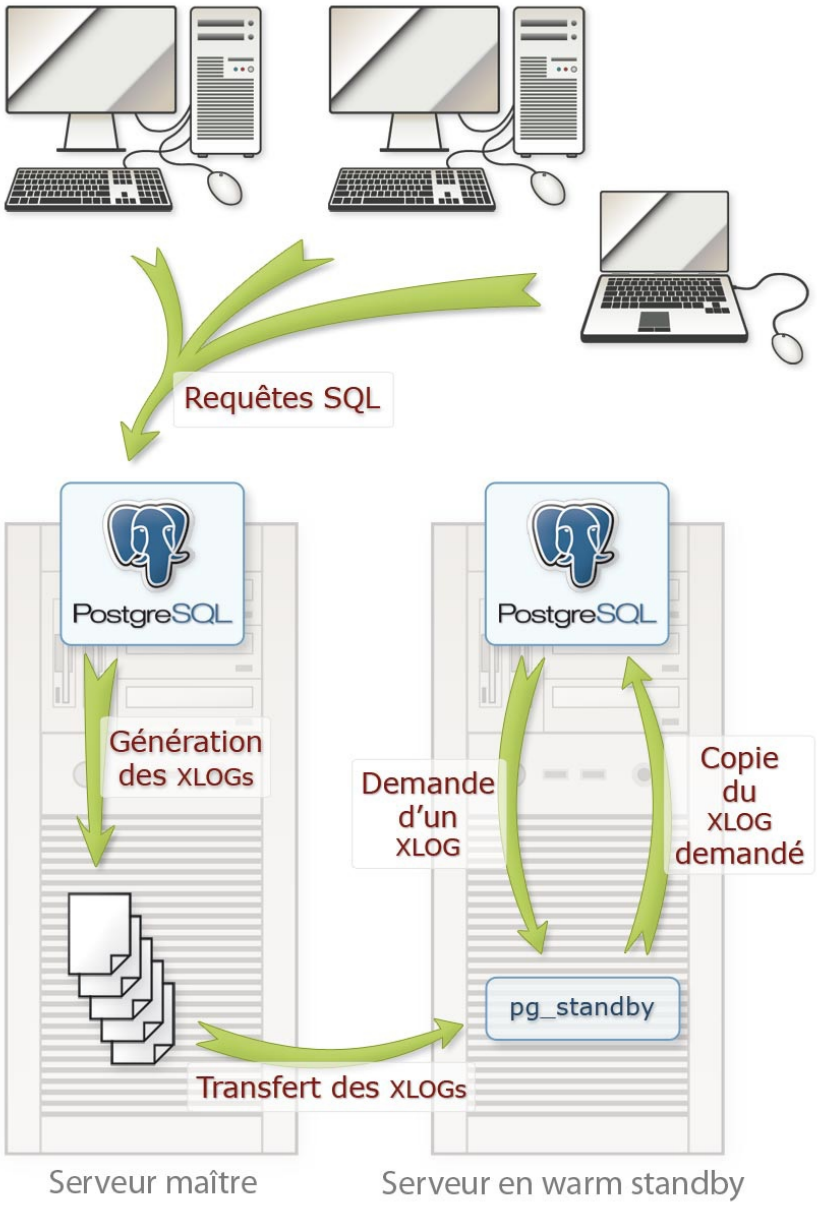
−Table des matières

La réplication par les journaux de transactions

Cet article, écrit par Guillaume Lelarge, a été publié dans le hors-série 44 du magazine GNU/Linux Magazine France, hors-série dédié à PostgreSQL. Il est disponible maintenant sous licence Creative Commons.

Toutes les données modifiées dans PostgreSQL sont tout d'abord stockées dans les journaux de transactions. Une solution particulièrement simple pour la réplication revient donc à archiver les journaux de transactions et à les rejouer sur un serveur distant.

Le schéma suivant présente la réplication en Log Shipping.



Les clients envoient leur requêtes SQL sur le serveur maître. Ce dernier génère du coup des journaux de transactions qui sont transférés vers le serveur en Warm Standby. PostgreSQL sur ce serveur exécute pg_standby pour récupérer le prochain journal.

Comment archiver les journaux de réplication

Cette opération est particulièrement simple. Elle nécessite la configuration d'au maximum trois paramètres de PostgreSQL. Il s'agit des paramètres suivants : archive_mode, archive_command et archive_timeout.

Le premier, archive_mode, apparaît en 8.3. Il permet d'activer ou non l'archivage. Attention au fait que, si vous activez l'archivage, vous devez avoir configuré archive_command sous peine de voir des tas de messages d'avertissements dans vos journaux applicatifs. De plus, le changement de cette variable nécessite un redémarrage du serveur. Pour les versions antérieures à la 8.3, l'archivage n'est activée que si le paramètre archive_command est configuré. Passons donc à ce paramètre.

archive_command est une chaîne de caractère. Cette chaîne est une commande que PostgreSQL exécutera dès qu'un journal de transactions sera prêt à être archivé. Cette commande peut contenir deux caractères joker : %p, chemin complet vers le journal de transactions à archiver, et %f, nom que doit avoir le journal de transactions une fois archivé. Généralement, scp est utilisé pour profiter d'une copie sécurisée ne nécessitant pas la saisie d'un mot de passe. Cependant, vous pouvez utiliser l'outil de copie de votre préférence : cp, ftp, rsync, etc. cp est utilisable, par exemple avec un montage NFS ou Samba, car il est essentiel de copier les journaux sur un autre serveur. Notez que cette commande peut aussi être un script ou un binaire. L'important est qu'il renvoie 0 quand il a pu archiver le journal et autre chose quand une erreur est survenue. Nous allons montrer l'utilisation de scp dans un premier temps.

