−Table des matières

Mise en place de la réplication avec PostgreSQL 9.0 - 2/2

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 132 (Novembre 2010). Il est disponible maintenant sous licence Creative Commons.

Lors du précédent article, nous avons vu comment mettre en place un serveur maître et son serveur esclave. Nous allons aller un peu plus loin dans cet article en proposant maintenant une réplication beaucoup plus synchronisée (mais toujours pas synchrone).

Nous allons aussi passer en revue un certain nombre de points importants à connaître pour administrer au mieux ses serveurs en réplication. Tout d'abord, le problème posé par les requêtes en lecture seule. Ces dernières posent des verrous et, aussi peu gênants soient-ils dans une utilisation normale, ils peuvent poser quelques soucis majeurs pour un serveur esclave. Nous verrons aussi comment effectuer une bascule dans deux cas : la bascule volontaire (« switchover ») et la bascule forcée (« failover »). Enfin, nous verrons comment surveiller l'état de la réplication et des deux serveurs concernés.

Mise en place du Streaming Replication

Le Streaming Replication, ou réplication en flux, permet d'envoyer les enregistrements des journaux de transactions par groupe inférieur à un journal de transactions complet. Le but est d'avoir une réplication plus rapidement synchrone. Au lancement de PostgreSQL, avec le Streaming Replication activé, le serveur va commencer par récupérer tous les journaux de transactions que la commande indiquée par restore_command est capable de lui fournir. Une fois que la commande renvoie un code d'erreur, il vérifiera si le journal qu'il recherche se trouve dans le répertoire pg_xlog. Enfin, s'il ne trouve rien là non plus, il basculera en mode streaming. Ce mode lui permet de recevoir directement les enregistrements des journaux de transactions.

Pour envoyer (à partir du serveur maître) et recevoir (sur le serveur en standby) les enregistrements, deux processus font leur apparition : le processus « wal sender » va envoyer les enregistrements de journaux de transactions à un processus dénommé « wal receiver » qui les reçoit et les applique.

WAL est l'acronyme de Write Ahead Log (autrement dit, écriture des journaux avant), la technologie à la base des journaux de transactions de PostgreSQL.

Le processus « wal sender » est lancé par le serveur maître. Il est d'ailleurs erroné de parler d'un processus. Il est possible d'en avoir plusieurs, un pour chaque serveur en hotstandby connecté à ce serveur maître. Un paramètre permet de configurer le nombre maximum de connexions. Son nom est max_wal_senders. Donc, pour mettre en place le Streaming Replication, la première action à entreprendre est la modification du fichier de configuration postgresql.conf sur le serveur maître.

Le processus « wal sender » est responsable de l'envoi des enregistrements de transactions au processus « wal receiver ». En fait, il fait cela dans une boucle : transfert des enregistrements disponibles et non envoyés, attente, transfert des nouveaux, attente, etc. La durée de l'attente est configurable avec le paramètre wal_sender_delay. Ce dernier vaut par défaut 200ms, ce qui est suffisamment rapide pour notre démonstration. Nous ne le changerons pas.

Nous configurons donc le maître pour accepter la connexion d'un serveur en hotstandby :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ cd ../../maitre
guillaume@laptop:~/tests/maitre$ cat >> maitre.conf << _EOF_
> max_wal_senders = 1
> _EOF_
```

Remarquez que nous conservons le paramétrage effectué pour le Hot Standby. Nous ne faisons ici que rajouter la nouvelle configuration du paramètre max_wal_senders. Sa configuration est fixée à 1 car nous n'aborderons que le cas où un seul serveur en standby se connecte au serveur maître.

Le processus « wal receiver » est lancé par le serveur en standby. Ce processus va commencer par se connecter au serveur maître. Il faut donc modifier le fichier de configurations des accès distants, le fichier pg_hba.conf, pour autoriser cette connexion. Attention, le processus va demander la connexion à une base virtuelle appelée replication. La configuration du fichier pg_hba.conf doit l'indiquer très précisément. Mettre all sur la colonne des bases de données ne permettra pas la connexion de ce processus au serveur maître. Voici le contenu du fichier :

```
# TYPE  DATABASE        USER            CIDR-ADDRESS          METHOD

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

La ligne ajoutée est la ligne suivante :

```
host    replication     guillaume       127.0.0.1/32          md5
```

Attention, l'ordre des lignes a une importance.

