DALIBO

*L'Expertise* PostgreSQL

# Réplication interne avec PostgreSQL 9.0

# Table des matières

# Réplication avec PostgreSQL 9.0

# 1 À propos des auteurs...

» Auteur : Guillaume Lelarge
» Société : DALIBO
» Date : Janvier 2011
» URL :
https://support.dalibo.com/kb/conferences/replication_postgresql_9.0/

# 2 Licence

- Licence Creative Common BY-NC-SA
- 3 contraintes de partage :
  - Citer la source (dalibo)
  - Pas d'utilisation commerciale
  - Partager sous licence BY-NC-SA

# 3 Au menu

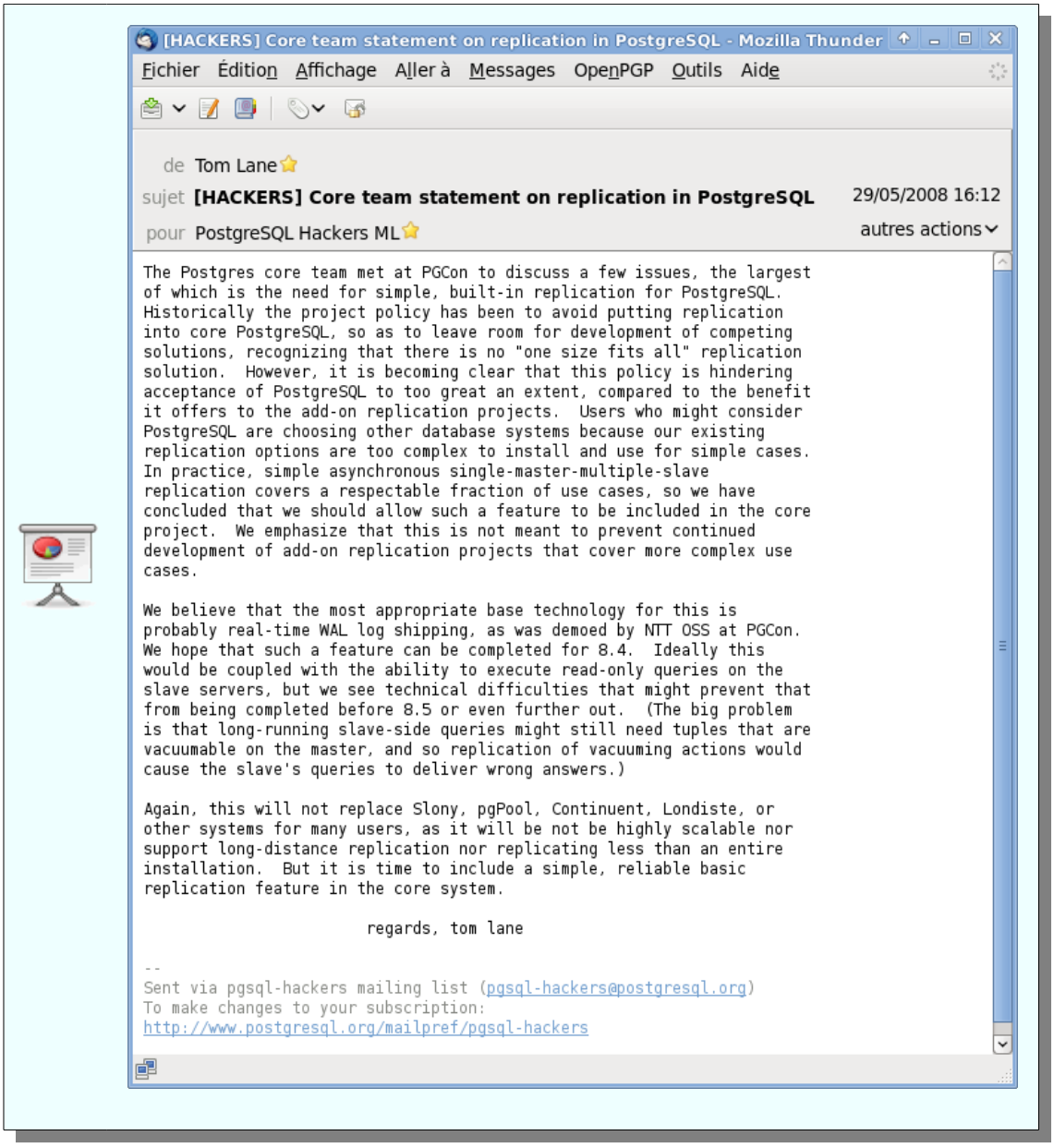
- Quelques rappels
  - Journaux de transactions
  - PITR
  - Warm Standby
- Nouveautés de la version 9.0
  - Hot Standby
  - Streaming Replication
- Et un coup d'œil dans le futur

Cette présentation fait le constat des améliorations des précédentes versions. Il présente ensuite les nouveautés de la version 9.0 concernant la réplication. Les deux points majeurs sont le Hot Standby et la Streaming Replication. Enfin, le document présente ce qui est prévu dans les prochaines versions.

# 4 Quelques rappels

- Déclaration de la Core Team
- Journaux de réplications
- PITR
- Warm Standby

## 4.1 Mail de la Core Team




Ce mail envoyé en mai 2008 a suscité un grand engouement auprès des développeurs et utilisateurs. Auparavant, la réponse traditionnelle était qu'il n'y avait pas une solution de réplication valable pour tout le monde. Du coup, il valait mieux se baser sur des projets externes, chacun pouvant répondre à un besoin de réplication spécifique.

