DALIBO
L'Expertise PostgreSQL

Note Technique

Installation du « Log Shipping »

Auteur : Guillaume Lelarge <guillaume.lelarge@dalibo.com>

Version : 1.3 du 6 juin 2008

Type : Note Technique

Résumé

Présentation du « Log Shipping »

Historique des révisions

Version	Date	Libellé
1.4	08/07/2008	Corrections de Jean-Christophe Arnu
1.3	06/06/2008	Dernière relecture avant libération
1.2	16/05/2008	Ajout de différentes informations, et quelques corrections
1.1	26/09/2007	Correction de l’explication détaillée de `pg_standby` Ajout d’un chapitre d’explications détaillées sur PITR
1.0	11/05/2007	Version initiale

Table des matières

1 Introduction

La version 8.2 est la première version à proposer enfin la fonctionnalité du `Warm Standby`, aussi connu sous le nom de `Log Shipping`. Le concept est simple : rejouer immédiatement les journaux de transaction archivés pour avoir un serveur en attente, prêt à prendre la main en cas de perte du serveur principal.

2 Explications techniques

2.1 Gestion habituelle des journaux de transactions

Le serveur PostgreSQL enregistre chaque modification (ajout, suppression ou mise à jour) dans des journaux appelés « journaux de transactions ». Ces fichiers sont stockés dans le répertoire `$PGDATA/pg_xlog`. Au bout d'un certain laps de temps mais aussi dans certains cas précis (comme l'exécution manuelle de la commande `CHECKPOINT`, le dépassement d'un certain nombre de journaux de transactions remplis sans `CHECKPOINT`, ou l'arrêt du serveur), PostgreSQL enregistrera les pages modifiées du cache disque de PostgreSQL dans les fichiers représentant les tables et index de la base de données.

PostgreSQL crée autant de journaux que nécessaire sachant qu'un journal peut être réutilisé une fois que son contenu n'est plus utile (autrement dit une fois que toutes les informations qu'il contient ont été répercutées dans les fichiers représentant les tables et index). Cela permet d'éviter d'avoir à supprimer le journal pour récréer un autre fichier de même taille, donc moins d'entrées/sorties disque.

2.2 Fonctionnement de PITR

La version 8.1 est la première version à proposer la fonctionnalité du PITR. Cette fonctionnalité propose l'archivage des journaux de transactions par copie de ces fichiers sur un répertoire local ou distant. Le gros intérêt du PITR est de pouvoir rejouer ces journaux de transactions. Il existe deux exemples typiques d'utilisation de l'archivage :

- Le serveur maître devient indisponible quelqu'en soit la raison... il suffit de restaurer la sauvegarde de base sur un nouveau serveur, d'y placer les journaux de transactions archivés, de créer un fichier de configuration pour une restauration complète, puis de lancer le serveur.
- Un utilisateur, ou plus vraisemblablement un administrateur, a supprimé par erreur un objet (colonne, table, tablespace) ou des données, et souhaite revenir au moment juste avant cette suppression. Là aussi, il suffit de restaurer la sauvegarde de base sur un nouveau serveur, d'y placer les journaux de transactions archivés, de créer un fichier de configuration pour une restauration partielle (histoire de ne pas récupérer la commande qui a réalisée la suppression), puis de lancer le serveur.

L'archivage des journaux s'active en configurant le paramètre `archive_command`.

Note : à partir de la version 8.3, `archive_mode` doit aussi être activé.

À partir de ce moment, dès qu'un journal est considéré comme archivable par PostgreSQL (parce que tout son contenu a été répercuté dans les fichiers de données soit après la fin de son remplissage soit après expiration du délai `archive_timeout`[1]), un fichier est créé dans le sous-répertoire `$PGDATA/pg_xlog/archive_status`. Ce fichier aura le nom du journal de transaction avec le suffixe `.ready`. La commande contenue par le paramètre `archive_command` sera exécutée via l'appel C `system()`. Suivant le code de retour, PostgreSQL sait si le journal a bien été archivé. S'il n'a pas été archivé, PostgreSQL place un message d'alerte dans les journaux applicatifs (« `archive command "la_commande" failed: return code code_de_retour` ») puis tente une deuxième fois au bout d'une seconde. Si la commande échoue encore, il tente une dernière fois toujours après une seconde. Après cela, il affiche un nouveau message d'alerte (« `transaction log file "journal_de_transaction" could not be archived: too many failures` ») et abandonne l'exécution de la commande pendant une minute. Ces deux délais ne sont pas paramétrables. Tant que le journal n'est pas archivé, il n'est ni remplacé ni ré-utilisé, ce qui peut poser des soucis d'espace disque. Il faut donc prévoir de bien surveiller les journaux applicatifs de PostgreSQL, voire de disposer d'un outil de vérification de la place disponible et d'alertes (comme Nagios par exemple).