Dernier paramètre, archive_timeout, disponible depuis PostgreSQL 8.2. Si l'activité du serveur n'est pas énorme, mais qu'on souhaite malgré tout archiver les journaux de transactions assez fréquemment, ce paramètre fait en sorte que PostgreSQL change de journal de transactions une fois le délai indiqué écoulé. Du coup, l'ancien est prêt à être archivé. Néanmoins, attention, cela ne veut pas dire pour autant qu'il sera archivé immédiatement. Une opération de CHECKPOINT doit avoir eu lieu. Avec un checkpoint_timeout bien supérieur au archive_timeout, l'archivage ne sera pas fait à chaque archive_timeout, mais plutôt à chaque checkpoint_timeout. Maintenant que nous avons vu la théorie, passons un peu à la pratique. Le serveur debian1 est le maître et doit envoyer (archiver) ses journaux de transactions sur le serveur debian2. Nous allons utiliser scp pour copier les journaux du serveur debian1 au serveur debian2. Utiliser scp nécessite de saisir un mot de passe ou d'utiliser le système de clé privé/clé publique sans phrase de passe. Nous allons passer par ce dernier. Les clés générés sont à faire pour l'utilisateur postgres car c'est cet utilisateur qui exécutera la commande scp.

Ajoutons un mot de passe à l'utilisateur postgres:

```
debian1:~$ passwd postgres
Entrez le nouveau mot de passe Unix :
Retapez le nouveau mot de passe Unix :
passwd : le mot de passe a été mis à jour avec succès
```

Connectons-nous sur debian2 et ajoutons là-aussi un mot de passe pour l'utilisateur postgres:

```
debian1:~$ ssh guillaume@debian2
guillaume@debian2's password:
[... texte de bienvenue …]
guillaume@debian2:$~ su -
Mot de passe :
debian2:~$ passwd postgres
Entrez le nouveau mot de passe Unix :
Retapez le nouveau mot de passe Unix :
passwd : le mot de passe a été mis à jour avec succès
debian2:~$ exit
logout
Connection to debian2 closed.
Connectons-nous à debian1 en tant qu'utilisateur postgres, puis lançons l'outil ssh-keygen pour générer les clés:
debian1:~$ su – postgres
postgres@debian1:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/postgresql/.ssh/id_rsa):
Created directory '/var/lib/postgresql/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/lib/postgresql/.ssh/id_rsa.
Your public key has been saved in /var/lib/postgresql/.ssh/id_rsa.pub.
The key fingerprint is:
3a:c1:6f:4c:ef:5f:2e:c1:0a:ff:ce:d5:70:8d:83:3f postgres@debian1
```

Surtout ne pas saisir de phrase de passe car là-aussi personne ne sera là pour la saisir lors de la copie.

Il ne nous reste plus qu'à copier la clé publique sur le compte postgres du serveur debian2:

```
postgres@debian1:~$ scp ~/.ssh/id_rsa.pub debian2:~/
```

Ensuite, il faut se connecter sur debian2 et ajouter le contenu du fichier envoyé dans le fichier authorized_keys du répertoire `~/.ssh` (en créant ce dernier si nécessaire) :

```
postgres@debian1:~$ ssh debian2
postgres@debian2's password:
[... texte de bienvenue ...]
postgres@debian2:~$ test -d ~/.ssh || mkdir ~/.ssh
postgres@debian2:~$ cat id_rsa.pub >> ~/.ssh/authorized_keys
```

Note : je préfère utiliser la commande ssh-copy-id plutôt que de faire toutes ces opérations. Néanmoins, j'ai subi quelques déboires sur RedHat qui ne m'ont pas permis de l'utiliser. Pour être le plus indépendant par rapport à la distribution, j'ai préféré, dans cet article, ne pas utiliser ssh-copy-id. Cependant, n'hésitez pas à utiliser cette commande après avoir lu sa page man.

Pour tester toute cette configuration, nous nous déconnectons de debian2 et nous essayons de nouveau de nous connecter:

```
postgres@debian2:~$ exit
logout
Connection to debian2 closed.
postgres@debian1:~$ ssh debian2
[... texte de bienvenue ...]
```

Excellent, pas de mot de passe à saisir.

Tant que nous sommes sur debian2, nous allons créer le répertoire qui va contenir les journaux archivés, lui donner le bon propriétaire (postgres) et les bons droits (700, donc lecture, écriture et exécution uniquement par le propriétaire). Pour cela, nous devons tout d'abord être utilisateur root:

```
postgres@debian2:~$ su -
Mot de passe :
debian2:~$ mkdir /var/lib/postgresql/8.4/main.shippedxlog
debian2:~$ chown postgres:postgres /var/lib/postgresql/8.4/main.shippedxlog
debian2:~$ chmod 700 /var/lib/postgresql/8.4/main.shippedxlog
```

Maintenant, nous allons retourner sur debian1 pour changer la configuration de PostgreSQL de cette façon:

```
archive_mode = on
archive_command = 'scp "%p" "debian2:/var/lib/postgresql/8.3/main.shippedxlog/%f"'
```

Les guillemets doubles ne sont pas obligatoires dans notre cas. Néanmoins, si vous avez des espaces dans les noms de répertoires à parcourir pour accéder au répertoire des journaux de transactions, là, ils deviennent essentiels pour que la commande réussisse. Du coup, pour être cohérent et pour éviter tout soucis futur (un de vos collègue qui déplace le répertoire en question à un emplacement dont le chemin contient des espaces), nous plaçons les guillemets doubles.

Nous ne positionnons pas archive_timeout car il n'a pas d'intérêt dans notre cas.

