à jour. Le démon autovacuum dispose d'indicateurs pour ces deux fonctions qui vont lui permettre de décider du moment ou il devra agir. Parallèlement il ne faut pas que les vacuum se lancent trop souvent car s'ils ne sont pas bloquants ils sont cependant consommateurs en ressource serveur.

Il y a des paramètres généraux qui s'appliquent à toutes les tables mais on peut changer ce paramétrage pour une table (disponible dans les propriétés de la table sur pgadmin):

```
# suspendre l'autovacuum pour une table
ALTER TABLE mytable SET autovacuum_enabled = false;
# le rétablir
ALTER TABLE mytable SET autovacuum_enabled = true;
# Mettre des réglages particuliers pour une table
ALTER TABLE mytable SET (
  autovacuum_vacuum_threshold = 25,
  autovacuum_analyze_threshold = 15,
  autovacuum_vacuum_scale_factor = 0.1,
  autovacuum_analyze_scale_factor = 0.001,
  autovacuum_vacuum_cost_delay = 10,
  autovacuum_vacuum_cost_limit = 100
```

```
);
```

La documentation nous donne ces deux formules:

```
limite du vacuum = limite de base du vacuum + facteur d'échelle du
vacuum * nombre de lignes
limite du analyze = limite de base du analyze + facteur d'échelle du
analyze * nombre de lignes
```

Si on mets les vrais noms de variables et des parenthèses on obtient:

```
limite du vacuum = autovacuum_vacuum_threshold+
(autovacuum_vacuum_scale_factor *nb lignes)
limite du analyze = autovacuum_analyze_thresold +
(autovacuum_analyze_scale_factor *nb lignes)
```

`autovacuum_vacuum_threshold`: Après que ce nombre de lignes mortes dans la table (dues à des **delete** ou **update**) soit atteint un vacuum sera lancé afin de récupérer de l'espace disque

`autovacuum_analyze_threshold`: Après avoir atteint, en gros, ce nombre d'**insert** ou **update** ou **delete** un vacuum analyze sera lancé afin de mettre à jour les statistiques utilisées par l'analyseur de requêtes

Les **`*_scale_factor`** sont des pourcentages appliqués à la taille de la table et vont ajouter des valeurs au « thresold » original. Donc une grosse table (2 millions de lignes) avec un scale factor de 0.1 (10%) va ajouter 200 000 au thresold:

```
limite du vacuum = 50+(0.1 * 2 000 000) = 50 + 200 000 = 200 050
```

Les paramètres `cost` et `delay` servent à éviter de trop impacter le fonctionnement normal de la base en forçant des suspensions/reprises des tâches d'autovacuum en cours de traitement quand celle-ci atteignent les coûts indiqués. Ces réglages ne sont à modifier que si vous observez des ralentissements généraux dus aux vacuums de certains tables importantes.

> ☑ **Comment savoir si on doit changer les réglages par défaut d'une table?** En utilisant les outils de monitoring qui indiqueront les tables qui n'ont pas subit de vacuum depuis très longtemps et en tracant les explain avec les requêtes lentes, afin d'y repérer des mauvaises estimations de coûts par l'analyseur de requête.

Quelques points pour vous aider à trouver les bon réglages

- **les valeurs indiquées pour analyse devraient être supérieures à celles de vacuum** car elles comptent aussi les insertions (si vous avez des insertions sur cette table)
- Pour une table statique (données de paramétrage par exemple), où les données bougent très rarement, ne vous occupez pas des problèmes de vacuum
- Une table qui ne subit **que des insertions** n'aura **jamais de vacuum** (mais n'a pas besoin de vacuum, si vous avez uniquement des insertions et un fillfactor à 100 la table ne va pas gaspiller d'espace disque)

1) Sur une table qui ne subit que des insertions ne vous occupez que des paramètres `analyze`.
1) faire tourner de petits `vacuum` fréquemment est moins coûteux que de faire tourner un gros `vacuum`
2) Pour les grosses tables (en nombre de lignes) qui subissent un grand nombre d'insertions vous aurez besoin de faire tourner `analyze` fréquemment, réduisez le `autovacuum_analyze_scale_factor`

3) Pour les grosses tables (en nombre de lignes) avec beaucoup de mouvements (`delete`, `update`, `insert`) vous devriez réduire le `autovacuum_vacuum_scale_factor` pour avoir plus souvent des opérations de vacuum. Vous pouvez aussi décider de monter le `autovacuum_vacuum_thresold` à un chiffre élevé (comme 1000 ou 5000) et mettre 0 au `autovacuum_vacuum_scale_factor`.

4) Les réglages par défaut seront très bien (au sens ou vous n'avez pas besoin de les modifier) si vous ne subissez pas de ralentissements sur vos requêtes et si votre espace disque occupé est raisonnable

5) Sur une grosse table avec beaucoup de mises à jour (chose assez peu fréquente en fait) un moyen simple d'obtenir un `vacuum` plus prévisible est de diminuer le factor et de mettre dans le thresold une valeure moyenne du nombre d'opérations par jour.

6) Pour une table qui reçoit des **mises à jour en mode batch**, par exemple elle reçoit 10 000 nouvelles lignes chaque nuit graĉe à un cron; mettez le thresold d'analyze à 10000 et le factor à 0, ainsi seul le nombre d'insertions va déterminer le lancement du vacuum. S'il ne s'agit pas que d'insertions mais aussi de mises à jour et de suppressions modifiez le thresold et le factor du vacuum de la même façon.

20.5. Paramètres de configuration principaux

Les paramètres de postgreSQL se trouvent dans le fichier postgresql.conf. Ce fichier est disponible dans les répertoires /etc/postgresql/* sur les distributions type debian, mais il s'agit en fait, comme le pg_hba.conf que nous avons vu dans les parties précédentes, d'un simple raccourci vers le répertoire de stockage physique de la base

La lecture complète de ce fichier et des commentaires qui s'y trouve vous apportera toujours une base de connaissance utile (les commentaires indiquent par exemple les paramètres qui nécessitent un redémarrage complet du serveur pour être pris en compte, dans le cas contraire un simple reload suffira) . Faisons le point sur les principaux paramètres:

20.5.1. Connexions

- **listen_addresses** : liste des interfaces réseau sur lesquelles le serveur est à l'écoute. Par défaut `'localhost'` et onc uniquement en local, mettez `'*'` pour utiliser toutes les adresses réseaux du serveur.
- **port** : le port sur lequel le serveur est en écoute sur les interfaces listées dans le paramètre précédent. Modifiez le si plusieurs instances de postgreSQL doivent tourner en parallèle (comme lors d'un upgrade)
- **max_connections** : nombre maximum de connexions acceptées, le défaut est à 100 ce qui est très peu. Ajoutez un zéro et passez à **1000**. Pensez par exemple que deux serveurs frontaux apache avec un MaxClients à 150 demanderont 300 connexions en pic (s'ils ne servent qu'une seule application, avec un seul login sur une seule base...). Si vous utilisez Apache en mode multi-threadé (worker) vous risquez d'avoir un `MaxClients` beaucoup plus élevé côté apache, si vous utilisez plusieurs rôles cela va aussi augmenter la consommation de connexions. Pensez à utiliser des **pooler de connexions** pour des besoins dépassants le milliers de connexions.

  ☑ Sur **Windows** réduisez ce nombre, peut-être à **50** par exemple et utilisez un pooler, pour réduire la taille du segment de mémoire partgée.

- **superuser_reserved_connections** : Parmi toutes les connexions disponible ce nombre de connexions (3 par défaut) sera réservé au superadmin postgres. Cela vous permettra de vous connecter à postgreSQL y compris au moment des pics. Intégrez dans ce nombre la consommation des membres de l'équipe **d'admin** et des logiciels de **supervision**.

20.5.2. Mémoire

La mémoire est utilisé par PostgreSQL de deux façons, une partie de cette mémoire est consommée par chaque connexion ouverte. Une seconde partie beaucoup plus importante est utilisée en tant que mémoire partagée par toutes les connexions. Chaque connexion est un processus différent (fork) et la principale particularité de PostgreSQL est l'utilisation de cette mémoire partagée.

> ☑ Pour information un autre logiciel faisant une utilisation intensive de la mémoire partagée est **Varnish**

Les paramètres les plus importants sont donc ceux de la mémoire partagée.

> ☑ Sur la plupart des distributions **les valeurs par défaut pour la mémoire sont très faibles**, adaptées à une utilisation légère de PostgreSQL par une seul utilisateur. Ceci parce que la plupart des distributions n'autorisent par défaut que des valeurs très petites pour la taille d'un fichier de mémoire partagée. Vous aurez donc très certainement besoin de modifier ces paramètres et de modifier la taille limite des fichiers partagés dans le système.

La mémoire partagée est utilisée pour cacher les données du disque, ce qui comprends les données des journaux de transactions et les fichiers des tables et autres objets physiques (le contenu du répertoire de données de la base en fait).

- **wal_buffers**: taille réservé au cache des journaux transactions: 64KB par défaut: sur un serveur ou des transactions sont réellement en œuvre (pas uniquement des opérations en lecture) utiliser des valeurs entre 1Mo et 10Mo.
- **shared_buffers**: 28MB par défaut (je crois) taille réservée au cache des données : le plus est le mieux, plus vous pourrez mapper de vos données physiques dans ce cache mieux le serveur se portera. Le problème étant que vous devez disposer de cette mémoire (sinon le serveur va swapper, ce qui serait pire). On conseille souvent d'utiliser au départ un quart de la mémoire du serveur (donc 500MB sur un serveur qui dispose de 2Go de RAM). En mode 32bit la limite est de 2GB. Surveillez la statistique `cache_miss` pour voir si votre paramètre est trop petit.

> ☑ **ATTENTION**: sur **Windows** ne jamais utiliser plus de **512MB** en **shared_buffers**. Les performances s'effondrent une fois ce seuil dépassé. Il faudra jouer sur d'autres paramètres.