Nous autorisons la connexion de l'hôte localhost (adresse au format IPv4) si la connexion indique l'utilisateur guillaume et la base replication. Il est essentiel que l'utilisateur, ici « guillaume », soit un rôle ayant les attributs de superutilisateur (SUPERUSER) et de connexion (LOGIN). Comme nous indiquons une authentification par mot de passe, nous devons donner un mot de passe à l'utilisateur guillaume :

```
guillaume@laptop:~/tests/maitre$ psql -q b1
b1=# ALTER USER guillaume PASSWORD 'supersecret';
```

```
b1=# \q
```

Le paramétrage du serveur maître étant terminé, nous pouvons le relancer :

```
guillaume@laptop:~/tests/maitre$ pg_ctl -D /home/guillaume/tests/maitre restart
en attente de l'arrêt du serveur.... effectué
serveur arrêté
serveur en cours de démarrage
```

Il nous reste à configurer le serveur en standby. Tout le travail se fera sur le fichier recovery.conf. Commençons par les paramètres de l'attente. Il faut activer le mode standby_mode (autrement dit, mettre ce paramètre à on). Au démarrage, PostgreSQL comprendra que, si la commande indiquée par le paramètre restore_command n'est pas capable de lui fournir le journal de transactions qu'il attend, il ne doit surtout pas arrêter le mode de restauration mais plutôt se mettre en attente des enregistrements des transactions via son processus « wal receiver ». Il exécutera alors ce processus, qui va se connecter au maître. Il faut donc lui indiquer comment se connecter à ce serveur. C'est le but du paramètre primary_conninfo (informations de connexion vers le serveur primaire). Ce dernier est une simple chaîne de connexion habituelle ressemblant fortement à un DSN. Un ensemble de paramètres peuvent être indiqués sous la forme suivante : paramètre=valeur. Chaque paramètre doit être séparé des autres par un espace. Les paramètres sont assez nombreux mais les plus fréquents sont host (alias ou adresse IP où se connecter), port (le port de connexion TCP), user (le nom de l'utilisateur), etc. Comme le serveur va rester en attente des enregistrements de transactions en permanence, il faut quand même un moyen pour lui demander d'abandonner l'attente. Avec l'outil pg_standby, on avait la possibilité de fournir un fichier trigger dont la présence déclenche l'abandon de l'attente. Le fichier recovery.conf dispose d'une variable partageant ce but. Elle se nomme trigger_file.

```
guillaume@laptop:~/tests/maitre$ cd ../hotstandby
guillaume@laptop:~/tests/hotstandby$ cat > recovery.conf << _EOF_
> restore_command = 'cp /home/guillaume/tests/archives_xlog/%f %p'
> standby_mode = 'on'
> primary_conninfo = 'host=127.0.0.1 port=5432'
> trigger_file = '/tmp/stopstandby'
> _EOF_
```

Remarquez aussi que le paramètre restore_command est à modifier. En effet, nous ne voulons plus que la commande exécutée attende le prochain journal de transactions. C'est le serveur PostgreSQL qui s'occupe de cette attente. Du coup, nous rebasculons la commande à un simple cp.

Il est tout à fait possible de se passer de restore_command. Dans ce cas, les journaux seront envoyés par le processus « wal sender ». Le problème de cette configuration est qu'un nombre minimum de journaux de transactions doit être conservé sur le serveur maître. Or, une fois que le journal est archivé, ce dernier les renomme dès que possible pour pouvoir les ré-utiliser. Un nouveau paramètre est donc apparu pour spécifier un nombre de journaux à conserver en cas de lag du serveur en hotstandby. Ce paramètre, appelé wal_keep_segments, a 0 comme valeur par défaut. En cas de non-utilisation du paramètre restore_command, il faut augmenter ce paramètre à une valeur suffisamment haute pour ne pas risquer le recyclage d'un journal de transactions qui pourrait toujours être utile au serveur en hotstandby. Ce qui reste assez difficile à estimer.

Dernière configuration avant de relancer le serveur en attente, le passage du mot de passe de l'utilisateur guillaume. En effet, la connexion va se faire sur le serveur maître à la condition que le mot de passe soit saisi. Il est tout à fait possible d'indiquer le mot de passe directement dans la chaîne du paramètre primary_conninfo. Il est aussi possible d'utiliser le fichier .pgpass. Ce fichier doit se trouver dans le répertoire personnel de l'utilisateur qui exécute PostgreSQL. C'est un fichier tabulé comprenant plusieurs colonnes (alias ou adresse IP, numéro de port, nom de la base, nom de l'utilisateur, et enfin son mot de passe). Créons ce fichier :

```
guillaume@laptop:~/tests/hotstandby$ cat >> ~/.pgpass << _EOF_
> 127.0.0.1:5432:replication:guillaume:supersecret
> _EOF_
```

Pour que personne d'autres ne puisse le lire, nous devons mettre les droits 600 (lecture/écriture uniquement pour le propriétaire du fichier, aucun droit pour les autres, y compris le groupe) sur ce fichier.

```
guillaume@laptop:~/tests/hotstandby$ chmod 600 ~/.pgpass
```

Attention, si le fichier n'a pas exactement ces droits là, le fichier ne sera pas utilisé.