## 4.2 Déclaration de la Core Team

- Réplication simple, interne, fiable
  - Par les journaux de transactions (technologie la plus appropriée)
  - Prévue pour la version 8.4
- Esclaves en lecture seule
  - Possibilité de repousser cette fonctionnalité à la version 8.5

La Core Team a noté l'intérêt d'avoir un système interne de réplication sans démentir que cela ne répondrait pas à tous les cas d'utilisation d'un système de réplication. La technologie la plus appropriée pour cela est les journaux de transactions. La Core Team ne fait qu'indiquer une envie d'intégration d'un système de réplication basé sur les journaux de transactions. Le système sélectionné ne dépendra encore une fois que des propositions faites par les développeurs et la communauté dans son ensemble. La Core Team espère voir arriver cela en 8.4.

La Core Team indique aussi qu'elle souhaite que les esclaves soient disponibles en lecture seule. Étant donné le travail que cela pourrait nécessiter, elle précise qu'il est probable que cela ne pourrait arriver qu'en version 8.5.

## 4.3 Journaux de transactions

- Contiennent toutes les modifications des fichiers de la base
- Par instance (*ie*, toutes les bases)
- Informations de bas niveau
  - Bloc par bloc, fichier par fichier
  - Ne contient pas la requête elle-même
- Déjà utilisé en cas de crash du serveur
  - Rejeu des transactions non synchronisées sur les fichiers

Chaque transaction, implicite ou explicite, réalisant des modifications sur la structure ou les données d'une base est tracée dans les journaux de transactions. Ces derniers contiennent des

informations d'assez bas-niveau, comme les blocs modifiés sur un fichier suite, par exemple, à un UPDATE. La requête elle-même n'apparaît jamais. Les journaux de transactions sont valables pour toutes les bases de données de l'instance.

Les journaux de transactions sont déjà utilisés en cas de *crash* du serveur. Lors du redémarrage, PostgreSQL rejoue les transactions qui n'auraient pas été synchronisées sur les fichiers de données.

Comme toutes les modifications sont disponibles dans les journaux de transactions et que PostgreSQL sait rejouer les transactions à partir des journaux, il suffit d'archiver les journaux sur une certaine période de temps pour pouvoir les rejouer.

## 4.4 PITR

- Point In Time Recovery
- Possibilité de rejouer
  - Tous les journaux de transactions
  - Jusqu'à un certain point dans le temps
  - Jusqu'à un certain identifiant de transaction
- Rejeu à partir d'une sauvegarde des fichiers à un instant *t*
- Disponible à partir de PostgreSQL 8.0

La technologie PITR est disponible depuis la version 8.0. Cette dernière permet le rejeu de tous les journaux de transactions préalablement archivés ou tous les journaux jusqu'à un certain point dans le temps, ou encore tous les journaux jusqu'à un certain identifiant de transaction.

Pour cela, il est nécessaire d'avoir une sauvegarde des fichiers de l'instance (réalisée à chaud une fois l'archivage activé) et des journaux archivés depuis cette sauvegarde.

## 4.5 Warm Standby

- Esclave mis à jour en permanence

- Fichier par fichier
  - Fichier == un journal de transactions
- Esclave non disponible pendant la restauration
- En 8.3, ajout de l'outil pg_standby

L'idée du serveur en Warm Standby est de rejouer en permanence les journaux de transactions archivés. Autrement dit, quand le serveur maître a terminé de travailler sur un journal de transactions, il l'archive sur un deuxième serveur où il sera récupéré par le serveur PostgreSQL esclave qui le rejouera dès la fin de la copie.

C'est un système de réplication complet. Mais deux gros inconvénients apparaissent : le délai de prise en compte des modifications dépend de l'activité du serveur maître (plus ce dernier sera actif, plus il enverra rapidement un journal de transactions, plus le serveur esclave sera à jour) et le serveur esclave n'est pas disponible, y compris pour des requêtes en lecture seule.

Plutôt que d'avoir à écrire son propre outil, la version 8.3 propose dans les modules contrib un outil appelé pg_standby. De même, Skype propose son propre outil appelé walmgr. Un dernier outil permet aussi de faciliter la restauration : pitrtools.

## 4.6 Démonstration Warm Standby

- Installation d'un serveur en Warm Standby
- Conditions
  - Installation sur mon portable
  - Deux instances
  - Donc deux ports et deux répertoires de données

Tout d'abord, il faut créer le répertoire où seront stockées les données du serveur maître et du serveur esclave. Comme tout se fait sur la même machine dans le cas d'un test, ce répertoire va se trouver dans le répertoire `/tmp` :

```
$ mkdir /tmp/replication
```

```
$ cd /tmp/replication
$ mkdir /tmp/replication/archivage
$ initdb s1
```

Nous allons modifier le fichier de configuration. Pour simplifier les choses, nous allons demander l'import du fichier `s1.conf` par le fichier `postgresql.conf`. Nous ajoutons donc à la fin du fichier `postgresql.conf` la ligne suivante :

```
include 's1.conf'
```

Le fichier `s1.conf` contient les lignes suivantes :

```
# Traces
logging_collector = on
log_filename = 'postgresql-%Y-%m-%d.log'
log_line_prefix = '%t '
lc_messages = 'C'

# Archivage
archive_mode = on
archive_command = 'cp %p /tmp/replication/archivage/%f'

# Journaux de transactions
wal_level = archive
```

La partie *Traces* permet de simplifier la gestion des traces de PostgreSQL. La partie *Archivage* précise que nous activons l'archivage des journaux de transactions et indique la commande à exécuter pour archiver un journal de transactions. Quant au paramètre `wal_level`, il s'agit d'un nouveau paramètre de la version 9.0. Ce dernier doit valoir au moins `archive` pour qu'il soit possible de faire de l'archivage.

