−Table des matières

Mise en place de la réplication avec PostgreSQL 9.0 - 1/2

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 131 (Octobre 2010). Il est disponible maintenant sous licence Creative Commons.

L'évolution majeure qu'apporte la version 9.0 de PostgreSQL est la réplication. Évidemment, ce n'est pas la seule nouveauté intéressante de cette version mais c'est certainement celle qui fera le plus parler de cette version, celle aussi qui aidera à une adoption plus massive de cet excellent système de gestion de bases de données. Avant de tester cette nouvelle fonctionnalité, prenons un peu de temps pour la définir. Les solutions de réplication sont très nombreuses et portent généralement sur des fonctionnalités différentes, le point en commun étant que les données modifiées sur un serveur sont transférées automatiquement sur un ou plusieurs autres serveurs. Cependant, chaque solution a ses particularités.

La première particularité est de savoir s'il est possible d'écrire sur tous les serveurs ou seulement sur un serveur, généralement appelé serveur maître ou serveur primaire. Le premier cas (écriture possible sur tous les serveurs) semble évidemment le plus intéressant mais pose des soucis de verrous. Quand un utilisateur veut écrire des données sur un objet du serveur A, le même objet ne doit pas pouvoir être modifié sur le ou les autres serveurs avant que les écritures du serveur A ne soient synchronisées sur les autres. Avoir plus de verrous a une implication au niveau des performances. La dénomination habituelle d'une réplication où seul un serveur peut faire les écritures est « réplication maître/esclave » alors que la dénomination d'une réplication où tous les serveurs sont disponibles en écriture est « réplication maître/maître ». En ce qui concerne PostgreSQL, la réplication maître/maître est très rare. Le seul projet libre capable de le faire, à ma connaissance, est Bucardo.

La deuxième particularité concerne le côté synchrone ou non de la réplication. Une réplication synchrone fonctionne de la façon suivante : une transaction n'est pas considérée validée (pas de COMMIT) tant que tous les serveurs n'ont pas validé cette transaction. Il y a plusieurs intérêts à ça :

- il n'y a pas de risque de perte de données au moment d'une bascule du maître ;
- le résultat d'une requête est cohérent quelque soit le serveur interrogé dans le cadre d'un cluster en répartition de charge.

Le soucis majeur de ce type de réplication est les performances (ou plutôt les contre-performances). En effet, il faut bien comprendre que pour chaque transaction modifiant des données, il ne suffit pas qu'un serveur enregistre cette information. Il faut que tous l'aient enregistré. Cela va ajouter une surcharge plus ou moins importante suivant la solution choisie. Des solutions asynchrones existent. Elles sont généralement plus performantes. Le serveur qui fait l'écriture renvoie au client le fait que l'enregistrement s'est bien passé et se charge un peu après d'envoyer l'information sur les données modifiées aux autres serveurs. En cas d'arrêt brutal (crash) du serveur maître, les données peuvent ne pas avoir été complètement synchronisées avec les serveurs esclaves. Cela occasionne une perte de données, généralement faible mais toujours très désagréable. Problème peut-être pire, la répartition de charge. Dans ce cadre, il est tout à fait possible que la même requête exécutée sur deux serveurs différents ne renvoie pas les mêmes résultats suivant le serveur sur laquelle elle est exécutée. Tout dépend évidemment du délai dans la synchronisation avec les autres serveurs. Là-aussi, il existe assez peu de solutions synchrones pour PostgreSQL. La seule, toujours à ma connaissance, n'est pas spécifique à PostgreSQL : il s'agit de DRBD. Pour plus d'informations sur ce type de réplication, n'hésitez pas à lire l'article consacré à ce sujet dans le hors-série 45 de GNU/Linux Magazine France (« Retours d'expériences pour sysadmins, 10 solutions concrètes »).