Si la commande s'exécute correctement (en renvoyant un code d'erreur `0`), PostgreSQL marque le fichier de suffixe `.ready` comme étant archivé en le renommant avec un suffixe `.done`. Le journal est donc ré-utilisable et sera renommé pour cela.

La restauration d'un serveur PITR demande de restaurer dans un premier temps la sauvegarde de base, puis de créer un fichier de configuration de la restauration nommé `recovery.conf` dans le répertoire `$PGDATA`. Une commande doit être spécifiée pour le paramètre `restore_command`. Dans le cas d'une restauration partielle (voir l'exemple 2 ci-dessus), un autre paramètre doit être renseigné : `recovery_target_time`. Il est ainsi possible de préciser une date et une heure quelques secondes avant l'heure fatidique. Il existe aussi un autre paramètre, `recovery_target_xid`, permettant de spécifier le dernier XID à récupérer. Cependant, aucun outil ne permet à l'heure actuelle d'associer un XID à une instruction/action particulière.

Une fois la configuration effectuée, il ne reste plus qu'à lancer le serveur. Celui-ci, découvrant le fichier `recovery.conf`, entre en mode restauration pour rejouer les différents journaux de

[1] `archive_timeout` permet d'archiver un journal de transaction au plus tard après ce délai en secondes.

transactions. Une fois tous les fichiers restaurés, il entre en mode production. Il est disponible pour les utilisateurs...

Mais cette disponibilité n'est pas immédiate. Il a fallu restaurer la sauvegarde, ce qui peut être très long (bien que cela puisse se faire avant le problème). Il faut aussi rejouer les journaux de transactions restaurés. Si la sauvegarde de base date d'hier, cela pourrait être assez rapide. Si elle date de la semaine dernière, cela peut prendre déjà beaucoup plus de temps. Et si elle date du mois dernier...

2.3 Fonctionnement du « Log Shipping »

Le concept du `Log Shipping` est donc simple. Comme la sauvegarde de base est déjà disponible et que les journaux de transactions sont archivés, il suffit de récupérer tout ça. La sauvegarde de base sert à préparer le nouveau serveur, les journaux de transactions archivés sont restaurés dès leur arrivée sur le nouveau serveur.

Dit autrement, le serveur principal fonctionne de la même façon que pour PITR, donc rien à changer de ce côté là. Néanmoins, il faut que le serveur principal archive les journaux de transactions sur le serveur cible. Ce serveur est celui qui sera en attente. Il doit être installé de la même façon que le serveur principal. Matériellement, il est préférable qu'ils soient identiques même si ce n'est pas obligatoire, mais, de toute façon, il n'est pas possible de faire de mix entre architecture 32 bits et architecture 64 bits. Si des tablespaces sont utilisés, il faut bien faire attention à recréer les points de montages nécessaires.

Une fois l'installation du matériel et des logiciels terminés, il faut faire une sauvegarde de base du serveur maître. Cette sauvegarde sera restaurée sur le serveur cible. N'oubliez pas de modifier le fichier de configuration (le paramètre `archive_command` peut être activé mais il ne doit pas copier les fichiers au même endroit que le serveur maître). Il faut ensuite créer le fichier `recovery.conf`. Un seul paramètre est nécessaire : `restore_command`. Cette commande s'occupe de la récupération des journaux de transactions et les fournit au processus de restauration lancé par PostgreSQL. Elle récupère le journal de transaction archivé demandé par PostgreSQL puis rend la main au serveur qui restaurera ce fichier et relancera la commande pour récupérer le journal suivant. Lorsqu'il n'est pas présent, la commande renvoie un code d'erreur qui fait que PostgreSQL se retire du mode restauration pour devenir disponible. Or, dans notre cas, nous ne voulons pas quitter ce mode. Le serveur cible doit rester en permanence en mode de restauration. La commande ne devra pas se terminer si le journal de transactions demandé est absent. Il faut donc soit écrire un script qui attendra l'arrivée d'un nouveau journal de transactions pour le fournir au serveur soit récupérer un programme déjà écrit.

Si le serveur principal est arrêté, le serveur secondaire reste en attente. Ce dernier recevra le prochain journal de transaction après le redémarrage du serveur principal.