Il ne nous reste plus qu'à démarrer le serveur PostgreSQL maître :

```
$ pg_ctl -D s1 start
server starting
```

Les traces indiquent que le système a bien démarré :

```
2011-01-24 15:30:39 CET LOG:  database system was shut down at 2011-01-24 15:29:41 CET
2011-01-24 15:30:39 CET LOG:  database system is ready to accept connections
2011-01-24 15:30:39 CET LOG:  autovacuum launcher started
```

Pour construire le serveur secondaire, nous avons besoin de copier les fichiers de données mais avant cela, nous devons utiliser la procédure stockée `pg_start_backup()` :

```
$ psql -c "SELECT pg_start_backup('postgresql-sessions #1', true)" postgres
 pg_start_backup
-----------------
 0/2000020
(1 row)
```

La copie se fait avec un simple `cp` :

```
$ cp -r s1 s2
```

Enfin, la procédure stockée `pg_stop_backup()` doit être exécutée :

```
$ psql -c "SELECT pg_stop_backup()" postgres
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/20000D8
(1 row)
```

Notez le nouveau message, de niveau NOTICE, indiquant que tous les journaux de transactions nécessaires à la restauration de cette sauvegarde de fichiers ont bien été archivés.

Maintenant, nous devons configurer le serveur esclave. Nous allons remplacer l'import du fichier `s1.conf` par celui de `s2.conf` dans le fichier `postgresql.conf`. La dernière ligne de ce fichier doit donc ressembler à :

```
include 's2.conf'
```

Quant à `s2.conf`, il doit contenir ce qui suit :

```
# Car les deux instances sont sur le meme portable
port = 5433

# Traces
logging_collector = on
log_filename = 'postgresql-%Y-%m-%d.log'
log_line_prefix = '%t '
lc_messages = 'C'
```

La partie *Traces* est déjà connue. Le port est modifié pour permettre aux deux instances d'être lancées sur la même machine.

Il nous faut ensuite un fichier de restauration appelé `recovery.conf` dont voici le contenu :

```
restore_command = 'pg_standby -s 1 -t /tmp/STOP /tmp/replication/archivage %f %p %r'
```

Avant de lancer le serveur esclave, nous devons supprimer certains fichiers comme les journaux de transactions et le fichier PID :

```
$ rm s2/pg_log/* s2/pg_xlog/* s2/postmaster.pid
rm: cannot remove `s2/pg_xlog/archive_status': Is a directory
```

Le message d'erreur est *normal*, nous ne voulons pas supprimer ce répertoire.

Lançons le serveur esclave :

```
$ pg_ctl -D s2 start
server starting
```

Voici les traces du démarrage :

```
2011-01-24 15:32:13 CET LOG:  database system was interrupted; last known up at 2011-01-24 15:30:58
CET
```

```
2011-01-24 15:32:13 CET LOG:  starting archive recovery
2011-01-24 15:32:13 CET LOG:  restored log file "000000010000000000000002" from archive
2011-01-24 15:32:13 CET LOG:  redo starts at 0/2000078
2011-01-24 15:32:13 CET LOG:  consistent recovery state reached at 0/3000000
```

PostgreSQL découvre une instance qui a été interrompue, ce qui est logique vu que nous avons sauvegardé les fichiers alors que le serveur était en cours d'exécution. Il commence donc la phase de restauration. Il restaure d'ailleurs un premier journal de transactions (000000010000000000000002) et atteint un état cohérent, mais reste en mode de restauration comme l'indique la commande ps :

```
$ ps --forest x | grep postgres | grep -v [g]rep
 6960 pts/0    S      0:00 /opt/postgresql-9.0/bin/postgres -D s2
 6964 ?        Ss     0:00  \_ postgres: logger process
 6965 ?        Ss     0:00  \_ postgres: startup process   waiting for 000000010000000000000003
 6970 ?        Ss     0:00  \_ postgres: writer process
 6867 pts/0    S      0:00 /opt/postgresql-9.0/bin/postgres -D s1
 6871 ?        Ss     0:00  \_ postgres: logger process
 6873 ?        Ss     0:00  \_ postgres: writer process
 6874 ?        Ss     0:00  \_ postgres: wal writer process
 6875 ?        Ss     0:00  \_ postgres: autovacuum launcher process
 6876 ?        Ss     0:00  \_ postgres: archiver process   last was
000000010000000000000002.00000020.backup
 6877 ?        Ss     0:00  \_ postgres: stats collector process
```

Cette commande nous indique bien les deux serveurs PostgreSQL. Notez le processus d'archivage (archiver process) sur le maître et le processus de restauration (startup process) sur l'esclave. Ce dernier est en attente du prochain journal de transactions, le 000000010000000000000003. Créons un peu d'activité pour voir que la restauration continue bien :

```
$ createdb b1
$ psql b1
psql (9.0.2)
Type "help" for help.

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

Cela résulte en la création d'une base et d'une table, cette dernière contenant un million de lignes.

Que nous disent les traces du serveur esclave ?

```
2011-01-24 15:33:08 CET LOG:  restored log file "000000010000000000000003" from archive
2011-01-24 15:33:10 CET LOG:  restored log file "000000010000000000000004" from archive
2011-01-24 15:33:11 CET LOG:  restored log file "000000010000000000000005" from archive
```

Les journaux 3, 4 et 5 ont été archivés. Le processus de restauration doit donc être en attente du journal 6.