La réplication proposée par PostgreSQL est une réplication asynchrone, en maître/esclave (plus exactement, un maître et un ou plusieurs esclaves). Le serveur maître est aussi appelé serveur primaire. Les serveurs esclaves sont appelés des serveur standby (ou en attente). Il existe deux types de serveurs standby : les serveurs warmstandby et les serveurs hotstandby. Cette nouvelle particularité permet d'expliciter si le serveur esclave est disponible en lecture seule (dans ce cas, il s'agit du hotstandby) ou non (ici, du warmstandby). « Warm Standby » est un terme déjà connu des utilisateurs de PostgreSQL car c'est une capacité qui a été intégrée à PostgreSQL à partir de la version 8.2. Depuis cette version, il est possible de répliquer une instance PostgreSQL à partir de ses journaux de transactions. Le serveur esclave ne peut que rejouer les journaux de transactions au fur et à mesure qu'il lui arrive. Un outil comme pg_standby a permis de faciliter ce travail dès la version 8.3. Mais il s'agit bien d'un serveur warmstandby, c'est-à-dire un serveur où il est impossible de se connecter, y compris pour y exécuter des requêtes en lecture seule. Ça limite l'intérêt de cette réplication. Heureusement, la version 9.0 va lever cette barrière en acceptant des serveurs en « Hot Standby ». Il est donc possible, avec la bonne configuration, d'accéder au serveur hotstandby en lecture seule. Il existe évidemment quelques limitations qu'on abordera en fin d'article. La version 9.0 ne se limite pas seulement à cette fonctionnalité pour la réplication. Elle ajoute en plus une technologie appelée « Streaming Replication ». Cette technologie permet d'envoyer les données modifiées, non plus journal de transactions par journal de transactions, mais groupe de transactions par groupe de transactions. Cela permet une granularité plus fine dans la réplication, et du coup un lag moins important. Ce premier article d'une série consacrée aux nouveautés de la version 9.0 se concentrera sur la mise en place des esclaves en lecture seule dans PostgreSQL. Un second article abordera la question de la réplication en flux.

Quelques prérequis

Les pré-requis n'ont pas changé par rapport aux anciennes versions de PostgreSQL.

Étant donné que le format des journaux de transactions peut évoluer d'une version majeure à une autre, il n'est pas possible de répliquer les données de ou vers une autre version majeure. Par contre, lorsqu'une version 9.0.1 sortira, il sera possible de répliquer les données d'une version 9.0.0 vers une version 9.0.1, et inversement. Autrement dit, la réplication interne de PostgreSQL est possible entre des versions mineures de la même version majeure (ici 9.0), mais pas entre des versions majeures différentes.

Il n'est pas nécessaire d'utiliser le même matériel. C'est souvent préférable pour que, lorsqu'une bascule se fera, il n'y ait pas de différence au niveau des performances. Cependant, il est possible d'avoir du matériel différent. Par contre, il est essentiel que la plateforme soit la même : pas de mixage entre des serveurs 32 bits et des serveurs 64 bits, pas de mixage non plus dans la gestion des octets (little endian / big endian).

Préparation du maître

Tous les tests que nous allons effectués ci-dessous sont fait à partir de mon portable, sur lequel est installée une Xubuntu 10.04, avec un PostgreSQL 9.0 provenant du dépôt CVS de PostgreSQL. Plus exactement, il s'agit d'une version postérieure à la beta 4.

Voici comment j'ai préparé le serveur maître :

```
guillaume@laptop:~$ mkdir tests
guillaume@laptop:~$ initdb -D /home/guillaume/tests/maitre
Les fichiers de ce cluster appartiendront à l'utilisateur « guillaume ».
Le processus serveur doit également lui appartenir.
[… tout un ensemble de messages provenant de la commande initdb …]

Succès. Vous pouvez maintenant lancer le serveur de bases de données par :

    postgres -D /home/guillaume/tests/maitre
ou
    pg_ctl -D /home/guillaume/tests/maitre -l journal_applicatif start
```

Mon instance maître se trouve donc dans le répertoire /home/guillaume/tests/maitre. La configuration est un peu modifiée avec ces deux commandes :

```
guillaume@laptop:~$ cd tests/maitre
guillaume@laptop:~/tests/maitre$ cat >> postgresql.conf << _EOF_
> include 'communs.conf'
> _EOF_
guillaume@laptop:~/tests/maitre$ cat >> communs.conf << _EOF_
> logging_collector = on
> log_line_prefix = '%t '
> lc_messages = 'C'
> _EOF_
```

