−Table des matières

PostgreSQL et ses journaux de transactions

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 108 (Septembre 2008). Il est disponible maintenant sous licence Creative Commons.

Les journaux de transaction constituent un domaine évoluant en permanence pour PostgreSQL. Auparavant simples journaux des modifications internes, ils ont évolués pour permettre une reprise à un moment précis dans le passé, voire même pour autoriser une réplication simple d'un serveur complet. Mais le plus important est que cela assure de la durabilité des informations stockées sur disque.

Technologie WAL

L'acronyme WAL est souvent utilisé dans les discussions sur les journaux de transaction. Sa signification est simple : Write Ahead Log, un journal qui est écrit avant. En fait, il s'agit de l'algorithme gérant les journaux de transactions sous PostgreSQL.

Lorsque qu'un utilisateur fait une modification dans sa base de données, un ou plusieurs fichiers de données seront modifiés. Ces changements seront tracés dans le journal de transactions en cours d'utilisation au plus tard lors de la validation de la transaction contenant ces changements. Cela peut arriver plus tôt si les changements à ajouter aux journaux de transaction ne tiennent pas dans la mémoire allouée pour cela (voir le paramètre wal_buffers). Ce n'est qu'un peu après que soit bgwriter soit un processus postgres enregistrera les modifications sur les fichiers de données.

Cela veut donc dire que chaque modification est enregistrée deux fois : une fois dans les journaux de transaction, une fois dans les fichiers de données. On pourrait croire à une perte de temps (et donc de performances), mais dans les faits, on gagne en rapidité et en fiabilité.

Commençons par la rapidité. Considérons un utilisateur qui met à jour une table avec une requête UPDATE. Son processus serveur va devoir modifier la ligne touchée par l'UPDATE pour indiquer qu'elle doit être considérée comme une ligne morte à partir de cet identifiant de transaction, il va aussi devoir ajouter la nouvelle ligne vivante, sans compter la mise à jour de la table TOAST (si nécessaire) et des index associés. Il peut y avoir beaucoup d'écritures à des emplacements non contigus sur le disque, donc nécessitant un déplacement de la tête de lecture, donc générateur de lenteur. Or, l'utilisateur a besoin d'avoir rapidement l'information que les données sont enregistrées (le message COMMIT). Du coup, il est préférable dans un premier temps d'écrire séquentiellement dans un fichier toutes les modifications à faire car une écriture séquentielle est bien plus rapide.

On gagne aussi en stabilité. Si le serveur PostgreSQL s'arrête brutalement pendant l'écriture des journaux de transactions, la transaction en cours d'écriture n'est pas validée, du coup il n'y a pas de pertes de données. Supposons maintenant que le crash survient après l'écriture dans les journaux de transactions mais avant l'écriture dans les fichiers de données. Au relancement de PostgreSQL, le serveur commence par relire les journaux de transaction et écrit dans les fichiers de données les données présentes dans les journaux de transactions mais absentes des fichiers de données.

Pour que ce système fonctionne, PostgreSQL doit s'assurer que, quand il enregistre des données dans les journaux de transactions, ces données sont bien sur disque, et pas dans le cache du système d'exploitation. Pour cela, PostgreSQL demande au système de synchroniser les données qui sont dans son cache sur le disque grâce à l'instruction système C fsync. Suivant le système d'exploitation, il est possible de choisir l'appel système chargé de synchroniser les disques par l'intermédiaire du paramètre wal_sync_method. Le paramètre fsync du fichier de configuration postgresql.conf permet de désactiver ce comportement mais, dans ce cas, la sécurité de vos données n'est pas assurée. Il est néanmoins possible de désactiver ce paramètre dans certains cas car le gain en performances (par exemple pendant une grosse insertion ou modification) est très important. C'est intéressant dans certains cas très précis où une corruption de l'instance n'est pas problématique (par exemple, lors du chargement d'une sauvegarde sur un nouveau serveur). Dans tous les autres cas (autrement dit, dans la très grande majorité des cas), le paramètre fsync doit être activé.

Stockage des journaux de transactions

Au moment de l'opération d'initialisation du cluster via initdb, le répertoire pg_xlog est créé. Il est par défaut dans le répertoire de l'instance PostgreSQL. Ce répertoire va contenir tous les journaux de transactions de cette instance. Le « XLOG », partie finale du nom du répertoire, est un autre acronyme souvent utilisé pour parler des journaux de transactions.