```
$ ps --forest x | grep postgres | grep -v [g]rep
 5830 pts/0    S      0:00 /opt/postgresql-9.0/bin/postgres
 5834 ?        Ss     0:00  \_ postgres: logger process
 5835 ?        Ss     0:00  \_ postgres: startup process   waiting for 000000010000000000000006
```

```
 5840 ?        Ss     0:00  \_ postgres: writer process
 5738 pts/0    S      0:00 /opt/postgresql-9.0/bin/postgres
 5742 ?        Ss     0:00  \_ postgres: logger process
 5744 ?        Ss     0:00  \_ postgres: writer process
 5745 ?        Ss     0:00  \_ postgres: wal writer process
 5746 ?        Ss     0:00  \_ postgres: autovacuum launcher process
 5747 ?        Ss     0:00  \_ postgres: archiver process   last was 000000010000000000000005
 5748 ?        Ss     0:00  \_ postgres: stats collector process
```

Ce qui est bien le cas.

Dernier point, n'essayez pas de vous connecter sur le serveur esclave, ce n'est pas possible:

```
$ psql -p5433 b1
psql: FATAL:  the database system is starting up
```

# 5 Version 9.0

- Comble les deux grosses lacunes des anciennes versions
- Esclaves disponibles en lecture seule
- Réplication avec moins de lag
  - Pratiquement synchrone
  - Mais techniquement toujours asynchrone

Entre autres problèmes du Warm Standby, le fait que les esclaves ne soient pas accessibles même en lecture seule est un soucis pour les entrepôts de données par exemple, où la génération d'un rapport est plus facilement faisable sur un serveur en lecture seule. De plus, le lag causé par l'attente du remplissage complet d'un journal de transactions avant son archivage est un soucis pour ceux qui veulent avoir des esclaves très proches des données du maître. Il est possible de configurer un délai avant un archivage forcé mais cela ne fait que consommer encore plus de place dans la partition contenant les journaux archivés.

La version 9.0 corrige donc ces deux soucis.

## 5.1 Hot Standby - introduction

- Patch développé par la société 2nd Quadrant
- Financement par un grand nombre de sociétés
- Deux ans de développement
- Initialement prévue pour la version 8.4
  - Mais pas suffisamment stable pour cette version
  - Donc repoussée à la suivante
- Permet l'accès en lecture seule aux serveurs esclaves

Ce patch est à l'initiative de Simon Riggs, de la société 2nd Quadrant. Il a bénéficié d'un financement de plusieurs entreprises pour terminer ce patch, au bout de deux années. Bien que ce dernier devait être intégré en 8.4, il n'était pas suffisamment abouti pour pouvoir être intégré dans le code source de PostgreSQL. Il a donc fallu attendre la version 9.0 pour le voir arriver.

Le but de ce patch est de permettre l'accès en lecture seule aux serveurs esclaves. Les ordres de modifications de la structure de la base comme des données sont interdits car l'utilisateur se trouve dans une transaction en lecture seule.

## 5.2 Hot Standby - maitre

- Configuration d'archivage habituelle
- Paramètre wal_level
  - Nouveau paramètre de la version 9.0
  - Permet d'indiquer le niveau d'informations dans les journaux de transactions
  - Valeur obligatoire : 'hot_standby'
- Redémarrage du maître

Le paramètre `wal_level` apparaît en version 9.0. Nous l'avons déjà placé au niveau `archive` pour permettre l'archivage des journaux de transactions. Pour permettre en plus l'accès en lecture seule aux serveurs esclaves, les journaux de transactions doivent contenir encore plus d'informations, d'où la montée au niveau `hot_standby`.

En fois ce paramètre changé dans le fichier `postgresql.conf` du serveur maître, il faut redémarrer le serveur maître.

## 5.3 Hot Standby - esclave

- Configuration normale d'un Warm Standby
- Et paramètre hot_standby
  - Nouveau paramètre de la version 9.0
  - Permet de démarrer un serveur en Hot Standby
  - Valeur obligatoire : on
- Redémarrage de l'esclave
- Message dans les traces
  - `LOG: database system is ready to accept read only connections`

Sur l'esclave, il faut aussi modifier un paramètre de configuration, à savoir `hot_standby`. Ce dernier doit être activé (autrement dit, valoir `on`) pour que le serveur en question permette les requêtes en lecture seule.

Ce paramétrage ne sera pris en compte qu'en redémarrant le serveur esclave. Vous devriez voir apparaître dans les traces le message :

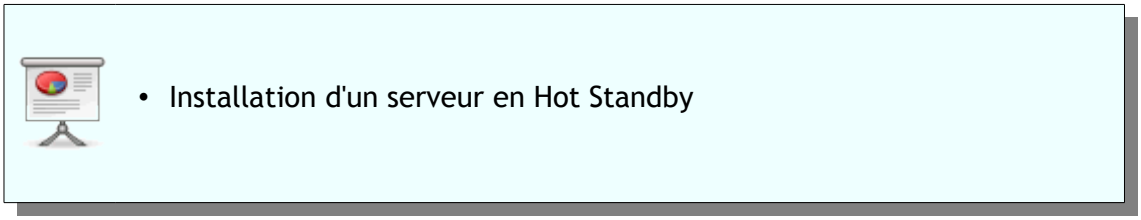
```
LOG:  database system is ready to accept read only connections
```

Dans le cas contraire, il est possible que la configuration ne soit pas correcte sur le maître. Assurez-vous que le paramètre `wal_level` soit bien configuré à `hot_standby` (non pas en regardant le fichier de configuration, mais en vous connectant sur le serveur maître et en exécutant l'ordre SQL `SHOW wal_level`). Si la valeur de ce paramètre est mauvaise, retournez à la section précédente et corrigez ce problème.

Il est possible que la configuration du maître et de l'esclave soit bonne et que vous continuiez à avoir ce message. Cela arrive quand le dernier journal de transactions archivé contient à la fois des transactions pour un `hot_standby` à `minimal` ou `archive` et des transactions pour un

`wal_level` à `hot_standby`. Dans ce cas, il vous faut désactiver `hot_standby` sur l'esclave, redémarrer l'esclave, forcer un changement de journal de transactions sur le maître avec la fonction `pg_switch_xlog()`, attendre le rejeu de ce journal, activer `hot_standby` sur l'esclave, et enfin redémarrer l'esclave.

## 5.4 Démonstration Hot Standby




Pour passer le serveur Warm Standby en Hot Standby, il faut modifier le fichier de configuration du serveur maître pour monter le niveau des journaux de transactions. Autrement dit, les lignes suivantes sont à ajouter dans le fichier `s1.conf` :