Il ne nous reste plus qu'à redémarrer le serveur en attente :

```
guillaume@laptop:~/tests/hotstandby$ pg_ctl -D /home/guillaume/tests/hotstandby restart
en attente de l'arrêt du serveur.... effectué
serveur arrêté
serveur en cours de démarrage
```

En lisant les traces du serveur maître, nous pouvons apercevoir cette ligne :

```
guillaume@laptop:~/tests/hotstandby$ tail -1 ../maitre/pg_log/postgresql-2010-08-21_105848.log
2010-08-21 11:02:49 CEST LOG:  replication connection authorized: user=guillaume host=127.0.0.1 port=46031
```

Quant aux traces sur l'esclave, elles indiquent la ligne suivante :

```
guillaume@laptop:~/tests/hotstandby$ tail -1 pg_log/postgresql-2010-08-21_110249.log
2010-08-21 11:02:49 CEST LOG:  streaming replication successfully connected to primary
```

Autrement dit, au démarrage du serveur en hotstandby, PostgreSQL a récupéré les journaux qu'il pouvait via la commande indiquée par le paramètre restore_command, puis a cherché dans son répertoire pg_xlog. Une fois ces restaurations terminées, il a lancé un processus « wal receiver », qui lui a tenté une connexion vers le serveur maître (les paramètres de connexions étant fournis par le paramètre primary_conninfo). Le serveur maître ayant autorisé la connexion et compris qu'il s'agissait d'une connexion de réplication a démarré un processus « wal sender ». Voici ce que cela donne sur mon système :

```
guillaume@laptop:~/tests/hotstandby$ ps -ef | grep postgres
1000      7783     1  0 10:58 pts/0    00:00:00 /opt/postgresql-9.0/bin/postgres -D /home/guillaume/tests/maitre
1000      7787  7783  0 10:58 ?        00:00:00 postgres: logger process
1000      7789  7783  0 10:58 ?        00:00:00 postgres: writer process
1000      7790  7783  0 10:58 ?        00:00:00 postgres: wal writer process
1000      7791  7783  0 10:58 ?        00:00:00 postgres: autovacuum launcher process
1000      7792  7783  0 10:58 ?        00:00:00 postgres: archiver process
1000      7793  7783  0 10:58 ?        00:00:00 postgres: stats collector process
1000      7883     1  0 11:02 pts/0    00:00:00 /opt/postgresql-9.0/bin/postgres -D /home/guillaume/tests/hotstandby
1000      7891  7883  0 11:02 ?        00:00:00 postgres: logger process
1000      7892  7883  0 11:02 ?        00:00:00 postgres: startup process   recovering 000000010000000000000011
1000      7895  7883  0 11:02 ?        00:00:00 postgres: writer process
1000      7896  7883  0 11:02 ?        00:00:00 postgres: stats collector process
1000      7899  7883  0 11:02 ?        00:00:00 postgres: wal receiver process   streaming 0/11000078
1000      7900  7783  0 11:02 ?        00:00:00 postgres: wal sender process guillaume 127.0.0.1(46031) streaming 0/11000078
1000      8021  3494  0 11:04 pts/0    00:00:00 grep --color=auto postgres
```

Les deux processus intéressants sont à la fin. Le premier, « wal receiver » a été lancé par le serveur en hotstandby, comme l'indique son identifiant de processus père (colonne PPID, valant 7883 pour ce dernier). Le second, « wal sender » a été lancé par le serveur maître suite à la connexion réussie du « wal receiver ». Là-aussi, on le voit grâce à son identifiant de processus père (le PPID valant 7783).

Les informations de statut que nous renvoie la commande ps sont intéressants. Nous savons ainsi que le « wal receiver » s'est connecté au serveur maître en utilisant l'utilisateur guillaume, et qu'il s'est connecté via l'adresse IP 127.0.0.1 et le numéro de port 46031. Nous apprenons qu'actuellement, le « wal sender » a envoyé l'enregistrement de transactions 0/11000078 et que le « wal receiver » l'a bien reçu.

Essayons maintenant de faire des ajouts sur le serveur maître :

```
guillaume@laptop:~/tests/hotstandby$ psql -q b1
b1=# CREATE TABLE t2(c1 integer);
CREATE TABLE
b1=# \d
```

```
        Liste des relations
 Schéma | Nom | Type  | Propriétaire
--------+-----+-------+--------------
 public | t1  | table | guillaume
 public | t2  | table | guillaume
(3 lignes)
```

Après la création de la table t2, nous voyons qu'elle est bien créée sur le serveur en standby.

Insérons des lignes dans cette nouvelle table :

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
b1=# SELECT count(*) FROM t2;
  count
---------
 1000000
(1 ligne)
```

Une courte vérification sur le serveur en hoststandby montre que les lignes se trouvent bien immédiatement sur le serveur en standby.

Insérons de nouveau des lignes :

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
INSERT 0 1000000
b1=# SELECT count(*) FROM t2;
  count
---------
 2000000
(1 ligne)
```

Là-aussi, les lignes sont immédiatement transférées. En fait, manuellement, il est difficile de trouver une différence entre les deux serveurs. Ça n'est pas pour autant synchrone. Par exemple, si vous n'avez plus d'espace disque sur le serveur en standby, la requête peut réussir sur le primaire et échouer sur le second. Néanmoins, le résultat est impressionnant :

```
guillaume@laptop:~/tests/maitre/pg_log$ psql -q -p 5433 b1
b1=# SELECT count(*) FROM t2;
  count
---------
 2000000
(1 ligne)
```