De même, il est possible d'arrêter le serveur secondaire sans casser la restauration, mais c'est un peu plus complexe à réaliser.

3

Explications plus détaillées sur `restore_command`

La commande de restauration doit avoir le comportement suivant :

- attente d'un nouveau journal de transaction ou d'un signal d'arrêt
- si le signal est reçu, quitter
- une fois le journal trouvé, attendre qu'il ait la bonne taille (16 Mo)
- une fois à la bonne taille, le copier à l'endroit attendu par le serveur
- recommencer l'attente d'un journal de transactions

Rien n'est prévu en 8.2 pour gérer cela. Il est donc nécessaire d'écrire un script (bash, perl, python, ruby, etc.) ou un programme (C ou autres). En 8.3 apparaît un petit programme bien sympathique qui fait le sujet du chapitre suivant.

3.1 pg_standby

Simon Riggs a écrit en C ce petit programme qui fait partie des modules contrib de la version 8.3. Il a le gros avantage d'inclure une bonne partie des fonctionnalités et des options nécessaires pour gérer à peu près tous les cas du `Log Shipping`.

`pg_standby` s'utilise ainsi :

```
pg_standby [OPTION]... [ARCHIVELOCATION] [NEXTWALFILE] [XLOGFILEPATH]
```

`ARCHIVELOCATION` correspond au répertoire de stockage des journaux de transaction archivés. `NEXTWALFILE` est le nom du prochain journal à récupérer et `XLOGFILEPATH` est l'emplacement des journaux de transaction. Ces deux derniers correspondent respectivement au `%f` et au `%p` du paramètre `restore_command`.

Parmi les options disponibles, voici les plus intéressantes :

- `-c` pour utiliser la commande cp pour la restauration des journaux de transactions.
- `-d` pour envoyer des informations de débogage sur `stderr`.

- `-k nombre_de_fichiers` pour conserver uniquement ce nombre de journaux de transactions. Il est souvent préférable de configurer une valeur importante, voire pas de valeur du tout (ce dernier cas est surtout vrai si vous voulez conserver les archives pour restaurer une autre machine).
- `-l`... il y avait `-c` pour copier, il y a aussi `-l` pour utiliser des liens symboliques. Cette option est plus efficace.
- `-s delai`, pour que le prochain test de vérification du journal de transactions n'intervienne pas avant ce délai (en secondes). Par défaut, 5.
- `-t fichier_trigger`... rien à voir avec les triggers d'une base de données. L'existence de ce fichier sera vérifié. S'il existe, le programme `pg_standby` s'arrêtera, stoppant ainsi la récupération des journaux de transactions archivés. PostgreSQL se retrouvera du coup en mode normal. Il sera en attente de connexion.
- `-w delai_max`, délai maximum avant l'abandon de la récupération. Par défaut à 0, ce qui désactive cette fonctionnalité.

Bien entendu, il existe d'autres options. Elles sont toutes expliquées dans le fichier `README.pg_standby` et dans la documentation officielle de PostgreSQL (http ://docs.postgresqlfr.org/8.3/pgstandby.html).

Utiliser `pg_standby` peut être très simple :

```
restore_command = 'pg_standby /var/pg_xlog_archives %f %p'
```

Dans ce cas, `pg_standby` est appelé par la restauration. Il cherche le fichier dont le nom est précisé par `%f` dans `/var/pg_xlog_archives`. Une fois trouvé, il copie ce fichier dans le répertoire pointé par `%p` et rend la main. PostgreSQL effectue la restauration des données à partir de ce journal de transaction, puis relance `pg_standby` en lui précisant le nom du nouveau journal de transactions à récupérer. Étant donné qu'aucun fichier trigger n'est fourni et qu'aucun délai max n'a été mentionné, le seul moyen d'arrêter en toute sûreté la récupération est d'envoyer un signal `SIGINT` au processus `pg_standby`.

La ligne de commande peut être beaucoup plus complexe :

```
restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5432
 /var/pg_xlog_archives %f %p 2>> standby.log'
```

Dans ce cas, `pg_standby` est appelé par la restauration. Il cherche le fichier dont le nom est précisé par `%f` dans `/var/pg_xlog_archives`. Dès qu'il trouve ce journal de transaction, il vérifie sa taille. S'il n'est pas complet, il attend deux secondes et teste de nouveau jusqu'à ce que ce fichier soit complet. Une fois complet, il copie le fichier en question dans le répertoire indiqué par `%p`. En vérifiant l'arrivée du nouveau journal de transactions archivé, il teste aussi l'existence du fichier `/tmp/pgsql.trigger.5432`. Dès qu'il le détecte, `pg_standby` s'arrête proprement. Plutôt que de créer ce fichier, il est aussi possible d'envoyer le signal indiqué dans l'exemple précédent.