```
# Hot Standby
wal_level = hot_standby
```

Ceci fait, PostgreSQL doit être redémarré pour que la modification de configuration soit prise en compte :

```
$ pg_ctl -D s1 restart
waiting for server to shut down.... done
server stopped
server starting
```

Le journal applicatif n'indiquera que les informations standards dans ce cas :

```
2011-01-24 15:34:14 CET LOG:  received smart shutdown request
2011-01-24 15:34:14 CET LOG:  autovacuum launcher shutting down
2011-01-24 15:34:14 CET LOG:  shutting down
2011-01-24 15:34:15 CET LOG:  database system is shut down
2011-01-24 15:34:15 CET LOG:  database system was shut down at 2011-01-24 15:34:15 CET
2011-01-24 15:34:15 CET LOG:  autovacuum launcher started
2011-01-24 15:34:15 CET LOG:  database system is ready to accept connections
```

Pour éviter d'avoir un journal de transactions en partie rempli par des transactions au niveau `archive`, nous allons forcer tout de suite le changement de journal :

```
$ psql -c "select pg_switch_xlog()" postgres
 pg_switch_xlog
----------------
 0/70000C8
(1 row)
```

Et nous allons attendre que ce dernier soit rejoué sur le serveur s2 (toujours en Warm Standby) :

```
2011-01-24 15:35:43 CET LOG:  restored log file "000000010000000000000007" from archive
```

Maintenant, nous pouvons configurer le serveur en standby, s2. Les lignes à ajouter dans le fichier de configuration sont les suivantes:

```
# Hot Standby
hot_standby = on
```

Et nous pouvons redémarrer le serveur s2 :

```
$ pg_ctl -D s2 restart
waiting for server to shut down.... done
server stopped
server starting
```

Les traces vont être intéressantes :

```
2011-01-24 15:36:48 CET LOG:  received smart shutdown request
2011-01-24 15:36:48 CET LOG:  shutting down
2011-01-24 15:36:48 CET LOG:  database system is shut down
2011-01-24 15:36:49 CET LOG:  database system was shut down in recovery at 2011-01-24 15:36:48 CET
2011-01-24 15:36:49 CET LOG:  starting archive recovery
2011-01-24 15:36:49 CET LOG:  restored log file "000000010000000000000007" from archive
2011-01-24 15:36:49 CET LOG:  redo starts at 0/7000078
2011-01-24 15:36:49 CET LOG:  consistent recovery state reached at 0/70000A8
2011-01-24 15:36:49 CET LOG:  database system is ready to accept read only connections
```

La dernière ligne indique que le serveur est prêt à recevoir des connexions en lecture seule. Faisons quelques tests

```
$ psql b1
psql (9.0.2)
Type "help" for help.

b1=# SELECT count(*) FROM t1;
  count
---------
 1000000
(1 row)

b1=# \q
$ psql -p5433 b1
psql (9.0.2)
Type "help" for help.

b1=# SELECT count(*) FROM t1;
  count
---------
 1000000
(1 row)

b1=# \q
```

Il y a bien un million de lignes sur les tables t1 des deux serveurs. Ajoutons de nouvelles lignes :

```
$ psql b1
psql (9.0.2)
Type "help" for help.

b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
INSERT 0 1000000
b1=# \q
guillaume@laptop:/tmp/replication$ psql -p5433 b1
psql (9.0.2)
Type "help" for help.

b1=# SELECT count(*) FROM t1;
  count
---------
 1000000
(1 row)

b1=# \q
```

Il y a toujours un million de lignes sur l'esclave. Le problème ici est dû au fait que le dernier journal de transactions, celui qui indique que la transaction d'insertion a été COMMITée n'est pas encore arrivé au serveur esclave. Insérons un autre million de lignes.