Comme nous avons modifié archive_mode, nous devons redémarrer PostgreSQL. Notez que ce n'est pas le cas si vous modifiez seulement archive_command.

```
debian1:~$ /etc/init.d/postgresql-8.4 restart
```

En se connectant sur debian2, on pourra voir les journaux de transactions arriver. Enfin, pour cela, il faudrait que notre serveur ait une certaine activité. Nous allons en simuler une en créant une base, puis une table, et en peuplant cette dernière.

```
debian1:~$ su – postgres
postgres@debian1:~$ createdb base1
postgres@debian1:~$ psql base1
psql (8.4.0)
Saisissez « help » pour l'aide.

base1=# CREATE TABLE t1 (id integer);
CREATE TABLE
base1=# INSERT INTO t1 SELECT i FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000
```

Maintenant, nous voyons quelques journaux de transactions apparaître dans le répertoire /var/lib/postgresql/8.4/main.shippedxlog du serveur debian2.

```
debian2~$ ll /var/lib/postgresql/8.4/main.shippedxlog
-rw------- 1 postgres postgres 16777216 aoû   1 12:22 000000010000000000000000
-rw------- 1 postgres postgres 16777216 aoû   1 12:22 000000010000000000000001
-rw------- 1 postgres postgres 16777216 aoû   1 12:22 000000010000000000000002
```

L'archivage fonctionne.

Comment les restaurer au fil de l'eau

Les journaux de transactions sont transférés sur debian2, c'est excellent. Mais pas suffisant. Nous avons au moins besoin d'une sauvegarde de base pour être capable de restaurer les journaux de transactions. En effet, les journaux ne contiennent que des enregistrements du type : sur tel fichier, à tel emplacement, modification de x blocs de données (un bloc faisant 8 Ko). Ils contiennent aussi d'autres types d'enregistrements, mais la grande majorité ressemble à celui indiqué précédemment. Pour pouvoir rejouer cette modification, PostgreSQL a besoin d'avoir les fichiers tels qu'ils étaient à un moment de l'archivage des journaux. Peu importe le moment en question si l'archivage des journaux est effectif. En effet, PostgreSQL va enregistrer la position actuelle dans le journal de transactions. Ainsi il saura à partir de quel endroit il doit rejouer le contenu des journaux de transactions. Tout ce processus est ce qu'on appelle une sauvegarde de base: une simple copie des fichiers de la base de données.

Pour commencer, nous exécutons une fonction SQL, pg_start_backup. Cette dernière demande à PostgreSQL de réaliser quelques actions : s'assurer que les données en cache soient bien enregistrées sur les fichiers de données, puis enregistrer cet état dans les journaux de transactions (cette opération est appelée un CHECKPOINT), et enfin créer un fichier nommé backup_label indiquant l'heure à laquelle a été exécutée cette fonction, le journal de transactions en cours d'utilisation, et l'emplacement où retrouver le CHECKPOINT dans ce journal. Dernier point important, la version 8.4 dispose de deux fonctions pg_start_backup(). La fonction standard et une fonction supplémentaire qui accepte un paramètre de plus pour forcer l'exécution immédiate du CHECKPOINT. En effet, pour les versions antérieures, le CHECKPOINT n'est pas forcé. Il peut donc survenir au bout du délai de checkpoint_timeout (5 minutes par défaut).

Cela étant fait, nous pouvons commencer la copie des fichiers. Un simple tar suffit. Les utilisateurs peuvent utiliser la base de données comme d'habitude. Aucune restriction n'est faite pendant la copie des fichiers. Il est donc probable que certains changeront pendant la sauvegarde, ce qui pourrait occasionner des messages d'avertissements de la part de l'outil de copie que vous utilisez. Peu importe. PostgreSQL sait à partir de quand rejouer les enregistrements compris dans les journaux de transactions grâce au fichier backup_label. Du coup, la copie peut prendre beaucoup de temps. Au niveau cohérence des données, cela n'a pas d'impact.

Enfin, nous devons indiquer à PostgreSQL la fin des opérations. Pour cela, nous utilisons la fonction SQL `pg_stop_backup()`.

Passons de nouveau à un peu de pratique. Exécutons la fonction pg_start_backup() en tant qu'utilisateur Unix postgres:

```
postgres@debian1:~$ psql -c "SELECT pg_start_backup('label')" postgres
 pg_start_backup
-----------------
 0/4000020
```

Un fichier backup_label a bien été créé:

```
postgres@debian1:~$ cat /var/lib/postgresql/8.4/main/backup_label
START WAL LOCATION: 0/4000020 (file 000000010000000000000004)
CHECKPOINT LOCATION:  0/4000020
START TIME: 2009-08-01 12:32:58 CEST
LABEL: label
```

Son contenu donne les informations dont nous avons déjà parlé. Maintenant, sauvegardons tous les fichiers dans une archive tar:

```
postgres@debian1:~$ cd /var/lib/postgresql/8.4
postgres@debian1:~$ tar cfj /tmp/main.tar.bz2 main
```

Enfin, nous pouvons exécuter la dernière fonction:

```
postgres@debian1:~$ psql -c "SELECT pg_stop_backup()" postgres
 pg_stop_backup
----------------
 0/4000080
```

Maintenant que nous avons obtenu la sauvegarde de base, nous allons la restaurer sur le serveur secondaire pour pouvoir placer ce dernier en Warm Standby. Ensuite nous allons créer un fichier appelé recovery.conf qui permet de paramétrer la restauration. Dans notre cas, nous n'avons besoin que d'un paramètre, restore_command, qui permet d'indiquer à PostgreSQL la commande à exécuter pour récupérer un journal de transactions archivés. Dans le mode Warm Standby, cette commande est supposée attendre l'arrivée d'un journal de transactions, et ne le fournir à PostgreSQL que lorsqu'il détecte sa présence et seulement une fois qu'il fait 16 Mo. En effet, un journal de transactions fait forcément 16 Mo, sauf si le transfert ne s'est pas bien fait (coupure réseau pendant la copie, manque d'espace disque, etc.) Il existe deux outils principalement utilisés pour cela : pg_standby (outil fourni dans les modules contrib, donc maintenu par les développeurs PostgreSQL) et walmgr (outil codé par les développeurs de Skype). Nous utiliserons le premier.