Quand vous essayerez de démarrer ou redémarrez postgreSQL avec des nouvelles valeurs dans ces champs il est très porbable que celui-ci refuse de se lancer à cause d'une taille de fichier partagé trop importante. Regardez le fichier de log, celui-ci indique la valeur que le serveur à tenté d'allouer:

```
> /etc/init.d/postgresql-9.0 restart
Restarting PostgreSQL 9.0:
waiting for server to shut down.... done
server stopped
waiting for server to
start.....................................................................pg
_ctl: could not start server
Examine the log output.
PostgreSQL 9.0 did not start in a timely fashion, please see
/path/to/data/pg_log/startup.log for details
```

```
> tail -f -n 10 /path/to/data/pg_log/startup.log
(...)
2011-10-26 18:06:29 CEST FATAL:  could not create shared memory
segment: Argument invalide
2011-10-26 18:06:29 CEST DETAIL:  Failed system call was
shmget(key=5439001, size=538116096, 03600).
2011-10-26 18:06:29 This error usually means that PostgreSQL's request
for a shared memory segment exceeded your kernel's SHMMAX parameter.
You can either reduce the request size or reconfigure the kernel with
larger SHMMAX.  To reduce the request size (currently 554467328
bytes), reduce PostgreSQL's shared_buffers parameter (currently 64000)
and/or its max_connections parameter (currently 1004).
 If the request size is already small, it's possible that it is less
than your kernel's SHMMIN parameter, in which case raising the request
size or reconfiguring SHMMIN is called for.
 The PostgreSQL documentation contains more information about shared
memory configuration.
```

Pour modifier les paramètre SHMIN et SHMMAX on utilise sysctl ainsi:

```
>#  voir les paramètres actuels
> sysctl -a |grep -i shm
kernel.shmmax = 33554432
kernel.shmall = 2097152
kernel.shmmni = 4096
```

On mets les nouvelles valeurs dans /etc/sysctl.conf:

```
kernel.shmmax = 600000000
kernel.shmall = 600000000
```

Puis on recharge ces valeurs:

```
> sysctl -f /etc/sysctl.conf
```

☑ Comme tous les serveurs de base de données PostgreSQL adore la RAM, pensez à le séparer des autres processus dévoreurs de RAM comme Apache ou un serveur J2EE. Donnez lui un machine dédiée, avec aussi des disques rapides et sûrs.

Pour la consommation par processus le paramètre important est :

– **work_mem** : cela représente la mémoire que le processus à le droit d'utiliser pour effectuer ses opérations de hachage et de tris (celles que le explain montre). S'il a besoin de plus de mémoire il devra passer par un stockage temporaire sur disque des opérations en cours. La difficulté de ce paramètre tient au fait que non seulement il est potentiellement à multiplier par le nombre de connexions parallèles (max_connections au pire) mais qu'en plus un même processus à le droit de l'utiliser plusieurs fois si la requête qu'il exécute comprends plusieurs opérations de hachage (certains `explain` nous on montré des plans d'exécution complexes où cette `work_mem` aurait été utilisée plusieurs fois pour la même requête). La valeur par défaut est `1MB`. Vous pouvez essayer de monter à **10MB** mais pour 1000 connexions parallèles qui auraient des requêtes ardues à effectuer cela veut dire potentiellement 10Go de RAM – il est improbable que toutes les connexions utilisent en même temps la maximum de ce qui leur est autorisé). Si vous mettez une valeur trop basse vous allez augmenter l'utilisation des fichiers temporaires ce qui ralentira le temps d'exécution des requêtes et augmentera l'activité sur disque (qui est lente).

☑ Vous pouvez affecter cette valeur à l'aide des variables utilisateur, par exemple au niveau des rôles. Ainsi certains rôles effectuant du travail sur des quantités importantes de données pourront avoir un `work_mem` par défaut plus important.

- **maintenance_work_mem** : 16MB par défaut, montez à 100MB voir plus. Il s'agit de la mémoire allouée aux processus du superutilisateur effectuant des opérations de maintenance comme les vaccuums ou les réindexations, les clusters etc. Il n'y a normalement pas de parallélisations de ces tâches

  ☑ **Lors d'un import de données massif**, il n'y aura à priori que des connexions destinées à cet import (si vous coupez les autres via le pg_hba.conf par exemple), pensez à augmenter les valeurs de `work_mem` et `maintenance_work_mem` temporairement pour accélérer l'import.

- **effective_io_concurrency**: indiquez le nombre de disque présents sur le système

Signalons enfin d'autre paramètres proches de l'utilisation mémoire mais qui sont plus des réglages informatifs:

- **effective_cache_size** : 128MB par défaut il ne s'agit pas d'un paramètre de consommation de mémoire par PostgreSQL. Il s'agit d'un paramètre d'information sur le système d'exploitation, la taille de RAM libre disponible pour effectuer des requêtes. Sur Linux mettez environ 2/3 de la RAM, sur Windows regardez la valeur du cache disque sur le gestionnaire de tâches. Il s'agit ici d'indiquer à PostgreSQL la taille du cache disque de l'OS pour que le planificateur de requêtes calcule la probabilité qu'une table et/ou son index soient dans le cache disque de l'OS.
- **random_page_cost** : il s'agit d'un indicateur de coût (unité arbitraire) pour accéder à une page de donnée sur le disque. La valeur par défaut est 4.0. Si vous pensez que votre machine dispose de disques qui valent mieux que la moyenne du marché baissez ce coût. Par exemple à 3.0 ou 2.0. Dans l'idéal faites des benchmarks pour mesurer les gains éventuels sur des requêtes.

Ces deux paramètres peuvent jouer en faveur des parcours d'index au lieu de parcours séquentiels lors de l'exécution des requêtes.

20.5.3. Les logs

Il y a de nombreux paramètres liés aux journaux dans le fichier postgresql.conf. On peut par exemple décider de loger au format CSV, de tracer toutes les requêtes, ou bien aucune, ou bien seulement celles qui modifient la structure de la base (ddl). On peut associer un explain avec les requêtes tracées, garder une trace du temps d'exécution, des connections, etc.

Les paramètres les plus importants pour les logs sont:

- `log_min_duration_statement` : indiquez une valeur au dessus de laquelle vous garderez une trace de la requête, cela vous permettra d'identifier les requêtes qui nécessitent un travail de réécriture ou d'indexation.
- `log_temp_files` : indiquez une taille, si une requête nécessite la création d'un fichier temporaire supérieur à cette taille elle sera loguée.
- `lc_messages = 'C'` : contrairement aux autres paramètres de locales (comme les monnaies, ordre de tris, heure) vous devriez laisser les messages dans la locale par défaut ©. Ceci vous permettra de **retrouver plus vite de l'aide sur Internet** en recopiant les messages d'erreur retrouvés dans les logs.

Si vous voulez tester les logs en CSV vous devrez paramétrer vos logs ainsi:

```
# eventlog & stderr ne sont pas requis pour les logs csv
# mais on voit qu'il s'agit d'une liste et pas d'un réglage
# unique
# windows
#log_destination = 'csvlog,eventlog,stderr'
# linux
log_destination = 'csvlog,syslog,stderr'
logging_collector = on
# Nous changeons le nom de fichier pour par exemple ne garder
# que le nom du jour (lundi)
log_filename = 'postgresql-%a.log'
# Après une semaine le fichier 'lundi' sera réutilisé, il devra
# avoir été remis à vide
log_truncate_on_rotation = on
# Ici si vous essayez de forcer la rotation des logs sur l'age
# ou la taille cela ne devrait plus fonctionner. Sans doute
# parce que l'heure ne figure plus dans log_filename
log_rotation_age = 1440
log_rotation_size = 100kB
```

Si vous voulez importer vos logs CSV dans une table PostgreSQL vous aurez besoin de cette table:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

Ensuite importez le log dans la table:

```
COPY postgres_log FROM '/chemin/complet/vers/logfile.csv' WITH csv;
```

Attendez que le fichier de log ne soit plus utilisé pour l'importer (sinon vous aurez des problèmes avec la clef primaire en essayant de le réimporter).

Si vous essayez de l'importer depuis une application faites attention au fait que certains des champs peuvent contenir des retours chariots et donc être sur plusieurs lignes (comme `internal_query` et `message`).

> ☑ Certaines traces ne peuvent être poussées dans le csvlog ou l'eventlog, comme les logs en provenance des erreurs de linkage des librairies partagées (python, perl) ou des corruptions de mémoire. Vous aurez donc toujours un fichier de log classique (*.log), qui sera le plus souvent vide, mais n'oubliez pas de le regarder. Le jour où ce fichier ne sera pas vide ce sera pour des problèmes critiques « on ne réalise pas qu'on en a besoin jusqu'à ce qu'on en ai besoin, et ce jour là on en a **vraiment** besoin ».

20.5.4. Les journaux de transactions (WAL) et CHECKPOINT

Quand des écritures on lieu dans postgresql il y a toujours une transaction. Chacune de ces transactions est tracée dans le WAL (**Write Ahead Logging**). Nous pouvons d'ailleurs observer que parmis les premiers processus de postgresql l'un d'entre eux est dédié à cette opération:

```
>ps auxf|grep postgres
postgres  S      0:00 /path/to/bin/postgres -D /path/to/data
postgres  Ss     0:00  \_ postgres: logger process
postgres  Ss     0:00  \_ postgres: writer process
postgres  Ss     0:00  \_ postgres: wal writer process
postgres  Ss     0:00  \_ postgres: autovacuum launcher process
postgres  Ss     0:00  \_ postgres: stats collector process
```

Le WAL enregistre toutes les transactions validées. Sans pour autant que ces opérations soient réellement transférées sur le disque au niveau des tables. Cela permet la reprise d'un état cohérent de la base en cas d'arrêt brutal, sans pour autant ralentir les opérations d'écritures trop fortement en forçant les fichiers binaires des tables à être raccord avec l'état réel des données en permanence.

Le WAL est constitué de fichiers. Ces fichiers contiennent des copies des pages mémoire des tables et des informations de modification à effectuer. Quand un fichier WAL est rempli un nouveau fichier WAL est créé. Ces fichiers font 16MB. Ils sont stockés dans le répertoire `pg_xlog` du dossier data de la base.

> ☑ Les fichiers WAL (journaux) sont stockés dans le dossier `pg_xlog` du répertoire des données. Il peut s'avérer très utile d'**utiliser un disque différent** sur ce point de montage du système de fichier (parallélisation, gestion du disque cache, taux d'IO du disque). Sur Windows voir les systèmes de JUNCTION.

Notez que même une base sans aucune activité en écriture aura des fichiers WAL générés dans ce dossier. Ceci parce qu'au moins une opération d'écriture arrive régulièrement, le `CHECKPOINT` et que cette opération est elle-même enregistrée dans un fichier WAL.