```
guillaume@laptop:/tmp/replication$ psql b1
psql (9.0.2)
Type "help" for help.

b1=# insert into t1 select generate_series(1, 1000000);
INSERT 0 1000000
b1=# select count(*) from t1;
  count
---------
 3000000
(1 row)

b1=# \q
guillaume@laptop:/tmp/replication$ psql -p5433 b1
psql (9.0.2)
Type "help" for help.

b1=# select count(*) from t1;
  count
---------
 2000000
(1 row)

b1=# \q
```

Suite aux insertions le COMMIT précédent a été poussé jusqu'au serveur secondaire, mais pas le deuxième COMMIT (implicite). Essayons un simple `pg_switch_xlog()` :

```
$ psql b1
psql (9.0.2)
Type "help" for help.

b1=# select pg_switch_xlog();
 pg_switch_xlog
----------------
 0/FA91F78
(1 row)

b1=# \q
guillaume@laptop:/tmp/replication$ psql -p5433 b1
psql (9.0.2)
Type "help" for help.
```

```
b1=# select count(*) from t1;
  count
---------
 3000000
(1 row)
```

Après avoir forcé le changement de journal de transactions, et donc son archivage, nous nous retrouvons bien avec trois millions de lignes dans la table t1.

La réplication est bien fonctionnelle, nous avons un esclave en lecture seule mais le lag dans la réplication rend l'utilisation de l'esclave assez complexe.

## 5.5 Streaming Replication - introduction

- Patch basé sur le développement réalisé par NTT
- Permet d'avoir moins de lag dans la réplication
- Deux nouveaux processus:
  - walreceiver sur l'esclave
  - walsender sur le maître
  - L'esclave se connecte au maître
- Semi-asynchrone
  - Pas synchrone
  - Mais très rapide malgré tout

NTT, la société de télécom japonaise, a présenté lors du PGCon 2008 un système de réplication synchrone sur PostgreSQL (voir http://www.pgcon.org/2008/schedule/events/76.en.html). Ayant décidé de le fournir à la communauté, les développeurs de NTT ont travaillé avec la communauté pour l'intégrer. L'intégration est partielle, la partie synchrone n'ayant pas été récupérée. Mais le reste y ressemble beaucoup : deux processus qui communiquent pour envoyer les informations des journaux de transactions, groupe de transactions par groupe de transactions (des groupes bien plus fins qu'un journal de transactions).

## 5.6 SR - maitre (1/3)

- Configuration du postgresql.conf
- Paramètre max_wal_senders
  - Nouveau paramètre de la version 9.0
  - Nombre de processus pouvant se connecter au maître pour de la réplication
  - Au minimum un par esclave
  - Peut-être supérieur si perte réseau fréquente

Le paramètre `max_wal_senders` du fichier `postgresql.conf` indique le nombre maximum de processus autorisés à se connecter sur le maître pour faire de la réplication. Il faut en effet éviter qu'un trop grand nombre d'esclaves ne prenne toute la puissance du serveur. Il faut donc au moins un processus par serveur esclave. Il est possible d'en avoir plus quand le réseau n'est pas fiable. En effet, après une déconnexion, le client s'en rend compte rapidement et peut demander à son processus de se connecter alors que le serveur maître croit toujours avoir une connexion du serveur esclave. Cela arrive notamment s'il faut passer des pare-feux ou des routeurs pour atteindre le maître.

## 5.7 SR - maitre (2/3)

- Configuration du pg_hba.conf
- Ligne supplémentaire pour chaque esclave
  - Champ database : replication
  - Champ user : intéressant d'avoir un utilisateur spécifique pour la réplication
  - Champ CIDR : bien préciser le serveur esclave
  - Champ authentification : md5 (de préférence)

Comme un processus client doit se connecter sur le serveur maître, la connexion doit être

autorisée dans le fichier de contrôle des accès distants (`pg_hba.conf`). Seule particularité de cette connexion, elle utilise un nom de base virtuelle : replication. Le reste des champs est aux choix de l'administrateur. Néanmoins, il est fortement conseillé que la connexion se fasse avec un utilisateur précis, qui a un mot de passe et que ce mot de passe soit vérifié. Du coup, une authentification `md5` est raisonnable.

## 5.8 SR - maitre (3/3)

- Redémarrage du maître

Après toute cette configuration (en fait surtout après celle du fichier `postgresql.conf`), il est nécessaire de redémarrer le serveur PostgreSQL maître.

## 5.9 SR - esclave (1/3)

- Configuration du recovery.conf
- Ne pas utiliser pg_standby !
  - Si le fichier à restaurer n'existe pas, PostgreSQL passe en mode streaming
- Exemple de configuration
  - restore_command = 'cp /tmp/replication/archivage/%f %p'

Toute la configuration de l'esclave se fait dans le fichier `recovery.conf`. La commande de restauration change par rapport à un serveur standby habituel. Il ne faut surtout pas que l'outil exécuté reste en attente du prochain journal de transactions. PostgreSQL passe en mode streaming (flux) à partir du moment où il a récupéré le dernier journal de transactions archivé. Il faut donc repasser à une commande du style `cp` ou `scp`.

## 5.10 SR - esclave (2/3)

- Nouveaux paramètres (toujours recovery.conf)
- archive_cleanup_command
  - Utilisation du module contrib pg_archivecleanup
  - Permet de supprimer les anciens journaux de transactions devenus inutiles
- trigger_file
  - L'existence du fichier déclenche la sortie du mode de restauration

Abandonner `pg_standby` nous empêche de bénéficier de ces avantages, comme le nettoyage des anciens journaux de transactions devenus inutiles et le passage en maître avec le fichier trigger. Ces deux fonctionnalités ont donc été ajoutées à PostgreSQL. Les paramètres permettant de les configurer sont respectivement `archive_cleanup_command` et `trigger_file`. Le premier permet l'exécution d'un outil qui fera le nettoyage, le second indique le fichier dont la présence permet à l'esclave de passer en lecture/écriture.

Pour le paramètre `archive_cleanup_command`, il est à noter l'apparition d'un module contrib `pg_archivecleanup` qui est parfait pour ce travail.

## 5.11 SR - esclave (3/3)

- Nouveaux paramètres (toujours recovery.conf)
- standby_mode
  - Activation du mode streaming
  - Valeur obligatoire : on
- primary_conninfo
  - DSN de connexion au maître
  - Exemple : host=localhost port=5432 user=repli
- Redémarrage de l'esclave

Le serveur esclave doit savoir auprès de quel serveur maître il doit se connecter. C'est le but du paramètre `primary_conninfo` qui est un simple DSN où tous les paramètres de connexion habituels sont utilisables (`host`, `port`, `application_name`, etc.). Pour que la valeur de ce paramètre soit utilisée, il faut que le paramètre `standby_mode` soit activé.

## 5.12 SR - suite

- Traces sur le serveur esclave :
  - `LOG: streaming replication successfully connected to primary`
- Traces sur le serveur maître :
  - `LOG: replication connection authorized: user=guillaume host=::1 port=41910`

Une fois le serveur esclave redémarré, ce dernier tente de restaurer les journaux de transactions archivés. Arrivé au dernier, il bascule en mode streaming. Si la connexion sur le serveur maître réussit, il affiche la ligne suivante dans les traces :