Autrement dit, au démarrage, PostgreSQL va lire le fichier postgresql.conf, qui ne comprend qu'une seule modification, une demande d'import du fichier communs.conf où est stocké un paramétrage spécifique à tous mes serveurs PostgreSQL. Ce paramétrage ne modifie pour l'instant que trois paramètres :

- logging_collector est activé pour que PostgreSQL gère lui-même les fichiers relatifs aux traces ;
- log_line_prefix est initialisé à '%t ' pour que chaque trace précise l'horodatage de cette trace ;
- enfin, lc_messages est initialisé à 'C' pour que les traces soient enregistrées en anglais

Le français, c'est bien. Je ne vais pas dire le contraire alors que je maintiens la traduction des applications et du manuel. Malheureusement, ça complique les recherches sur un moteur de recherche ainsi que les discussions avec des développeurs connaissant en majorité l'anglais et non pas le français. Et c'est sans même parler des outils d'analyse des traces qui, eux, ne connaissent que l'anglais. Bref, les traces doivent toujours être en anglais, surtout sur un serveur en production.

Mise en place de l'archivage des journaux de transactions

Cette étape est toujours présente en 9.0. Il y a une légère différence avec les versions précédentes avec l'arrivée d'un nouveau paramètre appelé wal_level.

Ce paramètre permet d'indiquer la quantité d'informations contenues dans les journaux de transactions. Il accepte trois valeurs :

- « minimal ». Avec cette configuration (qui se trouve être celle par défaut), certaines opérations de masse, comme la création d'index ou un COPY sur une table créée dans la même transaction, peuvent ignorer la journalisation de la transaction. Cela a pour effet d'améliorer grandement les performances de ces commandes. Cependant, dans ce cas, les journaux de transactions ne contiennent pas suffisamment d'informations pour pouvoir exploiter un archivage. Ce dernier est donc impossible à faire avec cette configuration.
- « archive ». Ce mode permet de s'assurer que l'archivage des journaux de transactions soit possible dans le cadre d'une restauration. Toutes les modifications sur les fichiers de données sont d'abord enregistrées dans les journaux de transactions.
- « hot_standby ». Des informations supplémentaires sont tracées dans les journaux de transactions pour pouvoir exécuter des requêtes en lecture seule sur les serveurs standby.

Évidemment, plus le nombre d'informations enregistrées dans les journaux de transactions est important, plus les performances s'en ressentiront. Sur les tests effectués, la différence de performance n'était pas distinguable entre archive et hot_standby.

Pour mettre en place l'archivage, il nous faut donc modifier la valeur du paramètre wal_level pour le passer à « archive ». De plus, pour éviter de polluer le fichier de configuration communs.conf, nous allons en ajouter un troisième appelé, maitre.conf :

```
guillaume@laptop:~/tests/maitre$ cat >> postgresql.conf << _EOF_
> include 'maitre.conf'
> _EOF_
guillaume@laptop:~/tests/maitre$ cat > maitre.conf << _EOF_
> wal_level = 'archive'
```

```
> _EOF_
```

Le reste de la configuration est identique à celle d'un archivage pour une version 8.4. Nous allons modifier la valeur des paramètres archive_mode et archive_command.

```
guillaume@laptop:~/tests/maitre$ cat >> maitre.conf << _EOF_
> archive_mode = on
> archive_command = 'cp %p /home/guillaume/tests/archives_xlog/%f'
> _EOF_
```

Nous aurions pu modifier aussi archive_timeout mais, dans le cadre de cet article, cela a peut d'intérêt.

La configuration du paramètre archive_command demande à ce que le répertoire soit déjà disponible. Nous allons le créer :

```
guillaume@laptop:~/tests/maitre$ mkdir /home/guillaume/tests/archives_xlog
```

D'ordinaire, il faudrait s'assurer que l'utilisateur qui lance le serveur PostgreSQL ait des droits d'écriture dans ce répertoire. Là, étant donné que l'utilisateur qui a créé le répertoire est aussi celui qui va lancer PostgreSQL, il n'y aura pas de problème de droits.