Les checkpoints peuvent se produire à plusieurs moments:

- `checkpoint_timeout` minutes (par défaut 5)se sont passées depuis le dernier checkpoints
- il y a eu plus de `checkpoint_segments` fichier WAL créés (par défaut 3)
- quelqu'un à lancé une commande SQL `CHECKPOINT;`

Lors du `checkpoint` les changements stockés dans les fichiers WAL **sont écrits dans les fichiers physiques des tables**.

On peut alors imaginer que l'opération de `CHECKPOINT` est une opération coûteuse pour l'OS, toutes les écritures sont reportés à un moment ultime, quand ce moment intervient un grand nombre d'écritures doivent se faire. Heureusement on peut répartir ces écritures entre deux checkpoints grâce au paramètre `checkpoint_completion_target`. La valeur par défaut 0.5 signifie que PostgreSQL dispose de `0.5*checkpoint_timeout` soit 2 minutes 30 par défaut pour effectuer les écritures réelles. En le fixant à `0.9` on permet un lissage plus fort encore de ces écritures. Mais vous pouvez aussi repousser `checkpoint_timeout` et utiliser une valeur assez basse pour `checkpoint_completion_target`.

Remarquez le paramètre `checkpoint_warning` à 30s par défaut. Si plus de 3 WAL sont créés en moins de 30s un nouveau `CHECKPOINT` très rapproché de l'ancien sera généré et vous aurez une ligne de warning dans les logs, si vous voyez un grand nombre de ces warnings lors d'une activité régulière de la base cela signifiera que vous devrez augmenter votre valeur de `checkpoint_segments` qui est à 3 par défaut.

20.6. Considérations matérielles pour la performance

- Plus vous aurez de **CPU** (nombre) plus vous pourrez traiter de requêtes en parallèle. Et cela pourra aussi devenir crucial lors de l'utilisation de `pg_restore` pour paralléliser les traitements d'import.
- Plus vous aurez de **RAM** plus vous pourrez espérer faire tenir l'intégralité des données de la base dans le `shared_buffers`, ou plus vous aurez la possibilité d'augmenter `work_mem` afin d'éviter l'utilisation de tables temporaires sur le disque lors d'un travail d'une requête sur un grand nombre de données.
- Si vous n'avez pas assez de **RAM** et que vous avez réglé des paramètres d'utilisation de la RAM trop élevés les performances s'effondreront. Attention aux programmes tournant sur le même serveur.
- Certains type de **RAM** sont plus sûrs que d'autres (comme la RAM ECC)
- les **Entrées-Sorties disques** seront importante si vous n'arrivez pas à faire tenir la base dans shared_buffers. Soit parce que votre base est très grande, soit parce que vous travaillez avec Windows. Si vous activez le **WAL** pour les backups ou la réplication les opérations d'écritures vont aussi impliquer des Entrées-Sortie disque, peut-être devrez-vous prévoir des disques très rapides, supportant un grand nombre d'Entrées/Sorties, spécifiquement pour le WAL.
- Si vos disques sont en **RAID** http://fr.wikipedia.org/wiki/RAID_%28informatique%29 notez qu'un **RAID5** peut ralentir les opérations d'écritures, le meilleur système de RAID est le **RAID1+0** (mirroring et agrégation), mais il est assez coûteux en nombre de disques. Pour des volumes vraiment très gros vous devrez étudier les différents système de SAN à disposition, mais évitez les système de disque réseau type NFS ou ISCSI si vous pouvez utiliser des vrais disques avec du RAID matériel.
- Multipliez les cartes **réseau** et ne mélangez pas les différents flux réseaux. Si vous avez des flux de réplication ils ne devraient pas passer par les mêmes cartes réseau que les flux à destination des serveurs d'application. Vous pourrez de plus mieux superviser les évolutions de trafic des différents flux. Utiliser un réseau d'administration peut aussi permettre de régler des politiques d'accès (pg_hba.conf) ou d'éviter d'engorger le trafic réseau au moment des sauvegardes.
- **Monitorez** et mesurez les impacts des changements de matériel et de configuration

20.7. Backup et Restaurations liés à l'archivage WAL

Nous verrons par la suite que le WAL peut servir à des politiques de réplication. Dans un premier temps nous analyserons sa première utilité qui est de permettre un backup par sauvegarde des journaux de transactions (associé à

un backup de l'état physique de la base à un instant couvert par ces journaux). Ce type de backup est très puissant puisque contrairement aux dumps il permet:

- la sauvegarde des modifications au fil de l'eau
- le **PITR** (Point In Time Recovery), la restauration à un état passé de la base.

20.7.1. Configurer l'archivage des WAL

Les fichiers WAL que nous avons étudié avec les principaux paramètres de configuration permettent de rejouer toutes les transactions qui ne sont pas encore écrites sur les fichiers physiques. Cela signifie qu'en partant d'une version ancienne des fichiers physiques de la base et en rejouant tous les fichiers WAL créés depuis on peut ré-obtenir une version récente de la base (et on peut s'arrêter à une transaction donnée).

Ce système de backup existe et s'appelle le **WAL Archiving**.

Ceci se fait en indiquant '**archive**' au lieu de '**minimal**' à l'option `wal_level` et en activant l'`archive_mode`, et l'`archive_command`. Il faut ensuite configurer quelques paramètres comme indiqué dans cet extrait de configuration commenté:

```
# Activation de l'archivage WAL
wal_level = archive

# Ceci va activer la commande d'archivage
archive_mode = on

# Ceci est la commande utilisée par PostgreSQL pour archiver les logs
# %p : chemin du fichier à archiver (le journal de transaction
original)
# %f : le nom du fichier sans le chemin
# la commande ne doit retourner 0 en code sortie
# qu'en cas de succès!!!
# Il s'agit ici d'un script de backup incrémental
# Quand le ficher WAL est plein ou trop vieux il sera archivé
# à l'aide de cette commande et un nouveau WAL sera utilisé par
# le serveur
# En cas de succès de l'archivage le fichier WAL peut être
# supprimé ou réutilisé par postgreSQL
# Ici nous pourrions aussi utiliser un script qui au passage ferait
# une compression une utiliser un pipe pour cela (|)
archive_command = 'test ! -f /mnt/serveur/archive/%f && cp -i %p
/mnt/serveur/archive/%f </dev/null'

# Ceci va forcer l'archivage d'un journal de transaction (WAL)
# même si'il n'y a pas eu beaucoup de modifications.
# Donc en cas de période d'inactivité sur la base nous aurons ic
# le temps le plus long avant qu'une modification ne soit
# réèllement archivée (300s->5min)
archive_timeout = 300s

# En utilisant la valeur par défaut (on) on indique que les commit
# ne sont considérés réèl qu'après que le WAL soit écrit sur le disque
synchronous_commit = on

# Ceci est la méthode qui est utilisée pour s'assurer que le fichier
# WAL est synchronisé sur le disque
#"fsync" ou "fsync_writethrough" : Force écriture réèlle sur le disque
```

```
# sans que l'OS puisse utiliser son cache disque
# "open_datasync" : valeur par dfaut pour windows mais cela implique
# un cache disque
# "fsync" apelle fsync() à chaque commit,"fsync_writethrough" aussi
# en forcant les « write-through »des caches internes aux disques
wal_sync_method = 'fsync_writethrough'

# checkpoint_completion_target indique que PostgreSQL peut utiliser
# ce % de checkpoint intervall time (5min par défaut)
# pour effectuer les vraies E/S disque lors des checkpoints
# ici on utilise seulement 0.3, car 0.3*15min=4min30s
checkpoint_completion_target = 0.3

# la valeur par défaut est 5min,
# on pourrait la garder mais cela provoque une modif dans le WAL
# toutes les 5 minutes (à cause du checkpoint enregistré dans le WAL)
# Donc comme un fichier WAL ne peut être plus vieuw que 5 minutes
# cela génèrerait un fichier WAL toutes les 5 minutes au moins.
# Nous le poussaons à 15 minutes.
# Cela signifie que les écritures de pages ne se feront sur le disque
# que toutes les 15 minutes si l'activité est faible
# Mais les fichiers WAL sont écrits sur disque à chaque fois eux,
# ce n'est pas trop grave.
# Note: 15 minutes est uniquement un maximum,
# si 3 (checkpoint_segments) sont près un checkpoint sera effectué
checkpoint_timeout = 15min
checkpoint_segments = 3
```

☑ Le point important est `archive_command`. Le but de cette commande d'archivage est de recopier les fichiers WAL stockés dans le dossier pg_xlog vers un endroit sûr. La commande peut être un script complexe ou une simple ligne de commande, **le résultat est pour PostgreSQL la certitude que ce WAL a été recopié à un endroit où il ne craint plus un arrêt brutal du serveur.**

Examinons la paramètres de temps sur la génération de segments et les checkpoints. Avec un checkpoint_timeout de 15 minutes et une base inactive on obtient sur une frise chronologique:

```
C: Checkpoint
W: création fichier WAL
WA: archivage fichier WAL

|C---------------C--------------------C--------------------C--------
 0               15                   30                   45
|W----WA---------W------WA------------W--------WA----------W-----WA
 0    5          15     20            30        35          45    50
```

Un fichier WAL est créé toutes les 15 minutes et il est archivé cinq minutes plus tard par la commande `archive_command`. Bien sûr en cas d'activité en écriture sur la base des fichiers WAL peuvent être créés beaucoup plus vite et seront archivés quand ils auront plus de cinq minutes d'âge.

Le rôle de la commande d'archivage est de faire une copie de ces fichiers sur un autre système. Vous pouvez utiliser rsync, ou une simple copie sur un disque réseau, voir sur un disque dédié à cette tâche.

☑ Les fichiers WAL font 16MB, avec une base inactive et un WAL tous les quarts d'heures cela représente **96*16=1.5Go** de donnes minimales à archiver **par jour**. N'hésitez donc pas à compresser ces fichiers lors de l'utilisation de l'`archive_command`, vous pourrez gagner plus de **90%** d'espace disque sur le backup. Surtout sur les WAL créés en période inactive. Un programme nommé pg_compresslog peut être utilisé à cette fin ainsi: `archive_command = 'pg_compresslog %p - | gzip > /var/lib/pgsql/archive/%f'` ce qui nous donnera pour la commande de restauration `restore_command = 'gunzip < /mnt/server/archivedir/%f | pg_decompresslog - %p'`.