Maintenant, essayons d'insérer directement sur le serveur en attente :

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
ERROR:  cannot execute INSERT in a read-only transaction
```

Sur le serveur en standby, la connexion est englobée dans une transaction en lecture seule, assurant qu'aucune modification, de données comme de structure, ne pourra se faire.

Il faut cependant bien comprendre que, malgré tout, beaucoup d'actions sont permises : requêtes d'extraction (SELECT, COPY TO), commandes de curseurs (DECLARE, FETCH, CLOSE), commandes de lecture et de modification des paramètres (SHOW, SET, RESET), commandes de gestion des transactions, etc.

Du coup, il n'est pas possible de modifier les données ou de changer la structure de la base. Il n'est pas possible non plus de créer des tables temporaires car cela nécessite de pouvoir écrire dans les tables systèmes, ce qui est interdit.

Le problème des requêtes en lecture seule

La gestion des verrous dans PostgreSQL est très poussée. Toute requête va nécessiter la pose de verrous. PostgreSQL a l'intelligence de poser les verrous les moins contraignants. Par exemple, sur une requête de lecture d'une table, un verrou sera posé, ce qui peut paraître étonnant de prime abord. Mais pensez par exemple à deux processus connectés au serveur. L'un d'entre eux lit une très grosse table. Un peu plus tard, alors que la requête de lecture est toujours en cours, le deuxième processus essaie de supprimer la table. Le deuxième processus sera bloqué tant que la lecture ne sera pas terminée grâce à un petit verrou (le mode exact est AccessShareLock, donc un verrou en accès partagé, qui n'empêchera ni la lecture ni la modification de données par d'autres processus, mais qui empêchera la suppression de la table). Ce fonctionnement permet à PostgreSQL d'avoir de bonnes performances, y compris avec un grand nombre d'utilisateurs travaillant en même temps.

Dans le cadre de la réplication, cela peut poser quelques soucis. Reprenons l'exemple ci-dessus. Imaginons que le serveur en hotstandby soit en train de lire une très grosse table. Imaginons qu'en même temps, un utilisateur supprime cette table sur le serveur maître. Ce dernier va rapidement dire à l'utilisateur que la table est supprimée si aucun verrou n'a été acquis pour cette table sur le serveur maître. Le « wal sender » va envoyer les informations de la suppression de la table au serveur en hotstandby… et ce dernier va s'apercevoir qu'il doit supprimer la table alors qu'un utilisateur exécute une requête dessus. Attendre la fin de la requête peut être acceptable dans certains cas. Dans d'autres, cela risque d'affecter la pertinence de l'esclave. Les autres enregistrements de transactions ne pourront pas être appliqués tant que celui qui a enregistré la suppression de la table n'aura pas été appliqué. Autrement dit, un bête verrou de lecture d'une table peut complètement bloquer la réplication.

De même, si un utilisateur cherche à supprimer une base de données sur le maître ou à modifier le tablespace d'une base de données alors que des utilisateurs sont connectés à cette base de données sur le serveur en hotstandby, il va être nécessaire de pouvoir les déconnecter pour pouvoir supprimer la base.

Pour contourner ces différents problèmes, les développeurs de PostgreSQL ont ajouté un algorithme dont le but est de s'assurer qu'un verrou ne bloque pas trop longtemps la réplication. Au bout d'un certain temps, cet algorithme annule toute requête qui bloque la transaction. Ce temps est configurable suivant la façon dont l'enregistrement de la transaction est reçue. S'il est reçu à partir d'un journal entier de transactions, il s'agit du paramètre max_standby_archive_delay. S'il est reçu à partir de la réplication en flux, il s'agit du paramètre max_standby_streaming_delay. Les deux valent par défaut 30 secondes, ce qui veut dire que la réplication ne peut pas avoir un lag de plus de 30 secondes à cause d'une requête trop longue. Si la récupération des verrous pour l'exécution d'une requête prend plus que le temps configuré, la requête bloquante est annulée et ce message est envoyé dans les journaux applicatifs :