Maintenant, nous pouvons démarrer le serveur maître, histoire de vérifier que l'archivage fonctionne bien. Nous allons créer une base, y ajouter une table, dans laquelle nous allons insérer un million de lignes pour que des journaux de transactions soient archivables.

```
guillaume@laptop:~/tests/maitre$ pg_ctl -D /home/guillaume/tests/maitre start
serveur en cours de démarrage
guillaume@laptop:~/tests/maitre$ createdb b1
guillaume@laptop:~/tests/maitre$ psql b1
psql (9.0beta4)
Saisissez « help » pour l'aide.

b1=# CREATE TABLE t1 (c1 integer);
CREATE TABLE
b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
INSERT 0 1000000
b1=# \q
```

Regardons ce qui se trouve dans le répertoire d'archivage :

```
guillaume@laptop:~/tests/maitre$ ll ../archives_xlog
total 65648
-rw------- 1 guillaume guillaume 16777216 2010-08-21 10:32 000000010000000000000000
-rw------- 1 guillaume guillaume 16777216 2010-08-21 10:32 000000010000000000000001
-rw------- 1 guillaume guillaume 16777216 2010-08-21 10:32 000000010000000000000002
-rw------- 1 guillaume guillaume 16777216 2010-08-21 10:33 000000010000000000000003
```

Quatre journaux de transactions sont déjà archivés, preuve du bon paramétrage de PostgreSQL.

Créer la sauvegarde de base

Pour cette partie, rien n'a vraiment changé. On commence par un appel à la procédure système pg_start_backup(), on sauvegarde le répertoire des données de PostgreSQL (ainsi que les tablespaces s'il y en a) et, une fois la sauvegarde terminée, on fait appel à la procédure système pg_stop_backup().

```
guillaume@laptop:~/tests/maitre$ psql -c "SELECT pg_start_backup('laptop_20100821_1033', true)" postgres
 pg_start_backup
-----------------
 0/5000020
(1 ligne)

guillaume@laptop:~/tests/maitre$ cd ..
guillaume@laptop:~/tests$ tar cfj laptop_20100821_1033 .tar.bz2 maitre
guillaume@laptop:~/tests$ psql -c "SELECT pg_stop_backup();" postgres
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/5004408
(1 ligne)
```

Le message de niveau NOTICE est nouveau en 9.0. Il précise simplement que la procédure s'est terminée correctement et que tous les journaux de transactions nécessaires ont été archivés. C'est une mesure de précaution au cas où la commande indiquée par le paramètre archive_command n'arrivait pas à archiver les journaux. Là, vous êtes prévenu que tout a bien fonctionné. Maintenant que la sauvegarde est terminée, nous allons pouvoir mettre en place un serveur en Warm Standby.

Préparation d'un serveur Warm Standby

Le répertoire des données de ce nouveau serveur sera sur /home/guillaume/tests/warmstandby. Pour l'instant, ce répertoire n'existe pas. Nous allons le créer en restaurant la sauvegarde du chapitre ci-dessus :

```
guillaume@laptop:~/tests$ tar --transform "s#^maitre#warmstandby#" -xjf laptop_20100821_1033 .tar.bz2
```

Je viens de découvrir une option très intéressante de tar, l'option –transform. Cette option permet de transformer le nom des fichiers créés lors d'une extraction de l'archive grâce à une expression rationnelle, style sed, indiquée après. Ainsi, la commande ci-dessous remplace le nom du répertoire « maitre » par « warmstandby ». Je ne connaissais pas cette option auparavant, il est donc très probable qu'elle n'existe que sur les dernières versions de GNU tar.

Une fois la sauvegarde restaurée, nous voilà avec un répertoire warmstandby contenant l'équivalent du répertoire maitre lors de sa sauvegarde. Certains fichiers doivent être supprimés avant d'aller plus loin :

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

Le fichier postmaster.pid est effacé pour éviter un message d'avertissement dans les traces de PostgreSQL. Ça ne devrait pas l'empêcher de démarrer car, malgré l'existence de ce fichier, il tentera quand même de se lancer mais, dans le meilleur des cas, cela affichera un message inintéressant dans les traces. Autant s'en débarrasser tout de suite. Le répertoire pg_xlog contient des journaux de transactions obsolètes du maître. Il faut donc les supprimer. De toute façon, au démarrage du serveur en warmstandby, ce dernier ira récupérer les journaux de transactions dans le répertoire d'archivage. Les fichiers stockés dans pg_log sont les traces du maître. Leur suppression permet de s'assurer que nous ne trouverons que les messages du serveur en warmstandby dans ce répertoire.