➢ *indiquez ces paramètres dans postgresql.conf, créez un dossier (`mkdir -p /mnt/serveur/archive; chown -R postgres /mnt/serveur;`) qui ne sera pas déporté mais qui pourrait l'être. Redémarrez PostgreSQL puis testez l'effet du programme de génération de commandes (populate_app.php) sur les fichiers présents dans pg_xlog et dans votre répertoire d'archivage (`/mnt/serveur/archive`).*

20.7.2. Et sur Windows?

Sur une machine Windows on pourrait utiliser pour l'archive_command une commande de ce type:

```
archive_command = 'copy %p E:\\ExternalBackup\\pg_xlog\\%f'
```

Un des problèmes par contre sur windows est le **`wal_sync_method`** qui est à `'open_datasync'` par défaut, avec comme indiqué dans le commentaire de ce paramètre le problème d'utilisation du cache disque par l'OS. On peut utiliser `'fsync_writethrough'` **ou tester** `'open_datasync'` mais il faut alors empêcher le cache disque de windows en allant dans:

```
# My Computer\Open\disk
drive\Properties\Hardware\Properties\Policies\Enable write caching on
the disk
```

20.7.3. Automatiser une sauvegarde WAL

Les opérations qui vont devoir être effectuées lors d'un backup sont:

- **1)** lancer un `SELECT pg_start_backup('iciunechaînedecaractères');`. Cette commande va créer un fichier dans le répertoire des données qui identifiera le backup en cours. Elle lance aussi un `CHECKPOINT` qui force l'écriture des données sur le disque. Si vous passez l'option `true` en deuxième argument ce `CHECKPOINT` sera forcé, sinon vous devrez attendre la fin du `CHEKPOINT` qui peut dépendre du paramètre `checkpoint_completion_target` que vous avez donné. La requête retournera un résultat quand le checkpoint se terminera.
  - Si cette commande renvoie des erreurs vous devriez sans doute arrêter le backup (un backup précédent qui ne s'est pas terminé?)
- **2) Faire une copie de tout le contenu du répertoire des données**
  - **Il n'est pas nécessaire de copier le sous-répertoire pg_xlog**. Icelui-ci est normalement déjà pris en charge par le système d'archivage des WAL.
  - Une des techniques de sauvegarde du répertoire des données et d'utiliser un système de fichier capable de faire des **snapshots** en faisant des freeze des fichiers et en gérant les modifications sur des système temporaires (comme XFS par exemple). Si vous pouvez faire un freeze du système de

fichier juste après le `pg_start_backup` les opérations de restauration seront simplifiées car aucun fichier ne contiendra de données datant de transactions ultérieures au début du backup.

- **3)** effectuer un `SELECT pg_stop_backup();` ceci arrête le backup en passant au prochain WAL et retire le fichier backup_label du répertoire des données (que l'on aura donc copié avec la sauvegarde, mais ce n'est pas grave). Si nous sommes en mode archive_mode (backup des journaux de transactions) cette commande va attendre jusqu'à ce que le dernier segment WAL soit considéré comme archivé (celui qui contenait les dernières opérations effectuées en live sur la base pendant que nous faisions la copie binaire du répertoire des données entre `pg_start_backup` et `pg_stop_backup`).
  - Si jamais l'archive_command ne retourne pas 0 pour la sauvegarde du dernier WAL `pg_stop_backup()` ne rendra pas la main. Votre script pourra donc inclure une gestion du timeout. Votre système de supervision devrait aussi vérifier que les backups se terminent (pour **Nagios** voir les services **passifs** et le paramètre `freshness`, fraîcheur).

Pour lancer les commandes SELECT en début et fin de backup on pourra utiliser `psql` avec l'option `-c`. Ce qui donne par exemple dans un script BAT (windows):

```
%BINDIR%\psql.exe -h %SERVER% -U %USERNAME% -d %DATABASE% -p %PORT%
--no-password --echo-all -c "SET lc_messages=\"en_US\"; SELECT
pg_start_backup(E'%BACKUPLABEL%');"
IF ERRORLEVEL 1 goto (...)
```

Ou dans un script Bash:

```
${BINDIR}\psql -h ${SERVER} -U${USERNAME} -d ${DATABASE} -p ${PORT}
--no-password --echo-all -c "SET lc_messages=\"en_US\"; SELECT
pg_start_backup(E'%BACKUPLABEL%');"
if [ $? ne 0 ]
    then (...)
```

Nous pourrions utiliser rsync, copy, ou un simple tar.

➢ *Testons un script simpliste (sans gestion d'erreur). Nommons ce script backup.sh, il faut adapter dans cescript le dossier DATADIR par rapport à votre vrai dossier de stockage de la base.*

```
#!/bin/bash
DATADIR=/var/lib/pgsql/data
ARCHIVEDIR=/mnt/serveur/archive
BACKUPDIR=/mnt/serveur/backup
# demarrage du backup
psql --username=postgres -d postgres -c "select
pg_start_backup('hot_backup');"
# backup binaire
tar -cf ${BACKUPDIR}/backup.tar ${DATADIR}
# fin du backup
psql --username=postgres -d postgres -c "select pg_stop_backup();"
```

Il nous faut rendre ce script exécutable et prévoir le dossier de stockage du backup:

```
chmod u+x backup.sh
sudo mkdir /mnt/serveur/backup/
```

➢ *Lancez le script. Vous pouvez normalement faire tourner le script php populate_app.php en parallèle du backup.*

```
sudo ./backup.sh
```

20.7.4. Recovery: Restaurer un archivage de WAL

Maintenant que nous avons au moins une copie de la base et un archivage des journaux de transaction nous devrions pouvoir tester une restauration.

Quelques notes utiles sur les restaurations:

- les segments de WAL qui ne seront pas retrouvés à l'emplacement d'archivage seront recherchés dans le dossier pg_xlog de la base s'il existe encore (nous somme en procédure de recovery, si ça se trouve on a plus ce dossier). Par contre les segments présents dans le dossier d'archivage seront prioritaires.
- Avec une restauration on peut voir la gestion du temps dans PostgreSQL comme une gestion parallèle du temps. Un monde parallèle dans lequel les transactions de la restauration et les transactions éventuellement présente dans des WAL locaux ne seront pas mélangés.
- Normalement une restauration va reprendre tous les WAL qu'elle a à disposition, et donc ramener la base à un point dans le temps qui est le plus proche possible du présent. Normalement une restauration se termine avec un message dans les logs signalant un équivalent de « **file not found** », rien d'alarmant. Il peut aussi y avoir un message d'erreur en début de restauration sur un fichier 00000001.history, ce n'est pas non plus un vrai problème.
- Il est possible d'écrire des fichiers `recovery.conf` avancés et de les stocker dans le répertoire des données de la base avant la restauration. Ceci permet un le Point in Time Recovery (**PITR**) qui permettra de s'arrêter un un temps ou un numéro de transaction donné. Ce point dans le temps doit être situé hors du temps du backup. Il ne peut être situé entre le temps du `pg_start_backup()` et du `pg_stop_backup()`. Des exemples de `recovery.conf` sont disponibles et commentés dans `recovery.conf.sample` qui est livré avec le package ou les sources (dossier share).
- La commande miroir de `archive_command` est `restore_command`. Elle doit permettre de récupérer les segments archivés. Comme la première cette commande doit renvoyer un code de sortie autre que 0 en cas d'erreur. Cette commande devra figurer dans un fichier `recovery.conf` situé dans le répertoire de la base.

Nous allons donc nous créer une situation de crash, un fichier de configuration spécifiquement dédié à la restauration (optionnel), un fichier de recovery (obligatoire) puis tenter cette restauration.

20.7.5. Fichier de configuration dédié à la restauration

Au moment d'une restauration vous allez avoir de nombreux journaux de transaction à rejouer, ou bien un dump important à intégrer. Il sera très certainement utile de préparer à l'avance un fichier de configuration optimisé pour les restaurations. Stockez ce fichier à côté du fichier postgresql.conf officiel et utilisez-le le jour J.

```
cp postgresql.conf postgresql.restore.conf
cp postgresql.conf postgresql.orig.conf
```

> ☑ Vous pouvez aussi utiliser ces réglages sur les machines de **développement** si la stabilité de la base n'est pas une priorité par rapport au temps de réponse puisqu'il s'agit d'un fichier optimisé en vitesse d'exécution et non en intégrité des données physiques.

Éditez donc cette copie `postgresql.restore.conf` puis changez ces paramètres:

```
# Ne pas tout mettre dans les fichiers WAL
# par exemple exclure les instructions COPY du dump
wal_level = minimal
# va désactiver l'archivage des WAL, ils ne sont donc plus archivés.
```

```
archive_mode = off
# repousser les écritures disques rèlles
# (très dangereux en cas d'arrêt brutal de la base ou de l'OS)
fsync = off
synchronous_commit = off
wal_sync_method = 'open_sync'

# repousser les checkpoints à un temps très long
checkpoint_timeout = 30min
# repousser le checkpoint pour qu'il n'intervienne qu'après
# un nombre très grands d'écritures de fichiers WAL
checkpoint_segments = 5000

# ici nous utilisons 0.00001, car 0.0001*30min est un chiffre très bas
# (180ms je crois) et que cela prendra de toute façon plus de
# temps dans la réalité
checkpoint_completion_target = 0.00001

# Si nous ne sommes pas sur windows on peut tenter une valeur
# très élevée du stockage des journaux de transactions en mémoire
# dans le segment de mémoire partagée
# sur windows un 64kb suffira, sur Linux 16MB est très bien
wal_buffers=16MB

# donnons plus de mémoire pour les opérations DDL
maintenance_work_mem = 500MB

# Assurons nous de ne pas avoir trop de logs
log_destination = 'stderr'
debug_print_parse = off
debug_print_rewritten = off
debug_print_plan = off
debug_pretty_print = off
# Ceci peut s'avérer utile pour vérifier que le checkpoint n'est
# pas lancé et regarder combien de buffers sont écrits lors des
# checkpoints manuels avec la commande CHECKPOINT;
log_checkpoints = on
log_connections = off
log_disconnections = off
log_duration = off
log_error_verbosity = default
log_hostname = off
log_statement = 'none'
log_temp_files=-1

# désactive les stats
track_activities = off
track_counts = off

# désactive l'autovacuum
autovacuum = off

# On s'assure de ne pas charger trop de modules annexes
shared_preload_libraries = ''
custom_variable_classes = ''
# et on enleve les settings des modules complémentaires s'il y en a
```

On teste ce fichier (assurez d'avoir une copie du postgresql.conf original avant de mettre celui-ci en place)

```
cp postgresql.conf postgresql.orig.conf
cp postgresql.restore.conf postgresql.conf
/etc/init.d/postgresql-9.0 restart
```

➢ *Testez le script populate_app.php sur cette version de PostgreSQL. Vous devriez obtenir des gains de vitesse très très importants. Mais vous n'avez plus en face un serveur de base de données très sûr en terme de stockage disque.*

20.7.6. Créer un crash

On va retirer le DATADIR de postgreSQL, soit pendant qu'il tourne soit à l'arrêt, la situation est la même, vous n'aurez plus de PostgreSQL et la base est « perdue ». Sur Linux un `kill -9` des processus postgres pour tuer brutalement le serveur sera utile. Le mv du dossier pouvant ne pas suffire à lui faire perde ses descripteurs de fichiers.