```
b1=# \d
FATAL:  terminating connection due to conflict with recovery
DÉTAIL : User was holding a relation lock for too long.
```

Évidemment, si plusieurs requêtes bloquent l'enregistrement de la transactions, toutes seront annulées.

En fait, il y aura deux types d'utilisations de ce paramètre :

- le serveur en standby est utilisé comme bascule potentielle, les requêtes en lecture seule sont peu fréquentes sur ce serveur et leur annulation ne pose pas vraiment de problème : dans ce cas, les deux paramètres auront une valeur assez faible pour diminuer le lag de réplication ;
- le serveur en standby est utilisé pour la génération de rapports (par exemple un serveur BI), les requêtes en lecture seule peuvent être longues, le lag ne pose pas de problème pendant la génération du rapport : là, les deux paramètres auront une valeur haute de quelques dizaines de secondes à quelques minutes.

Avoir des paramètres différents pour le délai d'attente avant annulation des requêtes permet d'accélérer le système de réplication quand on se trouve déjà avec du retard. En effet, par défaut, dans le cas du Streaming Replication, PostgreSQL appliquera les enregistrements au fur et à mesure de leur arrivée. Si jamais la réplication commence à accuser un certain retard, PostgreSQL bascule automatiquement dans une application journal de transactions par journal de transaction, ce qui est plus rapide. Dans ce cadre, il est préférable d'avoir un délai très petit pour rattraper plus rapidement le retard.

Basculer l'esclave en maître

Il existe principalement deux termes pour désigner la bascule de l'esclave.

Switchover indique l'exécution de la bascule de l'esclave en maître alors que le maître est toujours actif. Ce dernier va donc prendre le rôle d'esclave. C'est le meilleur des cas car, s'il y a un certain lag, l'opération de bascule doit attendre que les données entre le maître et l'esclave soient synchronisées pour exécuter la bascule (quitte à bloquer les écritures sur le maître le temps de la bascule).

Failover indique que le maître est mort. Là, la bascule doit se faire le plus rapidement possible pour pouvoir rétablir le service. De plus, il est essentiel de s'assurer que le maître ne pourra pas revenir sans avoir été « traité ».

Par un failover

Nous allons simuler une panne du maître en tuant ce dernier avec le signal SIGKILL.

```
guillaume@laptop:~/tests/hotstandby$ ps -ef | grep maitre
1000      7783     1  0 10:58 pts/0    00:00:00 /opt/postgresql-9.0/bin/postgres -D /home/guillaume/tests/maitre
1000      8328  3494  0 11:14 pts/0    00:00:00 grep --color=auto maitre
guillaume@laptop:~/tests/hotstandby$ kill -9 7783
guillaume@laptop:~/tests/hotstandby$ ps -ef | grep maitre
1000      8365  3494  0 11:14 pts/0    00:00:00 grep --color=auto maitre
```

Parfait. Maintenant, tentons une insertion sur l'esclave :

```
guillaume@laptop:~/tests/hotstandby$ psql -q -p 5433 b1
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
ERROR:  cannot execute INSERT in a read-only transaction
```

Ça ne fonctionne toujours pas. Et c'est normal. L'esclave ne se positionne pas de lui-même en maître. Il faut demander à ce qu'il bascule dans ce mode là. Pour cela, nous devons créer le fichier trigger défini par le paramètre trigger_file.

```
guillaume@laptop:~/tests/hotstandby$ touch /tmp/stopstandby
```

Voici la réaction du serveur en standby suite à la création de ce fichier :

```
guillaume@laptop:~/tests/hotstandby$ tail pg_log/postgresql-2010-08-21_110249.log
2010-08-21 11:15:34 CEST LOG:  trigger file found: /tmp/stopstandby
2010-08-21 11:15:34 CEST LOG:  redo done at 0/18AA5368
2010-08-21 11:15:34 CEST LOG:  last completed transaction was at log time 2010-08-21 11:08:25.325852+02
cp: cannot stat `/home/guillaume/tests/archives_xlog/000000010000000000000018': No such file or directory
cp: cannot stat `/home/guillaume/tests/archives_xlog/00000002.history': No such file or directory
2010-08-21 11:15:34 CEST LOG:  selected new timeline ID: 2
cp: cannot stat `/home/guillaume/tests/archives_xlog/00000001.history': No such file or directory
2010-08-21 11:15:35 CEST LOG:  archive recovery complete
2010-08-21 11:15:35 CEST LOG:  database system is ready to accept connections
2010-08-21 11:15:35 CEST LOG:  autovacuum launcher started
```

Les traces indiquent que PostgreSQL a trouvé le fichier trigger défini dans le fichier recovery.conf avec le paramètre trigger_file. Il termine le rejeu, s'assure que le prochain journal de transaction n'est vraiment pas disponible, crée une nouvelle timeline, arrête le mode de restauration, et finalement déclare être prêt à recevoir des connexions en lecture/écriture. Essayons maintenant une requête de modification de données sur l'ancien esclave :

```
guillaume@laptop:~/tests/hotstandby$ psql -q -p 5433 b1
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
```

Parfait. Il est intéressant de savoir que, si une connexion était déjà effective au moment de la bascule, cette connexion passerait elle-aussi du mode lecture seule au mode lecture/écriture automatiquement. Il n'est donc pas nécessaire de déconnecter les sessions en cours pour qu'elles puissent écrire sur le nouveau maître. De plus, il faut prendre en compte deux problèmes potentiels. Tout d'abord, s'il existe un lag entre le maître et l'esclave, les connexions en écriture ne seront possibles qu'à partir du moment où le lag est résorbé. D'autre part, PostgreSQL n'annulera pas les requêtes en lecture seule. Il attendra qu'elles se terminent pour basculer dans le mode maître. Il peut donc y avoir un certain délai entre la création du fichier trigger et la mise à disposition du mode lecteur/écriture sur l'ancien esclave.

Par un switchover

Le switchover est plus complexe car PostgreSQL ne propose aucune automatisation du système. Il y a fort à parier que des outils tiers verront le jour ou seront mis à jour pour intégrer ça. En attendant, il faudra le faire manuellement.

La première chose à faire est de mettre le serveur esclave en mode lecture/écriture. Pour cela, nous passons toujours par le fichier trigger :

```
guillaume@laptop:~/tests/hotstandby$ touch /tmp/stopstandby
```

Les traces sont identiques à celles indiquées pour le failover. Ensuite, nous arrêtons proprement le maître.