Il nous faut tout d'abord copier le fichier /tmp/main.tar.bz2 du serveur debian1 sur le serveur debian2:

```
postgres@debian1:~$ scp /tmp/main.tar.bz2 debian2:~/
```

Connectons-nous au serveur debian2 en tant qu'utilisateur root, plaçons-nous dans le répertoire /var/lib/postgresql/8.4, arrêtons PostgreSQL et supprimons l'ancien répertoire main (vous pouvez aussi le renommer si vous préférez):

```
postgres@debian1:~$ ssh debian2
[... texte de bienvenue ...]
postgres@debian2:~$ su – root
Mot de passe :
debian2:~$ /etc/init.d/postgresql-8.4 stop
debian2:~$ cd /var/lib/postgresql/8.4
debian2:/var/lib/postgresql/8.4$ rm -rf main
```

Ceci fait, nous pouvons extraire les fichiers de l'instance PostgreSQL du fichier tar:

```
debian2:/var/lib/postgresql/8.4$ tar xfj ~postgres/main.tar.bz2
```

Nous pouvons maintenant créer le fichier recovery.conf dans le répertoire main. Vous avez deux possibilités : soit le créer à partir de rien, soit copier le modèle recovery.conf.sample disponible dans /usr/share/postgresql/8.4. Nous allons faire sans, vu qu'il faut simplement y placer une ligne:

```
debian2:/var/lib/postgresql/8.4$ cat <<_EOF_ >>main/recovery.conf
> restore_command = '/usr/lib/postgresql/8.4/bin/pg_standby -d -t /tmp/stopstandby /var/lib/postgresql/8.4/main.shippedxlog %f %p %r >>/var/log/postgresql/pg_standby.log 2>&1'
> _EOF_
```

Cette ligne de commande indique que pg_standby doit attendre le journal %f dans le répertoire /var/lib/postgresql/8.4/main.shippedxlog. À l'arrivée de ce journal complet, il doit le copier dans %p. %r est le nom du journal de transactions au début de la restauration. Cela permet à pg_standby de savoir quels journaux sont inutiles (tout ceux avant ce journal) et lesquels doivent être conservés. Au cas où il détecte la présence du fichier /tmp/stopstandby (option -t), il doit arrêter son exécution et indiquer à PostgreSQL que le mode Warm Standby est terminée. L'option -d place pg_standby en un mode plus verbeux, d'où la redirection de la sortie standard vers un journal applicatif.

Le chemin indiqué pour pg_standby est spécifique aux distributions Debian. Les autres distributions auront généralement cet outil dans /usr/bin.

Pour que l'utilisateur postgres puisse écrire dans le fichier /var/log/postgresql/pg_standby.log, nous allons tout d'abord le créer avec l'utilisateur root et indiquer l'utilisateur postgres comme propriétaire:

```
debian2:/var/lib/postgresql/8.4$ touch /var/log/postgresql/pg_standby.log
debian2:/var/lib/postgresql/8.4$ chown postgres:postgres /var/log/postgresql/pg_standby.log
debian2:/var/lib/postgresql/8.4$ chmod 600 /var/log/postgresql/pg_standby.log
```

Pour utiliser pg_standby, il nous faudra les modules contribs de PostgreSQL 8.4. Comme nous ne les avons pas installé, nous devons le faire maintenant:

```
debian2:/var/lib/postgresql/8.4$ aptitude install postgresql-contrib-8.4
```

Parfait. Normalement, l'utilisateur et les droits doivent être bons suite à l'extraction des fichiers de l'archive tar. Par contre, le nouveau fichier, recovery.conf, a pour propriéaire root. De toute façon, il est préférable de s'assurer que tous les fichiers et répertoires ont le bon propriétaire et les bons droits. Voici les trois commandes à exécuter pour tout restaurer de ce côté:

```
debian2:/var/lib/postgresql/8.4$ chown -R postgres:postgres main
debian2:/var/lib/postgresql/8.4$ find main -type d -exec chmod 700 '{}' ';'
debian2:/var/lib/postgresql/8.4$ find main -type f -exec chmod 600 '{}' ';'
```

Il ne nous reste plus qu'à démarrer PostgreSQL pour que ce dernier entre en mode Warm Standby:

```
debian2:/var/lib/postgresql/8.4$ /etc/init.d/postgresql-8.4 start
```

Au démarrage, PostgreSQL détecte la présence du fichier recovery.conf, le lit pour récupérer le paramètre restore_command et commence la restauration. Il intègre les journaux de transactions déjà utilisés depuis la sauvegarde de base. En retournant générer de l'activité sur debian1 (le même INSERT que plus haut), les journaux applicatifs nous indiqueront l'arrivée des nouveaux journaux de transactions et leur restauration.