Le message d'avertissement de rm est normal. archive_status est un répertoire, il ne peut pas être supprimé par la commande lancée et c'est très bien ainsi car nous ne voulons pas le supprimer.

La configuration de PostgreSQL doit être modifiée. Actuellement, comme nous avons copié tout les répertoire des données de PostgreSQL du serveur maître, nous nous retrouvons aussi avec sa configuration. Nous devons tout d'abord supprimer la configuration spécifique du maître :

```
guillaume@laptop:~/tests/warmstandby$ rm maitre.conf
```

Ensuite, nous allons remplacer l'import de cet ancien fichier de configuration par un fichier que nous allons créer ensuite :

```
guillaume@laptop:~/tests/warmstandby$ sed -i -e 's/maitre.conf/warmstandby.conf/' postgresql.conf
```

Le nouveau fichier de configuration, appelé warmstandby.conf, sera beaucoup plus léger. Nous ne réactivons pas l'archivage. Nous allons laisser le paramètre wal_level à sa configuration par défaut, vu que nous n'avons pas réactivé l'archivage des journaux de transactions du serveur en standby. En fait, le seul paramètre à modifier est le numéro du port. En effet, le port doit être changé car j'exécute les deux serveurs sur la même machine. C'est évidemment une configuration qui n'a aucun intérêt si vos deux serveurs PostgreSQL sont sur des machines différentes (même si ce sont des machines virtuelles).

Nous allons donc placer cette configuration dans le nouveau fichier :

```
guillaume@laptop:~/tests/warmstandby$ cat > warmstandby.conf << _EOF_
> port = 5433
> _EOF_
```

Nous devons maintenant créer le fichier de configuration de la restauration. Nous pourrions copier le fichier d'exemple (qui se trouve dans le répertoire « share » de PostgreSQL, sous le nom de recovery.conf.sample). Néanmoins, dans le cadre de cet article, nous allons plutôt le créer directement. Nous n'avons qu'un paramètre à configurer, restore_command. Cette commande le fera simplement :

```
guillaume@laptop:~/tests/warmstandby$ cat > recovery.conf << _EOF_
> restore_command = 'pg_standby -d -t /tmp/stopstandby /home/guillaume/tests/archives_xlog %f %p %r >> /home/guillaume/tests/pg_standby.log 2>&1'
> _EOF_
```

Attention, à l'impression de l'article, il se peut que le contenu de restore_command soit affiché sur plusieurs lignes. Il ne faut pas s'y tromper, la valeur doit être sur une seule ligne. pg_standby est un outil provenant des modules contrib. Il fait partie de PostgreSQL depuis la version 8.3. Néanmoins, étant un module contrib, vous devrez l'installer en plus du moteur. Si vous utilisez le système de paquets de votre distribution, vous devez avoir un paquet dont le nom contient postgresql et contrib pour la version 9.0 (par exemple, postgresql-contrib ou postgresql-contrib-9.0) . Installez-le. Pour ceux qui ont compilé leur version de PostgreSQL, il leur faudra aller dans le sous-répertoire contrib/pg_standby du répertoire des sources et compiler/installer ce module contrib (la double commande « make && make install » suffit). Cet outil doit se trouver dans le PATH. Si ce n'est pas le cas, l'exécution de la commande indiquée par le paramètre restore_command échouera, provoquant la sortie du mode restauration du serveur PostgreSQL

Nous pouvons enfin lancer le serveur standby.

```
guillaume@laptop:~/tests/warmstandby$ pg_ctl -D /home/guillaume/tests/warmstandby start
serveur en cours de démarrage
```