```
LOG:  streaming replication successfully connected to primary
```

À ce moment-là, le serveur maître indique aussi la connexion dans son journal applicatif :

```
LOG:  replication connection authorized: user=guillaume host=::1 port=41910
```

## 5.13 Démonstration Streaming Replication

- Installation d'un serveur en Streaming Replication

Nous allons faire en sorte qu'un seul serveur esclave puisse se connecter au serveur maître en ajoutant les lignes suivantes dans le fichier `s1.conf` :

```
# Streaming Replication
max_wal_senders = 1
```

Le fichier des accès distants est modifié pour ressembler à ceci :

```
# TYPE  DATABASE        USER            CIDR-ADDRESS            METHOD

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

Ensuite, il faut redémarrer le serveur maître PostgreSQL :

```
$ pg_ctl -D s1 restart
waiting for server to shut down.... done
server stopped
server starting
```

Les traces n'indiquent rien de particulier.

Ensuite, il faut configurer le serveur esclave avec le fichier `recovery.conf` :

```
restore_command = 'cp /tmp/replication/archivage/%f %p'
archive_cleanup_command = 'pg_archivecleanup /tmp/replication/archivage %r'
trigger_file = '/tmp/STOP'

standby_mode = 'on'
primary_conninfo = 'host=localhost port=5432'
```

Là-aussi, pour que les changements dans la configuration soient pris en compte, il faut redémarrer PostgreSQL :

```
$ pg_ctl -D s2 restart
waiting for server to shut down.... done
server stopped
server starting
```

Par contre là, les traces vont changer :

```
2011-01-24 15:58:44 CET LOG:  received smart shutdown request
2011-01-24 15:58:44 CET LOG:  shutting down
2011-01-24 15:58:44 CET LOG:  database system is shut down
2011-01-24 15:58:46 CET LOG:  database system was shut down in recovery at 2011-01-24 15:58:44 CET
2011-01-24 15:58:46 CET LOG:  entering standby mode
2011-01-24 15:58:46 CET LOG:  restored log file "000000010000000000000010" from archive
2011-01-24 15:58:46 CET LOG:  redo starts at 0/100001D0
2011-01-24 15:58:46 CET LOG:  consistent recovery state reached at 0/11000000
2011-01-24 15:58:46 CET LOG:  database system is ready to accept read only connections
cp: cannot stat `/tmp/replication/archivage/000000010000000000000011': No such file or directory
2011-01-24 15:58:46 CET LOG:  streaming replication successfully connected to primary
```

Le serveur esclave a restauré le journal de transactions 10 mais n'a pas trouvé le 11, du coup il essaie le mode streaming. Et la dernière ligne indique que la connexion au serveur maître a réussi. Ce qui se voit aussi dans les traces du serveur maître :

```
2011-01-24 15:58:46 CET LOG:  replication connection authorized: user=guillaume host=::1 port=41910
```

Les deux nouveaux processus sont présents :

```
guillaume@laptop:/tmp/replication$ ps --forest x | grep postgres | grep -v [g]rep
 7555 pts/0    S      0:00 /opt/postgresql-9.0/bin/postgres -D s2
 7559 ?        Ss     0:00  \_ postgres: logger process
 7560 ?        Ss     0:00  \_ postgres: startup process   recovering 000000010000000000000011
 7563 ?        Ss     0:00  \_ postgres: writer process
 7564 ?        Ss     0:00  \_ postgres: stats collector process
 7567 ?        Ss     0:00  \_ postgres: wal receiver process   streaming 0/11000078
 7496 pts/0    S      0:00 /opt/postgresql-9.0/bin/postgres -D s1
 7500 ?        Ss     0:00  \_ postgres: logger process
 7502 ?        Ss     0:00  \_ postgres: writer process
 7503 ?        Ss     0:00  \_ postgres: wal writer process
 7504 ?        Ss     0:00  \_ postgres: autovacuum launcher process
 7505 ?        Ss     0:00  \_ postgres: archiver process
 7506 ?        Ss     0:00  \_ postgres: stats collector process
 7568 ?        Ss     0:00  \_ postgres: wal sender process guillaume ::1(41910) streaming
0/11000078
```

On voit d'ailleurs que les deux serveurs sont synchrones à cet instant-là.

Testons l'insertion d'un million de lignes :

```
$ psql b1
psql (9.0.2)
Type "help" for help.

b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);
INSERT 0 1000000
b1=# SELECT count(*) FROM t1;
  count
---------
 4000000
(1 row)

b1=# \q
guillaume@laptop:/tmp/replication$ psql -p 5433 b1
psql (9.0.2)
Type "help" for help.

b1=# select count(*) from t1;
  count
---------
 4000000
(1 row)
```

```
b1=# \q
```

Il n'y a pas eu besoin de forcer l'envoi du journal de transactions, les données sont immédiatement apparus sur le serveur esclave.