Néanmoins, ce n'est pas le seul moyen de regarder le travail en cours. Un autre moyen est de regarder les processus postgres, notamment le processus « startup process » pour ceux qui ont la chance d'être en 8.4 :

```
debian2:/var/lib/postgresql/8.4$ ps -ef | grep "startup process"
postgres 12196 12195  0 13:47 ?        00:00:03 postgres: startup process   waiting for 000000010000000000000008
```

Il est à noter que pg_standby est bien en attente de ce fichier. Nous le voyons de deux façons. Tout d'abord, en recherchant le processus pg_standby dans la liste des processus en cours d'exécution :

```
debian2:/var/lib/postgresql/8.4$ ps -ef | grep pg_standby
postgres 12283 12282  0 13:50 ?        00:00:00 /usr/lib/postgresql/8.4/bin/pg_standby -d -t /tmp/stopstandby /var/lib/postgresql/8.4/main.shippedxlog 000000010000000000000008 pg_xlog/RECOVERYXLOG 000000010000000000000004
```

Et nous devrions avoir en plus une trace dans le fichier /var/log/postgresql/pg_standby.log :

```
debian2:/var/lib/postgresql/8.4$ grep "Waiting for WAL file" /var/log/postgresql/pg_standby.log
Waiting for WAL file    : 00000001.history
Waiting for WAL file    : 000000010000000000000004.00000020.backup
Waiting for WAL file    : 000000010000000000000004
Waiting for WAL file    : 000000010000000000000005
Waiting for WAL file    : 000000010000000000000006
Waiting for WAL file    : 000000010000000000000007
Waiting for WAL file    : 000000010000000000000008
```

Il est donc bien en train d'attendre le journal 000000010000000000000008.

Pour arrêter la restauration, il suffit de créer le fichier /tmp/stopstandby:

```
postgres@debian2:~$ touch /tmp/stopstandby
```

Ceci fait, le serveur doit devenir disponible aux connexions. La restauration est interrompue, de façon permanente. Pour la remettre en place, il faut de nouveau restaurer la sauvegarde de base et rejouer tous les journaux disponibles depuis.

Automatisons un peu tout ça

La mise en place d'un serveur en Log Shipping et d'un autre en Warm Standby se révèle assez simple mais demande l'exécution d'un certain nombre d'actions dans le bon ordre. Il se révèle assez simple d'en oublier une ou d'en inverser deux. Du coup, deux sociétés ont développé des outils permettant l'automatisation d'un maximum de tâches.

Pour la partie pratique qui suit, nous reprenons tout à zéro. Pour l'écriture de cet article, j'ai simplement restauré la copie que j'avais de mes deux machines virtuelles. Les paquets PostgreSQL de base sont installés, les clés privés/publiques aussi.

La solution Command Prompt : pitrtools

Command Prompt (http://www.commandprompt.com/) est une société américaine spécialisée dans PostgreSQL. Elle crée ses propres outils, et notamment pitrtools dont le but est de faciliter la mise en place d'un serveur en Warm Standby. Voici les différentes étapes à suivre.

Tout d'abord, il faut se connecter sur le serveur maître (debian1 dans notre cas). Pitrtools dispose d'un paquet Debian pour sid (http://packages.debian.org/fr/squeeze/pitrtools), c'est celui-là que nous allons télécharger. À noter que je n'ai pas trouvé de paquet RPM pour cet outil. Cependant, le fichier spec est disponible dans les sources de pitrtools. Il est donc possible de créer facilement son propre paquet RPM.

```
debian1:~$ wget -q http://ftp.fr.debian.org/debian/pool/main/p/pitrtools/pitrtools_1.2-1_all.deb
```

Puis nous l'installons avec dpkg:

```
debian1:~$ dpkg -i pitrtools_1.2-1_all.deb
Sélection du paquet pitrtools précédemment désélectionné.
(Lecture de la base de données... 279305 fichiers et répertoires déjà installés.)
Dépaquetage de pitrtools (à partir de pitrtools_1.2-1_all.deb) ...
Paramétrage de pitrtools (1.2-1) ...
```

Comme pitrtools fonctionne uniquement avec rsync et ssh, nous devons installer rsync (si ce n'est pas déjà fait):

```
debian1:~$ aptitude install rsync
```

Sur debian 1, plutôt que de faire un simple scp, nous allons utiliser l'outil cmd_archiver. Ce dernier a pour but d'envoyer les journaux de transactions sur le serveur qui sera en mode Warm Standby. Il nous faut donc configurer cet outil et PostgreSQL. Commençons par l'outil. Le plus simple est de copier le fichier exemple (qui se trouve être /usr/share/doc/pitrtools/cmd_archiver.ini.sample) dans le répertoire de configuration de PostgreSQL, puis de le modifier.

```
debian1:~$ cd /etc/postgresql/8.4/main
debian1:~$ cp /usr/share/doc/pitrtools/cmd_archiver.ini.sample cmd_archiver.ini
debian1:~$ chown postgres:postgres cmd_archiver.ini
debian1:~$ chmod 600 cmd_archiver.ini
```

Ensuite, nous devons le modifier pour obtenir la configuration suivante :

```
[DEFAULT]
; online or offline
state: online

; The base database directory
pgdata: /var/lib/postgresql/8.4/main

; where to remotely copy archives
r_archivedir: /var/lib/postgresql/8.4/main.shipped_xlog

; where to locally copy archives
l_archivedir: /var/lib/postgresql/8.4/main.shipped_xlog

; where is rsync
rsync_bin: /usr/bin/rsync

; option 2 or 3, if running RHEL5 or similar it is likely 2
; if you are running something that ships remotely modern software
; it will be 3
rsync_version = 2

; IP of slave
slaves: 192.168.0.7

; the user that will be using scp
user: postgres

; if scp can't connect in 10 seconds error
timeout: 10

; command to process in ok
notify_ok: echo OK

; command to process in warning
notify_warning:  echo WARNING

; command to process in critical
notify_critical: echo CRITICAL

; if you want to debug on/off only
debug: on

; if you want ssh debug (warning noisy)
ssh_debug: off
```

Nous indiquons principalement le répertoire des données du serveur maître et les répertoires d'archivage sur le maître et sur l'esclave. Le répertoire du maître est utilisé si le ou les esclaves ne sont pas disponibles pour la copie. Il faut aussi donner la liste des esclaves intéressés par les journaux de transactions. En effet, le paramètre slaves est au pluriel, ce qui veut dire que vous pouvez indiquer plusieurs adresses IP pour ce paramètre. Dans ce cas, les journaux seront copiés à chacune des adresses.