En se positionnant dans le répertoire des traces de PostgreSQL, il est possible de lire les traces du démarrage :

```
guillaume@laptop:~/tests/warmstandby$ cd pg_log
guillaume@laptop:~/tests/warmstandby/pg_log$ ll
total 12
-rw------- 1 guillaume guillaume 386 2010-08-21 10:40 postgresql-2010-08-21_104049.log
guillaume@laptop:~/tests/warmstandby/pg_log$ cat postgresql-2010-08-21_104049.log
2010-08-21 10:40:49 CEST LOG:  database system was interrupted; last known up at 2010-08-21 10:34:04 CEST
2010-08-21 10:40:49 CEST LOG:  starting archive recovery
2010-08-21 10:40:49 CEST LOG:  restored log file "000000010000000000000005" from archive
2010-08-21 10:40:50 CEST LOG:  redo starts at 0/5000078
2010-08-21 10:40:50 CEST LOG:  consistent recovery state reached at 0/6000000
```

Jusqu'ici, tout va bien. Le serveur a démarré en remarquant que le système avait été interrompu, ce qui est logique car notre sauvegarde a été effectuée alors que le serveur était toujours en cours d'exécution. Il détecte le fichier et commence la procédure de restauration (« starting archive recovery »). Il va donc restaurer les différents journaux de transactions déjà archivés. Ceci fait, il va rapidement être en attente des journaux de transactions du serveur maître.

Ajoutons un peu d'activité sur le serveur maître :

```
guillaume@laptop:~/tests/warmstandby/pg_log$ psql -q b1
b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
b1=# \q
```

De nouveaux journaux de transactions apparaissent dans le répertoire d'archivage. Ils sont aussi traités par pg_standby, puis par le serveur PostgreSQL en standby, comme l'indique les traces ci-dessous :

```
2010-08-21 10:42:05 CEST LOG:  restored log file "000000010000000000000006" from archive
2010-08-21 10:42:06 CEST LOG:  restored log file "000000010000000000000007" from archive
```

```
2010-08-21 10:42:07 CEST LOG:  restored log file "000000010000000000000008" from archive
```

Ça fonctionne, il n'y a aucun soucis. En fait, nous nous retrouvons exactement dans le cas d'un serveur 8.4 en standby. La seule différence se trouve dans la configuration du nouveau paramètre wal_level. De plus, il restaure toujours que journal de transactions par journal de transactions, et les connexions sont interdites sur le serveur en standby comme le montre le test suivant :

```
guillaume@laptop:~/tests/warmstandby/pg_log$ psql -p 5433 b1
psql: FATAL:  the database system is starting up
```

L'ajout de l'option « -p 5433 » me permet d'accéder au serveur en standby car ce dernier a une configuration du port spécifique d'après notre configuration dans le fichier warmstandby.conf.

Nous allons donc voir maintenant comment s'assurer que l'accès en lecture seule soit possible sur le serveur en standby.

Préparation d'un Hot Standby

Commençons par arrêter le serveur en Warm Standby.

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

La préparation d'un serveur Hot Standby demande la réalisation de plusieurs étapes. Mais tout d'abord, nous allons devoir modifier la valeur du paramètre wal_level. En effet, pour que le serveur en standby accepte des requêtes en lecture seule, il a besoin de récupérer plus d'informations que ce que fournit le niveau « archive » du paramètre wal_level.

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

Remarquez que nous conservons le paramétrage de l'archivage. Pour que le serveur maître prenne en compte la nouvelle configuration, nous devons le redémarrer (un rechargement de la configuration ne suffira pas, ce paramètre n'est pris en compte qu'au démarrage).

```
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
```

Nous devons ensuite faire une nouvelle sauvegarde des fichiers, qui nous servira de base pour créer le serveur Hot Standby.

```
guillaume@laptop:~/tests/maitre$ psql -c "SELECT pg_start_backup('laptop_20100821_1045', true)" postgres
 pg_start_backup
-----------------
 0/B000020
(1 ligne)

guillaume@laptop:~/tests/maitre$ cd ..
guillaume@laptop:~/tests$ tar cfj laptop_20100821_1045.tar.bz2 maitre
guillaume@laptop:~/tests$ psql -c "SELECT pg_stop_backup();" postgres
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/B0000D8
(1 ligne)
```