```
guillaume@laptop:~/tests/hotstandby$ pg_ctl -D /home/guillaume/tests/maitre stop
en attente de l'arrêt du serveur.... effectué
serveur arrêté
```

Pour que l'ancien maître soit le nouvel esclave, il va falloir le reconstruire. Commençons déjà par modifier la configuration du nouveau maître :

```
guillaume@laptop:~/tests/hotstandby$ cat > hotstandby.conf <<_EOF_
> port = 5433
> wal_level = 'hot_standby'
> archive_mode = on
> archive_command = 'cp %p /home/guillaume/tests/archives_xlog/%f'
> max_wal_senders = 1
> _EOF_
```

Notez que le répertoire d'archivage n'a pas changé, le nom des fichiers sera différent grâce à la nouvelle timeline sélectionné lors de la bascule en maître. Profitons-en aussi pour modifier le fichier de configuration des accès distants, le fichier pg_hba.conf, en lui ajoutant la ligne suivante :

```
host    replication     all             127.0.0.1/32            md5
```

Nous pouvons enfin redémarrer le nouveau maître :

```
guillaume@laptop:~/tests/hotstandby$ pg_ctl -D /home/guillaume/tests/hotstandby restart
en attente de l'arrêt du serveur.... effectué
serveur arrêté
serveur en cours de démarrage
guillaume@laptop:~/tests/hotstandby$ cd pg_log
guillaume@laptop:~/tests/hotstandby/pg_log$ ll | tail -1
-rw------- 1 guillaume guillaume  226 2010-08-21 11:23 postgresql-2010-08-21_112345.log
guillaume@laptop:~/tests/hotstandby/pg_log$ cat postgresql-2010-08-21_112345.log
2010-08-21 11:23:45 CEST LOG:  database system was shut down at 2010-08-21 11:23:44 CEST
2010-08-21 11:23:45 CEST LOG:  autovacuum launcher started
2010-08-21 11:23:45 CEST LOG:  database system is ready to accept connections
```

Tout va bien. Occupons-nous maintenant de l'ancien maître. Il faut le reconstruire à partir d'une sauvegarde du base du nouveau maître. C'est parti :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ cd ../..
gguillaume@laptop:~/tests$ psql -p 5433 -c "SELECT pg_start_backup('laptop_20100821_1124', true)" postgres
 pg_start_backup
-----------------
 0/1D000020
(1 ligne)
```

```
guillaume@laptop:~/tests$ tar cfj laptop_20100821_1124.tar.bz2 hotstandby
guillaume@laptop:~/tests$ psql -p 5433 -c "SELECT pg_stop_backup()" postgres
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/1D0000D8
(1 ligne)

guillaume@laptop:~/tests$ rm -r maitre/
guillaume@laptop:~/tests$ tar --transform "s#^hotstandby#maitre#" -xjf laptop_20100821_1124.tar.bz2
```

La reconfiguration n'est pas beaucoup plus compliquée. Le fichier de configuration recovery.conf est exactement le même que l'ancien à un détail près. Une fois utilisé la première fois, il a été renommé en recovery.done. Il nous suffit donc de le renommer en recovery.conf et de modifier le paramètre primary_conninfo. En effet, la connexion doit se faire sur le serveur écoutant le port 5433. Il nous faut préciser ce port. Quant au fichier de configuration maitre.conf, nous allons simplement ajouter le paramètre hot_standby pour que la connexion en lecture seule soit possible.

```
guillaume@laptop:~/tests$ cd maitre
guillaume@laptop:~/tests/maitre$ sed -i -e 's/hotstandby.conf/maitre.conf/' postgresql.conf
guillaume@laptop:~/tests/maitre$ rm hotstandby.conf
guillaume@laptop:~/tests/maitre$ mv recovery.done recovery.conf
guillaume@laptop:~/tests/maitre$ sed -i -e 's/port=5432/port=5433/' recovery.conf
guillaume@laptop:~/tests/maitre$ cat > maitre.conf <<_EOF_
> hot_standby = on
> _EOF_
```

La connexion au nouveau maître, donc via le port 5433, doit aussi pouvoir se faire sans saisir de mot de passe. Nous allons de nouveau modifier le fichier .pgpass :

```
guillaume@laptop:~/tests/maitre$ cat >> ~/.pgpass << _EOF_
> 127.0.0.1:5433:replication:guillaume:supersecret
> _EOF_
```