Maintenant, nous devons configurer PostgreSQL. Auparavant, le paramètre archive_command faisait appel à scp. Avec pitrtools, il doit faire appel à cmd_archiver. Voici le nouveau paramétrage:

```
archive_command = 'cmd_archiver -C /etc/postgresql/8.4/main/cmd_archiver.ini -F %p'
```

L'option -C permet d'indiquer l'emplacement du fichier de configuration. L'option -F indique le mode d'action de cmd_archiver. Dans ce cas, il s'agit de pousser (Flush) les journaux de transactions vers les esclaves. Avant de recharger la configuration de PostgreSQL (ou de redémarrer PostgreSQL si vous avez dû activer archive_mode), nous devons exécuter cmd_archiver en mode initialisation:

```
debian1:~$ cmd_archiver -C /etc/postgresql/8.4/main/cmd_archiver.ini -I
```

Cette initialisation crée un répertoire par esclaves précisés dans le paramètre slaves. Le nom du répertoire de chaque esclave correspond à son adresse IP. Ils sont créés dans le répertoire pointé par le paramètre l_archivedir (donc /var/lib/postgresql/8.4/main.shipped_xlog dans notre cas). Ces répertoires sont appelés des queues. En cas d'impossibilité d'archivage d'un journal de transactions vers un esclave, ce journal est copié dans cette queue.

Nous pouvons enfin recharger la configuration de PostgreSQL:

```
debian1:~$ /etc/init.d/postgresql-8.4 reload
```

En nous connectant sur l'esclave, nous pouvons voir les nouveaux journaux de transactions arriver. Il nous reste à configurer le serveur debian2 en Warm Standby. Sur debian2, nous allons utiliser un autre outil, appelé cmd_standby, qu'il faudra configurer au préalable. Cet outil a pour but de créer la sauvegarde de base sur le serveur maître et de l'installer sur le serveur en Warm Standby.

Commençons avec le serveur debian1. Il faut créer quelques objets dans une base de données qui servira de base de maintenance. Pour cela, nous devons exécuter le script SQL cmd_standby.sql:

```
postgres@debian1:~$ psql -f cmd_standby.sql postgres
```

Nous le faisons ici sur la base postgres, mais il est possible d'en sélectionner une autre. La configuration de cmd_standby précise justement la base concernée. Continuons par l'installation de pitrtools sur debian2:

```
debian2:~$ wget -q http://ftp.fr.debian.org/debian/pool/main/p/pitrtools/pitrtools_1.2-1_all.deb
debian2:~$ dpkg -i pitrtools_1.2-1_all.deb
Sélection du paquet pitrtools précédemment désélectionné.
(Lecture de la base de données... 279305 fichiers et répertoires déjà installés.)
Dépaquetage de pitrtools (à partir de pitrtools_1.2-1_all.deb) ...
Paramétrage de pitrtools (1.2-1) ...
```

Nous allons aussi copier le fichier exemple (qui se trouve être /usr/share/doc/pitrtools/cmd_standby.ini.sample) dans le répertoire de configuration de PostgreSQL.

```
debian2:~$ cd /etc/postgresql/8.4/main
debian2:~$ cp /usr/share/doc/pitrtools/cmd_standby.ini.sample cmd_standby.ini
debian2:~$ chown postgres:postgres cmd_standby.ini
debian2:~$ chmod 600 cmd_standby.ini
```

Puis, nous le modifions pour obtenir la configuration suivante :

```
[DEFAULT]
; what major version are we using?
pgversion: 8.4

; Used for 8.2 (8.1?), should be set to something > than checkpoint_segments on master
numarchives: 10

; Commands needed for execution
; absolute path to ssh
ssh: /usr/bin/ssh

; absolute path to rsync
rsync: /usr/bin/rsync

; the path to to the postgres bin
pg_standby: /usr/lib/postgresql/8.4/bin/pg_standby
pg_ctl: /usr/lib/postgresql/8.4/bin/pg_ctl

; path to psql on the master
r_psql: /usr/lib/postgresql/8.4/bin/psql

; Generalized information

; the port postgresql runs on (master)
port: 5432

; ip or name of master server
master_public_ip: 192.168.0.10

; the ip address we should use when processing remote shell
master_local_ip: 127.0.0.1

; the user performed initdb
user: postgres

; on or off
debug: off

; the timeout for ssh before we throw an alarm
ssh_timeout: 30

; should be the same as r_archivedir for archiver
archivedir: /var/lib/postgresql/8.4/main.shippedxlog

; where you executed initdb -D to
pgdata: /var/lib/postgresql/8.4/main

; Confs

; This is the postgresql.conf to be used when not in standby
postgresql_conf: /etc/postgresql/8.4/main/postgresql.conf

; This is the pg_hba.conf to be used when not in standby
pg_hba_conf: /etc/postgresql/8.4/main/pg_hba.conf

; Alarms

notify_critical: ls
notify_warning:
notify_ok:

; On failover action

; Whatever is placed here will be executed on -FS must return 0

action_failover: /var/lib/postgresql/failover.sh
```