Créons une nouvelle instance pour voir ce que contient, tout de suite après l'initialisation de l'instance, le répertoire pg_xlog :

```
guillaume@laptop:~$ export PGDATA=/home/guillaume/glmf
guillaume@laptop:~$ initdb
The files belonging to this database system will be owned by user "guillaume".
[ ... coupé ... ]
Success. You can now start the database server using:

    postgres -D /home/guillaume/glmf
or
    pg_ctl -D /home/guillaume/glmf -l logfile start

guillaume@laptop:~$ ls -lhG $PGDATA/pg_xlog
total 17M
-rw------- 1 guillaume  16M 2008-07-13 16:19 000000010000000000000000
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

Nous discuterons du répertoire archive_status plus tard. Commençons plutôt par le seul fichier actuellement disponible.

000000010000000000000000 est le premier journal de transactions après la création d'une nouvelle instance. Il fait 16 Mo. Son nom est codé : les huit premiers chiffres correspondent au TimeLineID, les suivants résultent d'une séquence. Pour le reste de l'article, je ne mentionnerais que la fin du nom du fichier (1 pour 000000010000000000000001, 2 pour 000000010000000000000002, etc.).

Un journal de transactions fait toujours 16 Mo, quelque soit la quantité d'informations qu'il contient. Il est préalloué sur le disque pour que PostgreSQL ne soit pas confronté à des problèmes d'espace disque. Il est possible de savoir où aura lieu la prochaine écriture sur ce journal de transactions grâce à la fonction pg_current_xlog_location() ;

```
guillaume@laptop:~$ pg_ctl start
server starting
LOG:  database system was shut down at 2008-07-13 16:19:29 CEST
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
guillaume@laptop:~$ psql -q postgres
postgres=# SELECT pg_current_xlog_location();
 pg_current_xlog_location
--------------------------
 0/43F9C8
(1 row)
```

Créons une table :

```
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 17M
-rw------- 1 guillaume  16M 2008-07-13 16:19 000000010000000000000000
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
postgres=# CREATE TABLE t1 (id int4);
postgres=# SELECT pg_current_xlog_location();
 pg_current_xlog_location
--------------------------
 0/43FAC8
(1 row)
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
```

```
total 17M
-rw------- 1 guillaume  16M 2008-07-13 16:20 000000010000000000000000
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

Première point constaté : créer une table ne génère pas suffisament d'informations pour demander la création d'un autre journal de transactions. Par contre, le pointeur d'emplacement courant dans le journal des transactions indique qu'il a été déplacé de la position 0/43F9C8 à la position 0/43FAC8, donc des données ont été enregistrées. Maintenant, ajoutons des données dans la table :

```
postgres=# INSERT INTO t1 VALUES (1);
postgres=# INSERT INTO t1 VALUES (2);
postgres=# INSERT INTO t1 VALUES (3);
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 17M
-rw------- 1 guillaume  16M 2008-07-13 16:21 000000010000000000000000
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

Là-aussi, l'insertion des trois enregistrements continue sur le même journal de transactions. En fait, il faut ajouter beaucoup plus de données pour qu'un autre journal de transactions soit utilisé.

```
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 33M
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000000
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000001
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 49M
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000000
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000001
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000002
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

En continuant cette opération de nombreuses fois, PostgreSQL se met à créer les journaux de transactions dont il a besoin. Nous allons exécuter plusieurs fois la grosse insertion pour arriver à un seuil critique :

```
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
LOG:  checkpoints are occurring too frequently (16 seconds apart)
HINT:  Consider increasing the configuration parameter "checkpoint_segments".
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 113M
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000000
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000001
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000002
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000003
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000004
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000005
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000006
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

Nous avons donc là les journaux de transactions, du 0 au 6. Une dernière insertion :

```
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 129M
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000003
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000004
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000005
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000006
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000007
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000008
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000009
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000A
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

Que constatons-nous ? PostgreSQL semble s'être débarrassé des fichiers 0, 1 et 2 et a créé les fichiers 7 à A. Or nous avions vu que cette insertion générait un seul journal de transactions. En fait, il s'est passé ceci : dû au grand nombre de journaux créés (et du coup au dépassement de la valeur du paramètre checkpoint_segments), PostgreSQL a exécuté un CHECKPOINT automatique. Ce CHECKPOINT a enregistré un certain nombre de pages disques modifiées en mémoire dans les fichiers de données. Du coup, certains des journaux de transactions n'étaient plus nécessaires. Plutôt que d'être supprimés, ils ont été renommés pour être de nouveau utilisables. En fait, PostgreSQL ne travaille pas sur le fichier A actuellement. Pour s'en assurer, utilisons la fonction interne pg_xlogfile_name() :

```
postgres=# SELECT pg_xlogfile_name(pg_current_xlog_location());
     pg_xlogfile_name
--------------------------
 000000010000000000000007
(1 row)
```

Nous en sommes au numéro 7. En fait, nous aurions pu le deviner avec la dernière exécution de la commande ls. Le fichier 7 est le dernier à avoir été modifié (2008-07-13 16:23).

Nous avons passé rapidement le message niveau LOG qui disait « checkpoints are occurring too frequently (16 seconds apart) ». En fait, quand beaucoup de journaux de transactions sont générés, il est possible de faire en sorte que PostgreSQL écrive un message d'avertissement dans les traces. Ce moyen, c'est le paramètre checkpoint_warning. Si PostgreSQL exécute des CHECKPOINT automatiques dans un délai inférieur à la valeur de ce paramètre (en secondes), un message est enregistré dans les traces.