Si nous prenons `/var/lib/pgsql/data` comme DATADIR il suffit de faire (en root):

```
mv /var/lib/pgsql/data /mnt/otherdisk/olddata;
mkdir /var/lib/pgsql/data;
chown postgres:postgres  mkdir /var/lib/pgsql/data;
```

Dans cet exemple nous avons à disposition les WAL archivés dans `/mnt/serveur/archives` et un tar des fichiers physiques de la base dans `/mnt/serveur/archives`.

➢ *Vous pourrez tenter une variante de la restauration en recopiant les WAL de `/mnt/otherdisk/olddata/pgxlog` dans `/var/lib/pgsql/data/pgxlog`*

> ☑ Faites une copie de vos fichiers `postgresql*.conf` ou refaites le backup, car ces fichiers là ne sont pas dans le tar que nous avons fais.

20.7.7. Lancer la restauration

➢ *Modifiez pg_hba.conf pour n'autoriser que les accès locaux. Par exemple en indiquant uniquement: `local all postgres 127.0.0.1/32 trust`*

➢ *Recopiez le dernier snaphot binaire effectué dans `/var/lib/pgsql/data` à la racine puis décompressez le (sous sa forme actuelle le tar a stocké les chemins absolus)*

```
cp /mnt/serveur/backup/backup.tar /
cd /
tar xvf backup.tar
```

➢ *vérifiez que le fichier postgresql.conf du DATADIR est bien le fichier optimisé pour les recovery*

➢ *Créez un fichier recovery.conf et stockez le dans le DATADIR. Vous pouvez utiliser une copie du sample ou bien n'indiquer que l'option `restore_command` dans ce fichier:*

```
restore_command = 'cp -i /mnt/serveur/archive/%f %p'
```

☑ Sur Windows on aurait quelque chose comme: `restore_command = 'copy "E:\\ExternalBackup\\archives\\%f" "%p"'`. Mais utiliser un **script** plus complexe qu'un simple **copy** peut-être utile, notamment pour s'assurer qu'un FILE NOT FOUND devrait retourner une code de sortie 0.

☑ Si votre `archive_command` compressait les WAL la `restore_command` doit les **décompresser**. Si vous utilisiez `pg_compresslog` vous devez utiliser `pg_decompresslog`.

Cette commande va permettre à PostgreSQL de retrouver des segments de WAL afin de les recopier dans le pg_xlog local s'il en a besoin pour remettre l'état binaire des fichiers physiques de la base à jour.

Pour le PITR regardez les paramètres `recovery_target_time` et `recovery_target_xid`.

☑ Attention: PostgreSQL va essayer de renommer le fichier `recovery.conf` en `recovery.old` à la fin du processus. **L'utilisateur postgres doit donc être en mesure de bouger ce fichier, n'oubliez pas de lui donner les droits sur recovery.conf**!

```
chown postgres recovery.conf
```

Vous pouvez retirer le backup_label du DATADIR. Ici nous utilisons les WAL archivés et nou sn'avons plus de WAL dans le 'vrai' pg_xlog. Ce fichier contient des infos sur les derniers pg_xlog locaux valides, mais ils ne sont plus là.

➢ *Démarrez PostgreSQL*

```
/etc/init.d/postgresql-9.0 start
```

Vous pouvez faire un `tail -f` sur le log actif afin d'observer la restauration.

➢ *S'il n'y a pas de messages de restaurations retentez le démarrage jusqu'à ce que ceux-ci apparaissent dans les logs.*

➢ *Attendez tant que les messages sont du type:*

```
CET FATAL:  the database system is starting up
```

➢ *La fin de la restauration se traduit par un message:*

```
CET LOG: database system is ready to accept connections
```

20.7.8. Finir la restauration: tout remettre en état

Après la restauration on lancera à la main (en SQL) un

```
CHECKPOINT;
```

Quand cette commande se termine la restauration est terminée et les fichiers écrits sur disques (ou dans le cache disque de l'OS au moins, vous pouvez taper `sync` dans une console root).

Il est conseillé ensuite d'éteindre postgreSQL puis de **remettre le fichier postgresql.conf original**.

Remettez aussi le **pg_hba.conf original** en place.

Redémarrez PostgreSQL

Tapez ensuite un

```
VACUUM ANALYZE
```

Cette commande va permettre à l'analyseur de remettre à jour toutes ses statistiques, PostgreSQL mettra moins de temps à effectuer les bonnes opérations en fonction de la taille réelle des données. Même si vous n'avez pas utilisé un fichier de configuration dédié à la restauration, dans lequel les statistiques étaient suspendues, lancer cette commande accélèrera le retour « à la normale ».

☑ Avec la restauration et la coupure des archivages de WAL la chaîne de PITR/WAL à été brisée. Il faut donc refaire un backup complet des fichiers binaires de la base, afin que ceux-ci puissent servir de base à la prochaine restauration.

20.8. Tests de restauration de dump

Nous avons vu une récupération de base à partir des journaux de transactions. Cela n'exclu pas la possibilité d'effectuer des opérations de dumps et de restauration de dumps (c'est par exemple obligatoire pour les migrations de version majeures).

➢ *Effectuez un dump au format compress de votre base avec l'option utf8.*

```
pg_dump --host localhost --port 5432 --username "postgres" --format
custom --blobs --encoding UTF8 --verbose --file
"/path/to/formation_dump.backup" "formation"
```

➢ *Testez la commande de restauration du dump avec les deux configurations postgresql.orig.conf et postgresql.restore.conf*

A titre d'exemple sur un serveur Windows (donc pas le plus performant pour postgreSQL) et avec un dump d'une base de 653MB – donnée obtenue avec un `pg_size_pretty(pg_database_size('name'));` -- la différence de configuration permet de passer de **10 minutes et 40s** à **2 minutes et 49s** (8410 buffers dans le checkpoint). Sur la base formation avec 5000 commandes le temps varie sur un poste Linux de 20s à moins de 1s.

Commande de test sur Windows:

```
set foo1=%TIME%
"C:/Program Files/PostgreSQL/9.0/bin\pg_restore.exe" --host localhost
--port 5432 --username "postgres" --dbname "formation" --disable-
triggers --no-data-for-failed-tables --clean --verbose
"C:\Temp\formation_dump.backup"
set foo2=%TIME%
echo %foo1%
echo %foo2%
```

Commande de test sur Linux:

```
time  pg_restore --host localhost --port 5432 --username "postgres"
--dbname "formation" --disable-triggers --no-data-for-failed-tables
--clean --verbose /path/to/formation_dump .backup
```

On peut alors tester la parallélisation sur pg_restore avec l'option `--jobs=nb` (uniquement pour les formats de dump « compress »). indiquez le nombre de processeurs de votre serveur et essayez de le multiplier par deux ensuite. Ce qui pourrait par exemple faire 4 jobs en parallèle pour la restauration (attention on ne peut plus utiliser `--single-transaction` ici).

```
time  pg_restore --host localhost --port 5432 --username "postgres"
--dbname "formation" --disable-triggers --no-data-for-failed-tables
--clean --verbose --jobs=4 /path/to/formation_dump.backup
```

Notez aussi qu'avec la configuration par défaut de postgresql.conf (la version non optimisée pour un restore) il y a des chances pour que vous n'observiez aucune différence avec la parallélisation. N'hésitez pas à augmenter les chiffre si vous avez des CPU multithreadé et autres techniques multi-coeurs.

Pour ceux qui veulent plus d'infos sur les restaurations de grosses bases de données voici une histoire utile: http://www.depesz.com/index.php/2009/09/19/speeding-up-dumprestore-process/ dans cet exemple la restauration se situe sur une phase de `pg_upgrade`, donc un changement de version majeure de PostgreSQl qui demande un passage par le dump & restore.

20.9. Intégrité des données

PostgreSQL s'assure que les données écrites sur le disque sont valides et bien écrites, en travaillant à plusieurs niveaux ces vérifications d'écritures, en stockant des pages mémoire dans les WAL, etc. Malheureusement une fois ces données stockées sur le disque la prise en compte des défaillances du disque n'est pas faire par PostgreSQL.

PostgreSQl ne possède pas de checksum CRC de vérification des erreurs hardware qui pourrait détecter une incohérence des données survenant après leur écriture sur disque http://wiki.postgresql.org/wiki/Todo#Source_Code (« Add optional CRC checksum to heap and index pages »). PostgreSQl se fie au système de fichier pour assurer l'intégrité de la base de données.

> ☑ Utiliser un système de fichier qui assure lui même cette intégrité des fichiers peut vous aider à éviter qu'un tel problème de survienne (comme Solaris ZFS, Btrfs, NILFS).

Enfin on trouvera sur cette page http://www.postgresql.org/docs/9.0/static/wal-reliability.html des conseils divers sur les réglages de disques sur les différents OS afin d'éviter par exemple que le disque stockant les WAL ne soit en mode cache d'écriture.

> ☑ Pensez aussi à spécifier les options du système de fichier dans les points de montage. Ainsi un système ext3 sera plus rapide avec `--data=writeback, noatime, nodiratime` sans que l'intégrité du système de fichier soit diminuée.

20.10. Exemple de Politique de backups

Nous présentons ici une politique de backups. Il peut en exister d'autres.

20.10.1. Backup incrémental

L'archivage des WAL est mis en place. A chaque fois qu'un fichier WAL de 16MB est près ou qu'il est trop vieux il est recopié dans un répertoire externe à la base (1er niveau de backup). Ce dossier est synchronisé sur un serveur distant toutes les heures (2ème niveau de backup des WAL).