Le gros point négatif des pitrtools est qu'ils s'attendent, au moins sur l'esclave, à ce que les fichiers de configuration se trouvent dans le répertoires des données de PostgreSQL. C'est en effet généralement le cas, sauf sous Debian/Ubuntu. Du coup, nous allons créer des liens symboliques des fichiers de configuration dans le répertoire /var/lib/postgresql/8.4/main.

```
debian2:~$ cd /var/lib/postgresql/8.4/main
debian2:~$ ln -sf /etc/init.d/postgresql/8.4/main/postgresql.conf
debian2:~$ ln -sf /etc/init.d/postgresql/8.4/main/pg_hba.conf
```

Avant de recharger la configuration de PostgreSQL (ou de redémarrer PostgreSQL si vous avez dû activer archive_mode), nous devons exécuter cmd_standby en mode initialisation. Attention, ce script, contrairement au précédent, ne doit pas être exécuté par l'utilisateur root.

```
debian2:~$ su – postgres
postgres@debian2:~$ cmd_standby -C /etc/postgresql/8.4/main/cmd_standby.ini -I
```

L'option -C indique une fois encore le nom du fichier de configuration, et l'option -I est le mode d'action que doit adopter la commande. Bref, cette commande permet de vérifier que tous les répertoires de données de PostgreSQL sont créés. Il peut sembler étonnant de parler de plusieurs répertoires de données, mais c'est vraiment le cas. Nous parlons ici du répertoire de données principal et des répertoires des tablespaces.

Ensuite, nous allons lancer la commande pour la sauvegarde de base.

```
postgres@debian2:~$ cmd_standby -C /etc/postgresql/8.4/main/cmd_standby.ini -B
```

Cette dernière se connecte au serveur PostgreSQL distant, exécute un CHECKPOINT, exécute la fonction pg_start_backup, supprime le contenu du répertoire des archives (archivedir, soit /var/lib/postgresql/8.4/main.shippedxlog dans notre cas) et copie les fichiers et répertoires des répertoires de données de PostgreSQL. Certains fichiers et répertoires sont exclus : pg_log, postgresql.conf, pg_hba.conf et postmaster.pid. La copie se fait par l'outil rsync. Enfin, la fonction pg_stop_backup() est exécutée.

Pour activer le mode Warm Standby, il faut lancer de nouveau ce script, mais cette fois avec l'option -S.

```
postgres@debian2:~$ cmd_standby -C /etc/postgresql/8.4/main/cmd_standby.ini -S
```

À partir de là, le mode Warm Standby est enclenché. L'esclave se synchronise avec le maître.

Comme nous l'avions dit au début, pitrtools cache une grosse partie du travail pour ne demander que les opérations strictement nécessaires à l'administrateur.

Mais tout ceci n'est qu'une partie du problème. Vient ensuite la question de la bascule.

Pour le switchover, nous devons envoyer le journal non terminé et les journaux non reçus à l'esclave. Nous utilisons pour cela l'option -f (pour flush, vider) de la commande cmd_archiver:

```
debian1:~$ cmd_archiver -C /etc/postgresql/8.4/main/cmd_archiver.ini -f
```

Les journaux restants à envoyer sont envoyés. Il ne reste plus qu'à basculer l'esclave en maître.

```
postgres@debian2:~$ cmd_standby -F999
```

Pour un failover, nous n'utilisons que cette dernière commande, étant donné que le serveur debian1 n'est plus utilisable.

La solution Skype : walmgr

Skype utilise beaucoup PostgreSQL, ce qui fait qu'elle a développé des outils internes pour améliorer la mise en place ou la surveillance de ces réseaux. Elle a aussi eu le bon goût de rendre public certains de ses outils. Nous allons voir ici l'installation et l'utilisation de walmgr, outil faisant partie des Skytools (http://skytools.projects.postgresql.org/).

Skytools est constitué de deux paquets sous Debian : skytools et skytools-modules. Ce dernier est spécifique à la version installée de PostgreSQL. Il n'existe pas encore à ma connaissance de modules pour 8.4. Nous allons donc utiliser un PostgreSQL version 8.3, et utiliser les modules Skytools de cette version.

```
debian1:~$ wget http://ftp.fr.debian.org/debian/pool/main/s/skytools/skytools_2.1.8-2_i386.deb
debian1:~$ wget http://ftp.fr.debian.org/debian/pool/main/s/skytools/skytools-modules-8.3_2.1.8-2_i386.deb
debian1:~$ aptitude install python-psycopg2
debian1:~$ dpkg -i skytools_2.1.8-2_i386.deb skytools-modules-8.3_2.1.8-2_i386.deb
[… différents messages de progression …]
```

Nous faisons la même chose sur le serveur secondaire (debian2).

L'étape suivante est la configuration de wal-master.ini sur debian1 et de wal-slave.ini sur l'esclave. Commençons par le premier en copiant le fichier d'exemple dans /etc/postgresql/8.4/main:

```
debian1:~$ cp /usr/share/doc/skytools/conf/wal-master.ini /etc/postgresql/8.4/main
```

La configuration doit correspondre à celle-ci:

```
[wal-master]
job_name            = debian1_walgmr_master
logfile             = /var/log/postgresql/wal-master.log
use_skylog          = 0

master_db           = dbname=postgres
master_data         = /var/lib/postgresql/8.3/main
master_config       = /etc/postgresql/8.3/main/postgresql.conf

slave               = debian2:/var/lib/postgresql/8.3/main.shippedxlog
slave_config        = /etc/postgresql/8.3/main/wal-slave.ini

completed_wals      = %(slave)s/logs.complete
partial_wals        = %(slave)s/logs.partial
full_backup         = %(slave)s/data.master

# syncdaemon update frequency
loop_delay          = 10.0
# use record based shipping available since 8.2
use_xlog_functions  = 0
# pass -z flag to rsync
compression = 0

# periodic sync
#command_interval    = 600
#periodic_command    = /var/lib/postgresql/walshipping/periodic.sh
```

Rien de particulièrement compliqué. Le plus important est de faire très attention aux chemins.