Essayons une opération beaucoup plus petite, l'ajout d'une table :

```
guillaume@laptop:/tmp/replication$ psql b1
psql (9.0.2)
Type "help" for help.

b1=# CREATE TABLE t2();
CREATE TABLE
b1=# \q
guillaume@laptop:/tmp/replication$ psql -p 5433 b1
psql (9.0.2)
Type "help" for help.

b1=# \d
         List of relations
 Schema | Name | Type  |   Owner
--------+------+-------+-----------
 public | t1   | table | guillaume
 public | t2   | table | guillaume
(2 rows)
```

La table est bien présente.

Attention, ce n'est pas du synchrone et l'activité n'est pas très importante. Néanmoins, l'esclave est dans un état semi-synchrone.

# 6 Administration

- Failover/switchover
- Surveillance

## 6.1 Failover

- Le maître est mort
- L'esclave doit être disponible en lecture/écriture
  - Créer le fichier d'arrêt de la restauration sur l'esclave
  - Puis rediriger les connexions vers le serveur esclave

Si le maître est mort, il faut faire un failover. Cette opération est simple à réaliser car il suffit de créer le fichier indiqué par le paramètre `trigger_file`. Ensuite, les clients devront passer par le nouveau serveur maître, ce qui demandera de les configurer pour qu'ils utilisent une autre adresse IP ou de configurer l'adresse IP virtuelle pour qu'elle pointe sur le bon serveur.

## 6.2 Switchover

- Basculer le serveur maître
- Étapes
  - Arrêter le serveur maître
  - Créer le fichier d'arrêt de la restauration sur l'esclave
  - Reconstruire l'ancien maître en nouvel esclave
  - Reconstruire les autres esclaves ?
- Pas simple et peut être long
  - 1ère solution : rsync
  - 2è solution : repmgr

Pour un switchover, le maître n'est pas mort mais on souhaite malgré tout changer le maître (par exemple pour faire des opérations de maintenance sur le maître original). La première chose à faire est d'arrêter le maître pour être sûr que des clients ne continuent pas à écrire dessus. Ensuite, il faut créer le fichier trigger pour que l'esclave se comporte en maître.

Il reste deux problèmes : comment faire en sorte que l'ancien maître devienne un esclave du

nouveau maître et comment faire en sorte que les anciens esclaves de l'ancien maître deviennent des esclaves du nouveau maître. La réponse est simple et ennuyante : il faut les reconstruire. Étant donné que la majorité des données sont déjà présentes sur l'ancien maître et les anciens esclaves, la façon la plus rapide de recopier les données est d'utiliser `rsync`, ce qui sera beaucoup plus rapide que `cp` ou `scp`.

Un nouveau projet est sorti récemment pour gérer plus facilement ce genre de cas : `repmgr` (http://projects.2ndquadrant.com/repmgr), codé par la société 2ndQuadrant.

## 6.3 Surveillance

- Trois nouvelles fonctions système
  - pg_is_in_recovery()
  - pg_last_xlog_receive_location() et pg_last_xlog_replay_location()
- Plugin munin écrit par Magnus Hagander
- Action check_postgres.pl écrit par Nicolas Thauvin

Il y a très peu de possibilités pour surveiller un serveur en Hot Standby/Streaming Replication.

La fonction `pg_is_in_recovery()` permet de savoir si un serveur est un esclave en mode Hot Standby ou un maître (dans le cas d'un esclave en mode Warm Standby, il est impossible de se connecter et donc d'exécuter cette fonction).

Les fonctions `pg_last_xlog_receive_location()` et `pg_last_xlog_replay_location()` indiquent respectivement la dernière position reçue et la dernière position rejouée dans les journaux de transactions sur un serveur en Hot Standby. Cette information est à comparer au résultat fourni par l'exécution de la fonction `pg_current_xlog_location()` sur le serveur maître. C'est par exemple ce que fait pgPool-II pour savoir si un esclave accuse un retard trop important.

Le plugin munin, disponible sur https://github.com/mhagander/munin-plugins/blob/master/postgres/postgres_streaming_.in, utilise ces informations pour tracer le lag entre un maître et un esclave. L'action hot_standby_delay de check_postgres.pl permet aussi de connaître ce lag et de faire en sorte que Nagios prévienne un administrateur si le lag devient trop important.

# 7 Avantages de la version 9.0

- Facile à mettre en place
- Ne nécessite aucune modification des applications
- Ne désactive aucune fonctionnalité
  - Désactive néanmoins certaines optimisations
- Dispose des esclaves en lecture seule avec un lag très léger

# 8 Inconvénients de la version 9.0

- Administration complexe
  - Configuration plus complexe que l'attente exprimée
- Surveillance difficile
- Implication sur la sécurité mal appréhendée
- Pas de réplication synchrone
- Pas de réplication maître/maître
- Réplication de l'instance complète

# 9 Et les prochaines versions ?

- 9.1
  - Nouvel attribut REPLICATION pour les rôles
  - Nouvelle vue pg_stat_replication
  - Nouvelle vue pg_stat_database_conflicts
  - Nouvelle fonction de mise en pause de la réplication
  - Module contrib pg_basebackup
- 9.X ?
  - Réplication synchrone

# 10 Conclusion

- La réplication interne est fiable et rapide
- Elle a aussi des défauts
- Mais les développeurs travaillent à les corriger