Les fichiers WAL sont compressés, et les fichiers trop vieux (voir ci-dessous la partie snapshot) seront effacés du serveur de backup.

20.10.2. Snapshot

Toutes les nuits une copie binaire des fichiers physiques de la table devrait être effectuée. Cette copie serait effectuée pendant en parallèle du fonctionnement de l'archivage des journaux de WAL en utilisant les commandes de backup de PostgreSQL. Grâce à cette copie binaire plus à l'archivage des WAL (backup incrémental) des restaurations PITR

seront possibles. Le fait de disposer d'une version binaire de la table à J-1 permet de n'avoir pas trop de journaux WAL à rejouer lors des restaurations.

Suivant le nombre de jours que l'on veut pouvoir remonter sur un PITR on devra organiser le backup des différentes versions binaires de la base et des WAL associés (si on veut pouvoir remonter à n'importe quelle transaction survenue entre J-7 et J il faut une version de la base à J-7 et tous les WAL intervenus depuis).

20.10.3. Dump

Le rythme pourrait être quotidien. Extraction quotidienne d'un dump de la base au format `compress` (*.backup).

> ☑ Le dump est capable de détecter une corruption de page. Une corruption d'index ou d'une donnée l'intérieur d'un champ ne sera pas forcément détectée par contre.

Les dumps sont importants, y compris si un backup incrémental par archivage des WAL est en place car PostgreSQl ne possède pas de système de vérification d'intégrité des données qui sont réputées bien écrites sur le disque (cf partie intégrité des données).

Lors du dump tous les indexs sont parcourus et des « page faults » peuvent être détectées et faire échouer le dump.

En parallèle des dumps des différentes bases de données un `pg_dumpall` devrait être lancé pour sauvegarder les tables systèmes, les rôles et les GRANT d'accès aux bases.

```
pg_dumpall --globals-only -f DESTINATION
```

20.10.4. Réindexation

Une réindexation complète des bases, par exemple chaque semaine, peut permettre d'éviter la corruption des index par un problème hardware. Lors d'un `REINDEX` les index sont reconstruits de zéro. Si vous utilisez des index de type hash cela évitera aussi des corruptions d'index suite à des restaurations.

> ☑ Pour accélérer les opération de réindexation n'hésitez pas à modifier à la volée le paramètre `maintenance_work_mem` dans le script SQL avec une instruction `SET`.

Lors du REINDEX les tables qui subissent ces ré indexations sont lockées en écriture mais pas en lecture. Cela signifie qu'il est possible de faire tourner les réindexations en parallèle des opérations de dump. Pour une base `madatabase` les commandes à utiliser sont:

```
REINDEX SYSTEM madatabase;
REINDEX DATABASE madatabse;
```

> ☑ la commande `CLUSTER` par contre qui utiliserait un index pour réordonner le contenu d'une table sur disque bloquerait des opérations de lecture parallèles

20.10.5. Restaurations

En s'appuyant sur la politique de backup présentée, en cas de problèmes nous avons 4 cas:

cas 1)- il s'agissait d'un arrêt brutal du serveur (oups le fil), nous allons relancer le serveur et tout sera remis en place par le pg_xlog

Vous avez éteint le courant. Quand PostgreSQL va se relancer il va rejouer les transactions qui ne sont pas dans le stockage binaire des tables (en dehors de WAL qui ne sont pas passés au `checkpoint` la dernière écriture de `CHEKPOINT` en cours ne s'est peut-être même pas terminée proprement). **Vous n'avez rien à faire.** En fait il n'y a pas de problème,s PostgreSQl travaille pour vous.

Maintenant examinons les 2 cas ou vous n'arrivez pas à relancer le serveur à cause d'un problème un peu plus important. Par exemple le répertoire des binaires ou le répertoire des pg_xlog ont disparus (mauvaise journée quand même...)

cas 2)- vous avez votre backup de premier niveau des WAL

La commande d'archivage des WAL recopie les WAL quelque part, si vous avez encore ce « quelque part » vous n'avez pas tout perdu. Vous avez avec vous:

- les journaux de transactions du backup incrémental: leur âge est au pire de 15 minutes (`max_checkpoint`)
- vous avez le dernier snapshot de la base qui a eu lieu la nuit dernière donc vieux d'un jour maximum

La procédure résumée en ligne : http://www.postgresql.org/docs/9.0/static/continuous-archiving.html#BACKUP-PITR-RECOVERY mais peut être possédez vous une procédure encore plus détaillée, adaptée à votre cas, que vous avez déjà testé au moins une fois sur vos machines.

Nous avons déjà effectué cette procédure lors du teste de recovery. Ajoutons simplement qu'il ne faut pas hésiter à faire une copie du répertoire pg_xlog du DATADIR actuel s'il est encore présent.

- cas 3)- Vous n'avez plus le backup des WAL de 1er niveau

Allez chercher les WAL sur le 2ème niveau (serveur de backup). Vous devriez alors avoir:

- les journaux de transactions du backup incrémental: leur âge est au pire de 1 jour si votre backup de 2ème niveau est quotidien, une heure s'il est horaire
- vous aurez un snapshot de la base qui sera vieux d'une semaine (au pire) à 1 jour (au mieux)

Procédure:

- Allez cherchez les données sur le serveur de backup
- Effectuez la même procédure que pour le cas2, sauf que vous aurez sans doute perdu un jour de transaction, ou bien une heure de transactions (suivant le rythme de votre backup de 2ème niveau des WAL).

Procédure alternative:

- utilisez le dernier dump de la base. Qui devrait être vieux d'un jour au pire

case 4) Vous avez une corruption de donnée suite à un problème matériel

Vous pouvez détecter ce type de problème suite à un `pg_dumpall` ou à un `pg_dump`, ou bien certaines requêtes sont rejetées avec des erruers qui signalent que quelque chose est cassé.

- a) suspendez l'accès à PostgreSQL (modifiez lepg_hba.conf)
- b) essayez de backuper le maximum de choses de la base actuelle, mais le dump est cassé, donc:
  1. fixer le problème matériel si cela est possible

2. démarrez par un snapshot binaire des fichiers de la base (utilisez les scripts de backups prévus pour cela)
3. essayez de triturer pg_dump pour qu'il sauve un maximum de choses. Par défaut **pg_dump** s'arrête sur un « page fault ». Nous allons dire à PostgreSQL de vider les données qui provoquent des « page fault » **– ici nous perdrons des données –** et de continuer le dump sur les données valides. Ajoutez **temporairement** ce paramètre dans postgresql.conf et redémarrez postgreSQL.

> ☑ **Attention:** utilisez ce paramètre avec précautions et retirez ce paramètre ensuite
>
> ☑ **Perte de données**. Avec ce paramètre à chaque fois que PostgreSQL rencontre une page cassée sur le disque il la vide. Alors que par défaut il s'arrête.

```
zero_damaged_pages=on
```

- c) faites un **pg_dumpall complet (vous pouvez le faire avec --globals-only)**, vous aurez des avertissements, à chaque perte de données . N'oubliez pas de faire un pg_dump au format 'c' compress pour chaque base de donnée. Le format compress sera utilise pour les restaurations partielles alors que le pg_dumpall travaille en SQL pur.
- d) enlevez le paramètre `zero_damaged_pages` et redémarrez PostgreSQL
- e) Essayez d'identifier les données perdues. Un `diff` avec les précédents dump sauvegardés peut vous aider

1. f) faires un `REINDEX` sur tous les index existants ou sur les bases directement, il y a des chances que eux aussi aient été endommagés.

On réindexe d'abord le catalogue. L'instruction de réindexation a besoin d'une base de donnée en argument pour accéder au catalogue. Le catalogue est le même sur toutes les bases, ce n'est donc pas la peine de relancer `REINDEX SYSTEM` sur toutes les bases ensuite.

```
REINDEX SYSTEM nomdunebasededonnees ;
```

Puis pour chaque base:

```
REINDEX DATABASE dbname;
```

Si les index system (pg_catalog) sont cassés PostgreSQL pourrait refuser de démarrer. Démarrez alors PostgreSQL avec l'option -P pour qu'il effectue des vérifications sur les index system. Regardez aussi cette page pour voir ce que vous pourriez faire: http://www.postgresql.org/docs/9.0/static/runtime-config-developer.html

2. g) rétablissez le ph_hba.conf
3. h) faites une pause

20.11. Utiliser les WAL pour la réplication

Nous avons utilisé les WAL pour faire de la restauration. Ce type de fonctionnement peut être étendu pour faire de la réplication maitre esclave. Utiliser l'archivage des journaux de logs pour de la réplication signifie que cette réplication sera asynchrone (il faut attendre l'archivage d'un certain nombre de transactions et leur transfert sur l'esclave). Cette réplication existe depuis PostgreSQL 8.2 grâce à l'utilitaire `pg_standby` et est appelée `WARM STANDBY`, avec un défaut important qui est l'impossibilité d'accéder au serveur esclave tant qu'il est en mode réplication, il faut effectuer une bascule manuelle de cet esclave pour qu'il puisse prendre le relais du maître (il s'agit en fait d'une restauration en continu, pour être prêt à reprendre le service plus rapidement). PostgreSQL9 introduit une amélioration de cette réplication appelée `HOT STANDBY` qui permet l'accès en lecture au serveur esclave. Une deuxième amélioration

consiste à utiliser la réplication par flux de transactions (`STREAMING REPLICATION`) , et permet d'obtenir un esclave sans décalage dans le temps.

De très bons article publiés dans Linux Magazine France et rédigés en français par Guillaume Lelarge donnent des procédure détaillées sur la mise en place de tels système de réplication:

- http://www.dalibo.org/hs44_la_replication_par_les_journaux_de_transactions
- http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_1
- http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_2

20.11.1. Limites

Pour que deux serveurs fonctionnent en mode réplication par les journaux de transactions il faut qu'ils respectent certaines contraintes:

- il doivent avoir la mee version majeure de PostgreSQL. Le format binaire des données pouvent être modifié lors d'un changement majeur de version.
  - 9.0.4 et 9.0.5 seront compatibles
  - 9.0.4 et 9.1.0 ne le sont pas
- il faut être consistant au niveau du stockage binaire (32bit litlle endian != 64 bits big endian)

Une des autres limitation de ce type de réplication est qu'il concerne **l'ensemble d'un cluster** PostgreSQL, on ne travaille pas sur une base de donnée unique ou sur un set de tables unique (voir les replications par triggers type Slony et Londiste pour cela).