Continuons en copiant le fichier d'exemple dans /etc/postgresql/8.4/main pour le second (l'esclave):

```
debian2:~$ cp /usr/share/doc/skytools/conf/wal-slave.ini /etc/postgresql/8.4/main
```

La configuration doit correspondre à celle-ci:

```
[wal-slave]
job_name            = debian2_walmgr_slave
logfile             = /var/log/postgresql/wal-slave.log
use_skylog          = 1

slave_data          = /var/lib/postgresql/8.3/main
slave_bin           = /usr/lib/postgresql/8.3/bin
slave_stop_cmd      = /etc/init.d/postgresql-8.3 stop
slave_start_cmd     = /etc/init.d/postgresql-8.3 start

slave               = /var/lib/postgresql/8.3/main.shippedxlog
completed_wals      = %(slave)s/logs.complete
partial_wals        = %(slave)s/logs.partial
full_backup         = %(slave)s/data.master

keep_backups        = 0
archive_command =
```

Là non plus, rien d'extravagant.

Maintenant que la configuration est faite, n'oublions pas de rendre /var/lib/postgresql/8.3 modifiable par l'utilisateur postgres. Nous allons tout simplement modifier le propriétaire par postgres:

```
debian1:~$ chown postgres:postgres /var/lib/postgresql/8.3
debian2:~$ chown postgres:postgres /var/lib/postgresql/8.3
```

Nous allons enfin utiliser l'outil walmgr. Nous allons tout d'abord lui demander de configurer la réplication:

```
postgres@debian1:~$ walmgr /etc/postgresql/8.3/main/wal-master.ini setup
```

Cette étape de configuration consiste en la modification du paramètre archive_command dans le fichier /etc/postgresql/8.3/main/postgresql.conf. archive_mode n'est pas modifié, il vous est conseillé de le modifier immédiatement après cette étape. Ensuite, il crée le répertoire /var/lib/postgresql/8.3/main.shippedxlog sur le serveur debian2, ainsi que les sous-répertoires data.master (il contiendra la copie de base des fichiers de debian1), logs.complete (il contiendra les journaux une fois terminés et archivés) et logs.partial (il contiendra les journaux non terminés).

Nous pouvons maintenant copier les fichiers pour la sauvegarde de base:

```
postgres@debian1:~$ walmgr /etc/postgresql/8.3/main/wal-master.ini backup
```

Attention, si vous avez oublié d'activer archive_mode, vous aurez un message d'erreur ici.

Cette commande commence par créer un fichier verrou (/var/lib/postgresql/8.3/main.shippedxlog/BACKUPLOCK) sur debian2, exécute la fonction SQL pg_start_backup() sur la base de données indiquée par le paramètre DSN master_db, lance un rsync du répertoire /var/lib/postgresql/8.3/main sur debian1 avec le répertoire /var/lib/postgresql/8.3/main.shippedxlog/data.master sur debian2, puis fait un pg_stop_backup et supprime enfin le fichier verrou. Attention si cette étape échoue (par exemple si rsync n'est pas installé sur le maître et/ou l'esclave), vous devrez supprimer manuellement le fichier verrou créé sur debian2.

Une fois connecté sur l'esclave, il est possible de lancer le mode Warm Standby:

```
postgres@debian2:~$ walmgr /etc/postgresql/8.3/main/wal-slave.ini restore
```

Cette étape arrête le serveur PostgreSQL, renomme l'ancien répertoire des données, déplace /var/lib/postgresql/8.3/main.shippedxlog/data.master dans /var/lib/postgresql/8.3/main, crée le fichier de configuration recovery.conf, puis relance PostgreSQL.

La synchronisation est en cours. Les journaux sont restaurés dès leur arrivée.

Pour arrêter le mode Warm Standby, il suffit d'exécuter la commande:

```
postgres@debian2:~$ walmgr /etc/postgresql/8.3/main/wal-slave.ini boot
```

Si le serveur maître est toujours disponible, pensez avant à synchroniser le dernier journal de transactions, même s'il est partiellement rempli, grâce à cette commande:

```
postgres@debian2:~$ walmgr /etc/postgresql/8.3/main/wal-slave.ini sync
```

Un petit récapitulatif sur cette réplication

Avantages majeurs

- simple à mettre en place
- des outils pour simplifier encore plus la mise en place

Inconvénients majeurs

- pas de granularité sur les objets à répliquer
- au pire un journal perdu… ce qui peut représenter beaucoup de requêtes SQL
- les esclaves inaccessibles pendant la restauration

Conclusion

L'intérêt principal du Log Shipping est son intégration au cœur de PostgreSQL. Son autre intérêt est sa grande facilité de mise en place. Cela en fait un choix très intéressant, si ce n'était son gros défaut: les esclaves ne sont pas disponibles, y compris en lecture seule.

Quant aux outils de simplification de l'installation, j'avoue que je partais avec un à-priori assez défavorable pour la solution de Skype. Néanmoins, certaines nouveautés, depuis la version que j'avais un peu testé au tout début, sont très intéressantes. Notamment l'envoi de journaux non terminés, ce qui permet de s'assurer d'une restauration au plus près. D'un autre côté, j'aime beaucoup le rsync excluant certains fichiers pour pitrtools indiquant une attention aux détails qui donne confiance dans le reste du script. Du coup, je n'en conseillerais pas une avec force et conviction, mais j'avoue une préférence pour walmgr.