Il ne faut pas oublier de faire un petit ménage dans les fichiers restaurés :

```
guillaume@laptop:~/tests/maitre$ rm postmaster.pid pg_xlog/* pg_log/*
rm: impossible de supprimer «pg_xlog/archive_status»: est un dossier
```

Il ne nous reste plus qu'à démarrer l'ancien maître en tant que nouvel esclave.

```
guillaume@laptop:~/tests/maitre$ pg_ctl -D /home/guillaume/tests/maitre start
serveur en cours de démarrage
guillaume@laptop:~/tests/maitre$ cd pg_log
-rw------- 1 guillaume guillaume 746 2010-08-21 11:28 postgresql-2010-08-21_112809.log
guillaume@laptop:~/tests/maitre/pg_log$ cat postgresql-2010-08-21_112809.log
2010-08-21 11:28:09 CEST LOG:  database system was interrupted; last known up at 2010-08-21 11:24:54 CEST
2010-08-21 11:28:09 CEST LOG:  restored log file "00000002.history" from archive
2010-08-21 11:28:09 CEST LOG:  entering standby mode
2010-08-21 11:28:09 CEST LOG:  restored log file "0000000200000000000000001D" from archive
2010-08-21 11:28:09 CEST LOG:  redo starts at 0/1D000020
2010-08-21 11:28:09 CEST LOG:  consistent recovery state reached at 0/1E000000
2010-08-21 11:28:09 CEST LOG:  database system is ready to accept read only connections
cp: cannot stat `/home/guillaume/tests/archives_xlog/0000000200000000000000001E': No such file or directory
2010-08-21 11:28:09 CEST LOG:  streaming replication successfully connected to primary
```

Vérifions que les modifications sur le nouveau maître arrive bien sur le nouvel esclave :

```
guillaume@laptop:~/tests/maitre/pg_log$ createdb -p 5433 b4
guillaume@laptop:~/tests/maitre/pg_log$ psql -l | grep b4
 b4        | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
guillaume@laptop:~/tests/maitre/pg_log$ psql -l -p 5433 | grep b4
 b4        | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
```

Mission accomplie !

Comment surveiller les deux serveurs

La surveillance se passe principalement au niveau des journaux applicatifs. L'affichage de ps affiche aussi des informations sur les transactions en cours de transfert. Néanmoins, il existe quelques procédures stockées permettant d'avoir des informations plus directes.

La première s'appelle pg_is_in_recovery(). Elle permet de savoir si la session en cours d'exécution se fait sur un serveur maître (auquel cas cette fonction renvoie la valeur booléenne FALSE) ou sur un serveur en hotstandby (la valeur TRUE est renvoyée).

Un autre moyen de savoir si la session est en lecture seule revient à interroger le paramètre transaction_read_only. Ce dernier renvoie TRUE si vous êtes connecté au serveur en hotstandby. Contrairement au paramètre default_transaction_read_only, ce nouveau paramètre est seulement disponible en lecture. Il n'est donc pas possible de modifier manuellement sa valeur.