Restaurons cette sauvegarde de fichiers dans un répertoire hotstandby :

```
guillaume@laptop:~/tests$ tar --transform "s#^maitre#hotstandby#" -xjf laptop_20100821_1045.tar.bz2
```

Il nous faut de nouveau supprimer les fichiers inutiles de ce répertoire ainsi que le fichier de configuration du maître :

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

Ensuite, nous allons remplacer l'import de cet ancien fichier de configuration par un fichier que nous allons créer ensuite :

```
guillaume@laptop:~/tests/hotstandby$ sed -i -e 's/maitre.conf/hotstandby.conf/'  postgresql.conf
```

Le nouveau fichier de configuration, appelé hotstandby.conf, sera beaucoup plus léger. Nous n'allons toujours pas réactiver l'archivage, nous allons toujours laisser le paramètre wal_level à sa configuration par défaut et le numéro du port restera modifié. Mais cette fois, ça ne sera pas le seul paramètre à être modifié. Nous allons indiquer que ce serveur est un serveur Hot Standby. Pour cela, nous devons activer (mettre à « on ») le paramètre hot_standby. Cela nous donne cette configuration :

```
guillaume@laptop:~/tests/hotstandby$ cat > hotstandby.conf << _EOF_
> port = 5433
> hot_standby = on
> _EOF_
```

Le fichier de restauration, le recovery.conf, est identique à celui du Warm Standby. Nous allons donc le copier du répertoire warmstandby :

```
guillaume@laptop:~/tests/hotstandby$ cp ../warmstandby/recovery.conf .
```

Nous allons enfin pouvoir démarrer le serveur hotstandby :

```
guillaume@laptop:~/tests/hotstandby$ pg_ctl -D /home/guillaume/tests/hotstandby start
```

Voyons ce que nous raconte les traces :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ ll
total 12
-rw------- 1 guillaume guillaume 474 2010-08-21 10:49 postgresql-2010-08-21_104916.log
guillaume@laptop:~/tests/hotstandby/pg_log$ cat postgresql-2010-08-21_104916.log
2010-08-21 10:49:16 CEST LOG:  database system was interrupted; last known up at 2010-08-21 10:45:49 CEST
2010-08-21 10:49:16 CEST LOG:  starting archive recovery
2010-08-21 10:49:17 CEST LOG:  restored log file "00000001000000000000000B" from archive
2010-08-21 10:49:17 CEST LOG:  redo starts at 0/B000020
2010-08-21 10:49:17 CEST LOG:  consistent recovery state reached at 0/C000000
2010-08-21 10:49:17 CEST LOG:  database system is ready to accept read only connections
```

Exactement la même chose que pour le Hot Standby, en dehors de cette dernière phrase. Autrement dit, il nous indique que le système a été interrompu. Comme pour le Warm Standby, c'est tout à fait normal vu que la sauvegarde s'est produite alors que le serveur maître était en cours d'exécution. Il commence son processus de restauration et récupère tous les journaux déjà disponibles dans l'archivage. Une fois qu'il a terminé cette première partie de la restauration, il précise que le système est prêt à recevoir des connexions en lecture seule. Vérifions cela.

La commande psql nous renvoie la liste des bases de données avec l'option -l. C'est un bon moyen de savoir si on peut se connecter à une base et exécuter quelques requêtes en lecture seule.

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -l
                             Liste des bases de données
   Nom    | Propriétaire | Encodage |     Tri     | Type caract. |     Droits d'accès
----------+--------------+----------+-------------+--------------+-------------------------
 b1       | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 postgres | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 template0 | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/guillaume          +
          |              |          |             |              | guillaume=CTc/guillaume
 template1 | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/guillaume          +
          |              |          |             |              | guillaume=CTc/guillaume
(4 lignes)
```