3.2 wal_mgr

`wal_mgr` est un outil écrit en Python par la société Skype pour ces besoins propres. Elle a récemment mis à disposition différents outils sous une licence libre. L'utilisation de cet outil couvre le `Log Shipping` mais aussi le `PITR`.

Contrairement à `pg_standby`, cet outil s'installe sur les deux serveurs, le maître comme l'esclave. Autre différence, la configuration des serveurs PostgreSQL n'a pas besoin d'être modifiée. Suivant le mode d'exécution de `wal_mgr`, l'outil exécutera le serveur PostgreSQL dans un mode ou dans un autre :

- setup (vérification du serveur maître et du serveur esclave) ;
- backup (sauvegarde de base, copie sur l'esclave et lancement du serveur maître) ;
- restore (mise en place de la sauvegarde de base sur l'esclave et lancement du serveur esclave) ;
- sync (synchronisation des journaux de transactions avec l'esclave)
- boot (arrêt du mode de ré-exécution sur l'esclave et lancement du serveur en mode normal).

Voici, en gros, les différentes étapes d'installation :

- lancement du processus d'archivage :

```
maître$ walmgr.py /chemin/vers/master.ini setup
```

- sauvegarde complète :

```
maître$ walmgr.py /chemin/vers/master.ini backup
```

- et enfin restauration :

```
esclave$ walmgr.py /chemin/vers/esclave.ini restore
```

Parfois, il est nécessaire de synchroniser les journaux de transactions. La commande suivante s'en charge :

```
maître$ walmgr.py /chemin/vers/esclave.ini sync
```

Et pour arrêter la re-exécution automatique des journaux de transactions sur le client :

```
esclave$ walmgr.py /chemin/vers/esclave.ini boot
```

Attention, contrairement à ce qu'indique la documentation officielle, il est nécessaire de fournir le fichier configuration comme premier argument de la commande `walmgr.py`.

Quelques liens intéressants :

- https ://developer.skype.com/SkypeGarage/DbProjects/SkyTools
- https ://developer.skype.com/SkypeGarage/DbProjects/SkyTools/WalMgr

4 Un cas complet avec wal_mgr

Il faut tout d'abord configurer le serveur principal :

```
postgres@laptop :/opt/postgresql-8.2$ walmgr.py master.ini setup
2007-06-19 15 :44 :11,246 18245 INFO Configuring WAL archiving
2007-06-
19 15 :44 :11,249 18245 INFO Using config file : /etc/postgresql/8.2/main/postgresql
2007-06-19 15 :44 :11,250 18245 INFO Sending SIGHUP to postmaster
2007-06-19 15 :44 :11,250 18245 INFO Done
```

Ensuite, nous réalisons la sauvegarde de base :

```
postgres@laptop :/opt/postgresql-8.2$ walmgr.py master.ini backup
2007-06-19 15 :45 :30,222 18291 INFO Backup lock obtained.
2007-06-19 15 :45 :30,222 18291 INFO got SystemExit(0), exiting
2007-06-19 15 :45 :30,229 18286 INFO Execute SQL : se-
lect pg_start_backup('FullBackup') ; [dbname=template1]
2007-06-19 15 :45 :31,265 18286 INFO Checking tablespaces
2007-06-19 15 :45 :32,298 18316 INFO First use-
ful WAL file is : 000000010000000000000000
2007-06-19 15 :45 :33,198 18328 INFO Backup lock released.
2007-06-19 15 :45 :33,206 18286 INFO Full backup successful
```

Enfin, il s'agit de l'étape de restauration sur le serveur de secours :

```
guillaume@laptop :/opt/postgresql-8.2-
cible$ walmgr.py slave.ini restore
2007-06-19 15 :48 :32,469 18357 INFO Move /opt/postgresql-8.2-
cible/walshipping/data.master to /opt/postgresql-8.2-cible/data
2007-06-19 15 :48 :32,470 18357 INFO Write /opt/postgresql-8.2-
cible/data/recovery.conf
2007-06-
19 15 :48 :32,470 18357 INFO Starting postmaster : pg_ctl start
```

```
postgres n'a pas pu accéder au fichier de configura-
tion «/opt/postgresql-8.2-cible/data/postgresql.conf» : Aucun fi-
chier ou répertoire de ce type
serveur en cours de démarrage
```

5 Un cas complet avec pg_standby

5.1 Introduction

L'utilisateur a deux serveurs : maitre (qui sera le serveur actif) et cible (qui aura le rôle du serveur en attente).