20.11.2. WARM STANDBY

Obtenir des serveurs en WARM STANDBY est assez proche de la problématique de la restauration basée sur les WAL. Travaillez en binôme. L'un des deux serveurs sera le maître et l'autre l'esclave.

➢ *On commence par arrêter les serveurs PostgreSQL sur le maître et l'esclave.*

Sur le maître on garde notre configuration où les WAL sont archivés avec l'`archive_command`. Nous allons simplement modifier cette commande pour que l'archivage sur fasse sur le serveur esclave. Nous avions:

```
archive_command = 'test ! -f /mnt/serveur/archive/%f && cp -i %p
/mnt/serveur/archive/%f </dev/null'
```

Nous le transformons en:

```
archive_command = 'scp "%p" "serveuresclave:/mnt/serveur/archive/%f"'
```

Où `serveuresclave` est le nom ou l'adresse IP du serveur esclave.

> ☑ Nous pourrions garder notre commande d'archivage en utilisant un système de fichier réseau comme NFS pour monter le répertoire distant du serveur esclave sur le `/mnt/serveur/archive` du maître.

Il faut aussi bien sur tester une commande `scp` avec le user postgres à destination de cette machine (par exemple en recopiant une vieiile archive que nous avions dans notre `/mnt/serveur/archive` dans le même répertoire sur l'autre machine). Nous avons écris un commande scp où la connexion ssh se fait sans spécifier d'utilisateur ou de mot

de passe, ceci est possible en configurant une clef ssh pour l'utilisateur postgres et en la déployant sur le serveur distant.

```
> sudo su – postgres
> ssh-keygen -t dsa
# nous devrions obtenir une clef : /var/lib/postgresql/.ssh/id_rsa.pub
# nous recopions cette clef sur le serveur esclave
# (il faut le mot de passe root distant), c'est une commande sur
# une seule ligne
> scp /var/lib/postgresql/.ssh/id_dsa.pub \
   root@esclave:/tmp/id_dsa_master.pub
```

Sur le serveur esclave il faut installer cette clef de le HOME de l'utilisateur postgres, dans le fichier `.ssh/authorized_keys`. Nous aurions pu automatiser cette étape en utilisant `ssh-copy-id` depuis le serveur escalve, mais il faudrait que l'utilisateur postgres possède un mot de passe.

Sur le serveur esclave on passe donc root et on tape ces commandes:

```
> sudo su -
> # on en profite pour vider le répertoire d'archivage du serveur
esclave.
> rm -rf /mnt/serveur/archive/
> # on devient l'utilisateur postgres pour installer
> #les autorisations ssh
> chown postgres /tmp/id_dsa_master.pub
> su – postgres
> mkdir ~/.ssh
> cat /tmp/id_dsa_master.pub >> ~/.ssh/authorized_keys
```

Sur le serveur maître on teste que le `scp` fonctionne:

```
> scp /mnt/serveur/archive/00000002000000000000050 \
  serveuresclave:/mnt/serveur/archive/
```

On peut alors relancer le serveur, l'archivage devrait se faire dans le dossier /mnt/serveur/archive du serveur esclave (vous pouvez commencer à lancer le script populate_app.php).

On en profite aussi pour relancer notre script de backup afin d'acvoir une copie binaire de la base à donner à l'esclave comme point de départ.

```
> sudo ./backup.sh
# puis on recopie ce backup binaire sur l'esclave, quelque part
> scp /mnt/serveur/backup/backup.tar \
   serveuresclave:/mnt/serveur/backup/
```

Sur l'esclave on modifie quelques éléments:

– on utilise le backup binaire que l'on a reçu du maître pour initialiser le contenu des fichiers physiques de la base

```
cp /mnt/serveur/backup/backup.tar /backup.tar
[root@localhost ~]# cd /
[root@localhost /]# tar xfv backup.tar
```

☑ Vérifiez que les deux serveurs sont à la même heure! Utilisez un protocole comme ntp pour avoir des serveurs à l'heure.

– on suspend l'archivage des WAL et on passe le `wal_level` à sa valeur par défaut. Sur l'esclave on n'a pas besoin de générer une deuxième version des journaux de transactions

```
wal_level = minimal
archive_mode = off
```

> ☑ **ATTENTION: en recopiant le dossier physique venant du maître on a sans doute recopié son pg_hba.conf et son fichier postgresql.conf.** Il faut donc bien modifier ces valeurs de configuration **après** la copie. Un script pourrait utiliser du sed, ou recopier un fichier de configuration sauvegardé sous un autre nom.

Nous ne démarrons toujours pas le serveur sur l'esclave. Au préalable il nous faut lui donner un fichier `recovery.conf,` afin qu'il fonctionne comme un serveur en mode restauration. Nous créons donc un fichier `recovery.conf` dans son répertoire de données dans lequel nous indiquons une commande de restauration un peu spéciale puisqu'elle utilise le programme pg_standby. (faites un « `locate pg_standby` » pour trouver le votre, qui fais partie des programmes contrib de postgreSQL).

```
# cette commande tient sur une ligne
restore_command = '/path/to/9.0/bin/pg_standby -d -t \
   /tmp/trigger_stanby_end /mnt/serveur/archive %f %p %r \
   >>/var/log/postgresql/pg_standby.log 2>&1'
```

Nous pouvons voir que le programme va essayer de faire un log de ce qui lui arrive dans `/var/log/postgresql/pg_standby.log`, on va donc initialiser ce fichier et autoriser postgres à écrire dans ce fichier. On vérifiera aussi que le DATADIR récupéré du maitre appartient bien à l'utilisateur postgres.

```
> chown -R postgres /path/to/postgresql/data/*
> touch /var/log/postgresql/pg_standby.log
> chown postgres /var/log/postgresql/pg_standby.log
```

Enfin on peut démarrer notre serveur esclave

```
/etc/init.d/postgresql start
```

Celui-ci se lance en mode restauration.

Pour visualiser ce que le serveur est en train de faire vous pouvez lancer ces commandes:

```
ps auxf|grep postgres
tail -f /var/log/postgresql/pg_standby.log
```

Si la commande de popualtion de la base (populate_app.php) tourne sur le master on pourra visualiser l'arrivée progressive des WAL sur l'esclave.

Par contre toute tentative d'accès direct à la base esclave est impossible.f

```
psql -Upostgres -d postgres -h localhost -p 5432
```

Par rapport à une resturation classique nous avons le programme pg_standby qui est en fait en attente d'un fichier que nous lui avons indiqué dans la commande. Tant que ce fichier « trigger file » `/tmp/trigger_stanby_end` n'existe pas pg_standby force le serveur à rester en mode restauration (en attente de WAL), et donc le serveur est injoignable.

Arrêtons la réplication en créant ce fichier:

```
touch /tmp/trigger_stanby_end
```

On obtient alors une base indépendant de la base maître. La création de ce « fichier trigger » est donc plutôt à la charge d'un service comme keepalived qui démarre le service sur l'esclave lors d'une bascule.

- repasser l'esclave en statut esclave demande de repartir d'un backup binaire du maître
- une bascule retour (failback) devrait être prévue dans vos procédures, sans doute à partir d'une sauvegarde de ce nouveau maître (peut-être alors faudra-t-il activer l'archivage des WAL sur cet esclave afin d'effectuer un backup binaire)

> ☑ A noter: il existe des outils libres pour simplifier la mise en œuvre d'une réplication `WARM STANDBY`: **walmgr** de Skype et **pitrtools** de Command Prompt.

20.11.3. HOT STANDBY

PostgreSQL 9 introduit une variation sur le Warm Standby qui est donc le Hot Standby. Dans ce mode la connexion au serveur esclave est possible et nous affiche le contenu de la base en décalé (l'esclave ne dispose que des journaux transférés – on parle de log shipping).

> ☑ On peut réduire le décalage en jouant sur des paramètres archive_timeout de l'ordre de quelques secondes sur le maître si les canal de transfert des fichiiers WAL est rapide entre le maître et l'esclave et que les scripts d'archivage et de restauration font un nettoyage efficace des WAL qui ne sont plus utiles.

➢ *On commence par **arrêter** le serveur esclave qui est sans doute devenu maître à la fin de l'exercice précédent, et on supprime le fichier* `/tmp/trigger_stanby_end`

➢ *On modifie le fichier postgresql.conf du maître pour passer à un niveau un peu supérieur de WAL:*

```
wal_level = 'hot_standby'
```

Puis comme dans le `WARM STANDBY` nous devons:

- redémarrer le serveur maître
- effectuer un backup binaire
- transférer ce backup sur l'esclave.
- Décompresser le backup sur l'esclave
- s'assurer que tous les fichiers appartiennent bien à postgres

➢ *Là nous allons modifier le fichier postgresql.conf de l'esclave qui doit contenir le paramétrage du maître, pour lui indiquer le paramétrage de l'esclave en hot_standby, en gras j'indique ce qui change par rapport au warm standby:*

```
wal_level = minimal
archive_mode = off
hot_standby = on
```

Ensuite nous devons remettre en place un fichier recovery.conf, identique à celui du warm standby, contenant la commande pg_stanby. Et enfin nous pouvons démarrer

```
/etc/init.d/postgresql-9.0 start
```

Nous obtenons un serveur en lecture seule et en décalage léger avec le maître.

➢ *En utilisant `populate_app.php` sur le maître constatez les différences entre les deux serveurs en effectuant des requête de `count(*)` sur `app.commandes`.*

20.11.4. STREAMING REPLICATION

Pour obtenir un serveur de type hot standby avec une latence plus court, un temps plus court de répercution des WAL, on peut donc réduire l'archive_tiemout. Mais une meilleure solution existe. La réplication par flux va ajouter aux Hot Standby un flux d'envoi direct des transactions entre le maître et ses esclaves.