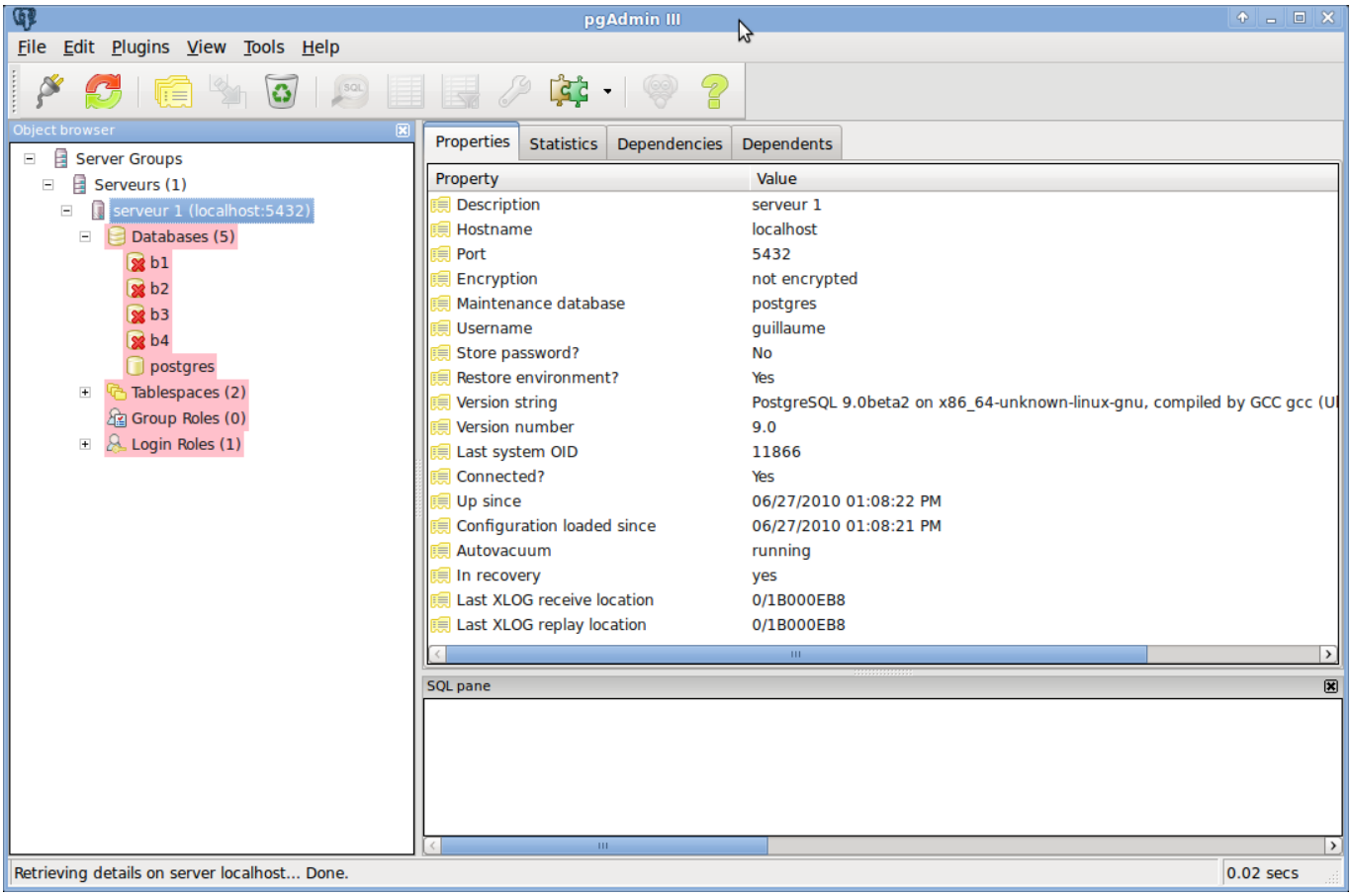
La procédure stockée pg_last_xlog_receive_location() indique l'emplacement du dernier enregistrement de transaction reçu, alors que pg_last_xlog_replay_location() précise l'emplacement du dernier enregistrement rejoué. La différence entre ces deux informations permet de connaître le lag sur l'enregistrement. Plus ce dernier est important, plus la bascule causée par la création du fichier trigger sera longue à exécuter (car il y aura plus d'enregistrements à copier).

Ces deux informations sont aussi intéressantes à comparer à l'emplacement de l'enregistrement renvoyé par l'exécution sur le maître de la procédure stockée pg_current_xlog_location(). En effet, la différence entre cet emplacement et celui du dernier enregistrement reçu (pg_last_xlog_receive_location()) indique le lag de transmission entre le serveur maître et le serveur en hotstandby, alors que la différence entre l'emplacement du pg_current_xlog_location() et celui du pg_last_xlog_replay_location() indique le lag réel (c'est-à-dire au niveau disque) entre serveur maître et le serveur en hotstandby.

Le lag de transmission est très gênant car il représente les informations modifiées sur le serveur maître mais non transmise au serveur esclave. Autrement dit, en cas de crash du serveur maître, vous aurez perdu toutes ces informations.

Le lag d'écriture est moins gênant. Il n'y aura pas de perte de données mais la bascule sera moins rapide.

L'outil d'administration pgAdmin, en version 1.12, affiche les informations en question :



À quoi doit-on s'attendre dans le futur

Il est clair que cette double fonctionnalité (esclave en lecture seule, et réplication en flux) est une excellente nouvelle pour les utilisateurs de PostgreSQL. Il est tout aussi clair que beaucoup de choses reste à faire. Une réplication synchrone fait partie de ces améliorations que nous pouvons attendre de la future 9.1. D'autres améliorations sont en discussion. Par exemple, autoriser une bascule immédiate du serveur serait un plus. Permettre le transfert de la sauvegarde de base par un processus PostgreSQL en serait un autre. Bref, les développeurs de PostgreSQL ne se croisent pas les bras maintenant que la réplication est intégrée. Des améliorations vont arriver dès la version 9.1.

Les développeurs d'outils tiers ne sont pas en reste. Par exemple, Tatsuo Ishii travaille actuellement beaucoup sur l'intégration de pgPool-II avec le Hot Standby. La version 3 de pgPool-II, qui vient de sortir en beta 3, permettra d'utiliser sa fonctionnalité de répartition de charge entre un serveur maître et un serveur en hotstandby.

Il ne faut pas croire pour autant que les autres outils de réplication deviennent obsolètes pour autant. Prenez Slony, Londiste ou Bucardo, ils ont encore tous de beaux jours devant eux. Il se passera du temps avant qu'il soit possible d'obtenir la granularité de ces systèmes de réplication avec la réplication en flux de PostgreSQL. De plus, ces outils permettent une bascule de type switchover beaucoup plus simplement que ce que ne permet la réplication en flux.