Toutes les commandes sont à exécuter en tant qu'utilisateur postgres. La variable d'environnement PGDATA doit être positionnée sur les deux serveurs.

Le répertoire `/var/pg_xlog_archives` est le répertoire d'archivage des journaux de transactions. Il doit être présent sur le serveur cible et appartenir à l'utilisateur postgres.

Comme la commande d'archivage passe par `scp`, il faut qu'une clé soit générée puis copiée sur le serveur cible :

- sur le serveur maitre
  - ⇒ ssh_keygen
  - ⇒ scp ~/.ssh/id_rsa.pub postgres@cible :~/
- sur le serveur cible
  - ⇒ mkdir -p ~/.ssh
  - ⇒ cat ~/id_rsa.pub >> ~/.ssh/authorized_keys

5.2 Étapes de la préparation des serveurs

- (initdb si nécessaire sur le premier)
- configuration du serveur maître pour PITR (`$PGDATA/postgresql.conf`)

⇒ archive_command = 'scp %p postgres@cible:/var/pg_xlog_archives/%f'
⇒ archive_timeout = 60

- lancement du serveur maître
  ⇒ pg_ctl start
- sauvegarde de base sur le serveur maître
  ⇒ psql -c "select pg_start_backup('debut')"
  ⇒ tar cvfj data_maitre.tar.bz2 $PGDATA
  ⇒ psql -c "select pg_stop_backup()"
  ⇒ scp data_maitre.tar.bz2 postgres@cible:~/
- restauration de la sauvegarde de base sur le serveur cible
  ⇒ test -d data && mv data data.old
  ⇒ tar xvfj data_maitre.tar.bz2
- configuration de la récupération en attente du serveur cible (`$PGDATA/recovery.conf`)
  ⇒ restore_command = 'pg_standby -d -k 255 -r 2 -s 2 -w 0 \ -t /tmp/pgsql.trigger.5
  /var/pg_xlog_archives %f %p 2>> standby.log'
- lancement du serveur cible
  ⇒ pg_ctl start

5.3 Ce qui se passe après

Le fichier de traces contiendra deux types d'informations. Lorsque `pg_standby` trouve un journal de transaction à restaurer, les traces générées ressembleront à ceci :

```
Trigger file             : /tmp/pgsql.trigger.5442
Waiting for WAL file     : /var/pg_xlog_archives/000000010000000000000000
WAL file path            : 000000010000000000000000
Restoring to...          : pg_xlog/RECOVERYXLOG
Sleep interval           : 2 seconds
Max wait interval        : 0 forever
Command for res-
tore     : cp /var/pg_xlog_archives/000000010000000000000000 pg_xlog/RECOVERYXLOG
Num archived files kept : last 255 files
running restore          : success
```

Tant que `pg_standby` ne trouve rien, il affichera simplement :

```
WAL file not present yet. Checking for trigger file...
WAL file not present yet. Checking for trigger file...
WAL file not present yet. Checking for trigger file...
WAL file not present yet. Checking for trigger file...
```

5.4 Arrêter la récupération

Il suffit de créer le fichier `/tmp/pgsql.trigger.5442` pour arrêter la récupération des journaux de transactions :

```
touch /tmp/pgsql.trigger.5442
```

Envoyer un signal `SIGINT` ou `SIGKILL` aura le même effet :

```
kill -INT pid_de_pg_standby
```

Le fichier de trace se terminera ainsi :

```
WAL file not present yet. Checking for trigger file...
WAL file not present yet. Checking for trig-
ger file...trigger file found

Trigger file            : /tmp/pgsql.trigger.5442
Waiting for WAL file    : /var/pg_xlog_archives/000000010000000000000003
WAL file path           : 000000010000000000000003
Restoring to...         : pg_xlog/RECOVERYXLOG
Sleep interval          : 2 seconds
Max wait interval       : 0 forever
Command for res-
tore     : cp /var/pg_xlog_archives/000000010000000000000003 pg_xlog/RECOVERYXLOG
Num archived files kept : last 255 files
running restore         : success
```

6 Conclusion

Le gros problème pour la mise en place du `Log Shipping` était l'écriture du script d'attente et de copie des journaux de transactions sur l'esclave. Les deux solutions présentées ici y répondent en tout point et conviennent pour tout type d'installation.

Le `Log Shipping` devient donc une solution de réplication basique mais d'une grande efficacité et d'une grande simplicité pour sa mise en oeuvre.