Au niveau du ùaître des processus `wal_sender` vont se charger d'envoyer des informations en flux tendus à des processus `wal_receiver` situés au niveau des esclaves. Plusieurs nouveaux paramètres entre en jeu:

- `max_wal_senders` : nombre de processus chargés de la synchronisation au niveau du maître (un par esclave)
- `wal_sender_delay` : délai d'attente, par défaut à 200ms entre chaque « exécution » du cycle de synchronisation, la valeur doit être un multiple de 10ms
- `wal_keep_segments` : nombre de WAL qui peuvent être conservés dans pg_xlog pour la réplication flux. Si l'esclave prends du retard et que les segments ne sont plus dans pg_xlog il devra attendre la récupération via l'archivage des WAL (comme en hot_standby ou warm standby classique). Le défaut à 0 signifie que le maître ne fais pas attention à conserver ou pas des WAL pour la réplication flux, il gère ces WAL dans le pg_xlog comme d'habitude, en fonction des `CHECKPOINT` principalement.

> ☑ Si l'esclave ne fonctionne qu'avec un wal_receiver et sans restore_command capable de récupérer des WAL archivés il faut absolument spécifier un nombre assez élevé dans `wal_keep_segments` afin d'éviter de désynchroniser un esclave en cas de charge d'écriture importante qu'il n'aura su effectuer dans les temps.

Cette réplication 'en flux tendu' va demander des connexions depuis l'esclave vers le maître. Il faut donc que l'esclave utilise un compte utilisateur (éventuellement avec un mot de passe) et que le maître autorise la connexion avec cet utilisateur depuis l'adresse IP de l'esclave.

➢ *Il va donc nous falloir éditer le `pg_hba.conf` du maître. L'accès se fait sur une base 'spéciale' nommée `replication`. Nous ajoutons donc en fin de fichier `pg_hba.conf` cette ligne (adaptez l'adresse IP à votre cas):*

```
# TYPE  DATABASE        USER            CIDR-ADDRESS            METHOD
host    replication     aicha        192.168.1.13/24            trust
```

➢ *On ajoute ensuite le 1 processus d'envoi des WAL en flux dans le postgresql.conf du maître (qui est déjà configuré pour du hot_standby):*

```
max_wal_sender = 1
```

➢ *faites un restart du serveur PostgreSQl du maître*

```
/etc/init.d/postgresql-9.0 restart
```

Si nous regardons du côté du serveur **esclave** nous avons toujours notre esclave en `HOT STANDBY`, avec un recovery.conf actif. Nous allons devoir l'adapter au niveau de son `recovery.conf` pour qu'il se mette en écoute du flux de transactions du maître. Nous allons le configurer pour la réplication de flux en indiquant quelque sparamètres supplémentaires dans ce fichier, dont le principal est standby_mode. A partir du moment où nous entrons dans ce mode l'utilitaire pg_standby que nous utilisions pour le HOT STANDBY n'est plus utile (on a de nouvelles options de

configurations pour le trigger file par exemple) et la commande de restauration se simplifie pour n'être plus qu'une simple copie des fichiers d'archive:

➢ *On modifie donc le recovery.conf (pas le postgresql.conf) de cette façon (ici 192.168.1.10 est l'IP du maître):*

```
standby_mode = 'on'
restore_command = 'cp /mnt/serveur/archive/%f %p'
primary_conninfo = 'host=192.168.1.10 port=5432 user=aicha'
trigger_file = '/tmp/trigger_stanby_end'
```

☑ Il faut un user **superutilisateur** de la base. Vérifiez que l'utilisateur possède ce droit. Ici nous indiquons trust en nous filtrons sur l'adresse IP plus le masque de sous-réseau de l'esclave. Vous pourriez utiliser md5 pour forcer la vérification du mot de passe.

➢ *Consultez les logs de l'esclave pour d'éventuels problèmes de connexion et pour observer la réplication (faires un `grep replication` sur les fichiers de log).*

➢ *Faites tourner populate_app.php et faites des requêtes de comptage sur le maître et l'esclave, observez la réplication qui semble instantanée.*

Cela nous permet de faire un premier aperçu de la réplication par PostgreSQL. Nous obtenons une réplication maître esclave quasi-synchrone. Le risque étant un décalage sur le serveur esclave à cause de requêtes en lecture longues qui posent des locks sur des opérations d'écritures importantes. Une fois encore je citerai l'article de Guillaume Lelarge:

– http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_2

Vous trouverez dans cet article des réglages assez fins des problématiques de lock de lecture (sic) et des principes de bascules (switchover, failover, failback).

20.12. Autres systèmes de réplication

D'autres système de réplication existent autour de PostgreSQL. Certains sont utilisées depuis très longtemps. On citera les principaux:

– **SLONY**: réplication par les triggers. Historiquement Slony était le principal outil de réplication pour les solutions Web sur lesquelles on voulait disposer d'un esclave accessible en lecture et synchrone avec les modifications de la base. Slony impose de ne pas modifier le schéma de la base et de disposer d'une liste des tables et des clefs primaires de chaque table. Slony se charge ensuite, base par base, table par table, de répercuter les modifications quand elles arrivent sur les esclaves (un serveur peut être maître d'une base ou d'une partie des tables de la base, et esclaves sur d'autres bases et/ou tables)

– **Pgpool II** : réplication des requêtes. Pgpool est un pooler de connexion, une des fonctionnalités offertes par un pooler est de répercuter sur tous les serveurs d'une grappe l'ensemble des requêtes effectuant des opérations en écriture. Si toutes les connexions passent bien par le pooler et que les requêtes ne font pas appel à des valeurs aléatoires on obtient des bases identiques.

– **Londiste**: réplication par les triggers. Utilitaire de réplication de Skype. Sa configuration est plus simple que celle de SLONY, par contre sa documentation est encore plus succinte que celle de SLONY, qui n'est déjà pas un modèle du genre.

- **Bucardo**: réplication master-master : le système le plus complexe et le plus avancé, vous obtenez un cluster de serveurs PostgreSQL dans lequel vous pouvez effectuer vos écritures sur n'importe quel serveur
- **DRBD** : DRBD est une solution de réplication des disques entre serveurs, il ne s'agit donc pas d'une réplication de base de donnée. Les deux serveurs, le maître et l'esclave, partagent un même disque dur. Toutes les écritures effectuées sur le disque du maître sont répliquées sur le disque de l'esclave. On utilise le plus souvent DRBD pour faire des bascules d'urgence, l'esclave peut à tout moment reprendre la main sur le maître avec une bascule de type keepalived, il devient le maître de DRBD et peut alors lancer sa propre version de la base qui disposera des mêmes fichiers physiques (un partage DRBD devrait inclure le partage du pg_xlog pour éviter d'avoir à faire une restauration). Certains système des fichiers avancés comme OCFS-2 peuvent supporter des écritures concurrentes sur le partage DRBD, mais il serait dangereux d'utiliser DRBD dans ce sens avec des systèmes de base de données.

20.13. Autres outils

20.13.1. Monitorer PostgreSQL

http://bucardo.org/wiki/Check_postgres

Pour intégrer la supervsion de PostgreSQl dans vos solutions de monitoring le principal outil sera la sonde Nagios check_postgres, sonde écrite en Perl (mais très orientée Linux). Cette sonde est maintenue par Bucardo, un des acteurs majeurs du monde PostgreSQL. L'ensemble des vérifications pouvant être effectuées est assez important. Vous devrez créer plusieurs services Nagios utilisant cette sonde de différentes manière. Ces différentes vérifications sont les Actions de la sonde dont vous pouvez voir la liste sur cette page:

- http://bucardo.org/check_postgres/check_postgres.pl.html

En terme de supervision passive (graphiques) on pourra consulter ces liens:

- http://wiki.postgresql.org/wiki/Cacti (Cacti)
- http://munin-monitoring.org/wiki/PluginCat (Munin)
- http://muninpgplugins.projects.postgresql.org/ (Munin)
- http://tigreraye.org/Modules%20PostgreSQL%20pour%20Munin (Munin)

On trouvera beaucoup plus de ressources, et d'un meilleur niveau, pour Munin que pour Cacti.

20.13.2. PgAgent

http://www.pgadmin.org/docs/1.14/pgagent.html

PgAgent est un programme complémentaire de pgAdmin qui permet la mise en place de scripts de maintenance récurrents. On peut l'utiliser pour planifier des scripts SQL de traitements batchs, ou pour simplement faire des appels à des procédures stockées de batchs. PgAgent peut aussi lancer des scripts systèmes et découper ses « jobs » en plusieurs étapes (steps).

20.13.3. PgPool II

- http://pgpool.projects.postgresql.org/
- http://pgpool.projects.postgresql.org/pgpool-II/doc/pgpool-fr.html

pgpool II est un epooler de connexions. Un des apports important d'un pooler de connexion est de pouvoir mettre en attente les demandes de connexions supplémentaires plutôt que de les rejeter. Vous obtiendrez, en cas d'un nombre de connexions trop élevé, des lenteurs d'accès (plus ou moins longue suivant le dépassement), mais aucun message d'erreur. Il y a un effet de lissage.

L'autre apport utile consiste à utiliser le pooler pour de la répartition de charge sur un ensemble de serveurs. Si vous disposez d'esclaves en lecture seule avec des données synchrones, le pooler pourra répartir la charge des requêtes hors-transaction sur ces serveurs (les select au sein de transactions devant bien sur rester sur le serveur de la transaction).

Enfin l'apport premier d'un pooler est de maintenir des connexions ouvertes sur le serveur pour éviter les latences dues au temps d'établissement de ces connexions.

Mais si vous lisez en détail la documentation de pgpool II vous découvrirez certainement de nombreuses autres applications utiles (réplication des requête, parallélisation de traitement, bascules failover, etc).

20.13.4. PgSnap!

http://pgsnap.projects.postgresql.org/

http://pgsnap.projects.postgresql.org/pagila2_snap_20111029/ (démo)

PGSnap! Est un programme PHP qui génère un rapport sur l'état de la base. Je devrais plutôt dire qu'il génère un ensemble de rapports. Il s'agit d'un bon outil complémentaire de la supervision et qui permettra à un DBA d'avoir une vision de ses serveurs à la fois synthétique et détaillée pour les éventuels problèmes.

- *Parcourez la démonstration pour découvrir les différents rapports. Remarquez la possibilité de demander les requêtes effectuées sur le catalogue pour pouvoir les réutiliser de votre côté en les adaptant.*

20.13.5. pgfouine

http://pgfouine.projects.postgresql.org/

pgFouine est un programme PHP , c'est un analyseur de logs.

- *Examinez les démonstrations de rapports générés par pgFouine sur cette page http://pgfouine.projects.postgresql.org/reports.html*

C'est un bon outil en complément de pgSnap ou d'une recherche des requêtes provoquant des erreurs ou des lenteurs.