Le résultat ci-dessus vient du maître. Ré-exécutons la commande précédente mais sur le serveur hotstandby (grâce à la précision de son numéro de port) :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -p 5433 -l
                             Liste des bases de données
   Nom    | Propriétaire | Encodage |     Tri     | Type caract. |     Droits d'accès
----------+--------------+----------+-------------+--------------+-------------------------
 b1       | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 postgres | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 template0 | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/guillaume          +
          |              |          |             |              | guillaume=CTc/guillaume
 template1 | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/guillaume          +
          |              |          |             |              | guillaume=CTc/guillaume
(4 lignes)
```

Le fait que nous ayons le résultat nous indique que, non seulement psql s'est connecté au serveur en hotstandby, mais que en plus, il a pu exécuter des commandes SQL.

Maintenant, créons une base et vérifions sa présence sur les deux serveurs :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ createdb b2
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -l | grep b2
 b2       | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -p 5433 -l | grep b2
```

Rien pour le deuxième serveur. Étonnant ? Non, pas vraiment. En fait, nous ne pouvons avoir cette nouvelle base qu'à partir du moment où le journal de transactions aura été envoyé au serveur en hotstandby. Or ce journal n'est certainement pas complet. D'ailleurs les traces n'indiquent rien sur la restauration d'un journal de transactions. Donnons un peu plus d'activité sur le serveur maître, histoire de transférer quelques journaux de transactions :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -q b1
b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
b1=# \q
```

Ça a suffit pour terminer trois journaux de transactions. Ces derniers ont été archivés par le maître, découvert par pg_standby, qui les a fourni au serveur PostgreSQL hotstandby, qui lui-même les a restauré, comme le montrent les traces :

```
2010-08-21 10:51:32 CEST LOG:  restored log file "00000001000000000000000C" from archive
2010-08-21 10:51:37 CEST LOG:  restored log file "00000001000000000000000D" from archive
2010-08-21 10:51:38 CEST LOG:  restored log file "00000001000000000000000E" from archive
```

La base devrait donc être présente sur les deux serveurs maintenant.

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -l | grep b2
 b2       | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -p 5433 -l | grep b2
 b2       | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
```

Et c'est bien le cas. Comptons maintenant le nombre de lignes dans la table t1 :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -q b1
b1=# SELECT count(*) FROM t1;
  count
```

```
---------
 3000000
(1 ligne)

b1=# \q
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -q -p 5433 b1
b1=# SELECT count(*) FROM t1;
  count
---------
 2000000
(1 ligne)

b1=# \q
```

La dernière insertion n'apparaît. En fait, seule une partie de la transaction a été envoyée sur le serveur hotstandby, mais pas la transaction entière. Elle n'est donc pas encore prise en compte. Comme le reste de cette transaction doit toujours se trouver sur le journal en cours, non transmis, forçons sa transmission avec la procédure système pg_switch_xlog().

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -q b1
b1=# SELECT pg_switch_xlog();
 pg_switch_xlog
----------------
 0/FD4BB50
(1 ligne)

b1=# \q
```

Dans les traces, nous constatons bien sa restauration sur le serveur hotstandby :

```
2010-08-21 10:53:05 CEST LOG:  restored log file "000000010000000000000000F" from archive
```

Et la table a maintenant bien le bon nombre de lignes :

```
guillaume@laptop:~/tests/hotstandby/pg_log$ psql -q -p 5433 b1
b1=# SELECT count(*) FROM t1;
  count
---------
 3000000
(1 ligne)
b1=# \q
```

Ce délai d'application est gênant. Il est généralement considéré comme trop long, avec raison. C'est moins visible sur les anciennes versions car il n'était pas possible de se connecter au serveur en standby pendant la restauration. Donc aucun moyen d'utiliser de la répartition de charge par exemple. Le délai dépend évidemment de l'activité de la base de données. Plus la base de données sera active, plus le serveur en standby sera proche des données du serveur maître. Si la base n'est pas utilisée, il est tout à fait possible que la base esclave ait un délai de plusieurs minutes, voire de plusieurs heures (si vous avez très peu d'activité évidemment). Dans le cadre de la répartition de charge, c'est injouable. Pour un serveur de rapports, ça peut être suffisant.

Et voilà !

Dès maintenant, nous disposons d'un serveur maître et d'un serveur esclave. On a l'accès en lecture seule, ce qui est déjà un gros plus, il serait aussi bon d'avoir une réplication avec un lag beaucoup moins important. C'est ce que nous allons mettre en place avec le Streaming Replication dans le prochain article.