L'autre moyen d'avoir un CHECKPOINT automatique est de laisser passer un certain temps, exactement plus de checkpoint_timeout secondes. Après un tel temps d'attente, voici ce qu'il nous reste comme journaux de transactions :

```
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 129M
-rw------- 1 guillaume  16M 2008-07-13 16:23 000000010000000000000006
-rw------- 1 guillaume  16M 2008-07-13 16:30 000000010000000000000007
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000008
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000009
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000A
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000B
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000C
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000D
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

Suivant l'activité de la base de données, un CHECKPOINT permettra de s'assurer d'avoir des journaux de transactions prêt à être utilisés. PostgreSQL utilise dans le cadre normal (2+checkpoint_completion_target) * checkpoint_segments + 1 journaux de transactions. Dans le pire des cas, vous pourrez en avoir plus. Si vous avez trois fois plus de journaux de transactions que la valeur indiquée dans checkpoint_segments, PostgreSQL les supprimera au lieu de les renommer, ceci pour éviter d'utiliser trop d'espace disque.

Faisons un CHECKPOINT manuel :

```
postgres=# CHECKPOINT;
postgres=# \! ls -lhG /home/guillaume/glmf/pg_xlog
total 129M
-rw------- 1 guillaume  16M 2008-07-13 16:32 000000010000000000000007
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000008
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000009
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000A
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000B
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000C
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000D
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000E
drwx------ 2 guillaume 4,0K 2008-07-13 16:19 archive_status
```

On se retrouve pratiquement dans le même cas qu'au tout début. Nous travaillons sur le premier journal (le 7), mais nous avons aussi les sept autres journaux prêts à être utilisés.

Pour résumé, que nous indiquent toutes ces manipulations ?

1. PostgreSQL écrit fréquemment dans les journaux de transactions ;
2. Il crée les journaux quand il en a besoin s'ils n'existent pas déjà ;
3. Dans le meilleur (et le plus fréquent) des cas, il réutilise les anciens journaux pour éviter d'avoir à créer des fichiers vides de 16 Mo ;
4. Chaque journal a un nom propre (codé en hexadécimal) dont l'identifiant est incrémenté de 1 pour chaque journal ;
5. Il cherche à conserver un nombre constant de journaux de transactions. Ce nombre dépend de la configuration.

Contenu d'un journal de transactions

Maintenant que nous savons où ils sont stockés et à quoi ils servent, intéressons-nous un peu à leur contenu. Pour cela, nous allons utiliser un outil appelé xlogviewer. Cet outil fait partie du projet XlogViewer sur pgFoundry (http://pgfoundry.org/projects/xlogviewer/). L'installation n'est pas très simple, notamment parce qu'il faut appliquer un correctif aux sources pour que l'outil soit compatible avec PostgreSQL 8.3, donc l'installation ne sera pas expliquée ici. Cet outil s'utilise par le biais de deux procédures stockées qu'il faut ajouter à la base utilisée :

```
guillaume@laptop$ psql -f /opt/postgresql-8.3/share/contrib/xlogviewer.sql postgres
CREATE TYPE
CREATE FUNCTION
CREATE FUNCTION
```

La procédure stockée que nous allons utiliser demande comme argument le nom du journal de transactions à décoder. La procédure décode à chaque fois tout le journal. Seules certaines colonnes renvoyées par cette procédure sont intéressantes pour nous :

- xid est l'identifiant de la transaction (il y aura donc plusieurs lignes avec le même identifiant pour représenter toutes les modifications effectuées par une transaction) ;
- rmname correspond au type d'enregistrement ;
- total_len est le nombre d'octets enregistrés dans le journal de transactions ;
- infoname tente de décoder l'opération effectuée.

Pour ne pas être gêné par les anciennes transactions, nous allons commencer par changer de journal de transactions :

```
postgres=# SELECT pg_switch_xlog();
```

Maintenant, créons une base de données :

```
postgres=# CREATE DATABASE db1;
CREATE DATABASE
```

Et regardons le contenu du journal de transactions :

```
postgres=# SELECT *
postgres-# FROM xlogviewer('/home/guillaume/glmf/pg_xlog/'||
postgres-# pg_xlogfile_name(pg_current_xlog_location()));
 rmid | xid | rmname | info | len | total_len |  infoname
------+-----+--------+------+-----+-----------+-------------
   10 | 390 | HEAP   |  9   |  18 |       710 | insert
   11 | 390 | BTREE  |  9   |  18 |       410 | insert_leaf
   11 | 390 | BTREE  |  9   |  18 |       170 | insert_leaf
    4 | 390 | DBASE  |  0   |  16 |        44 |
    0 |   0 | XLOG   | 10   |  36 |        64 | checkpoint
    1 | 390 | XACT   |  0   |  16 |        44 | commit
(6 rows)
```

La première ligne nous indique une insertion (« insert ») dans une table (« HEAP »). Nous sommes tout à fait en droit de supposer qu'il s'agit de l'insertion dans le catalogue système pg_database.

Ensuite nous avons deux lignes concernant des index Btree. Elles indiquent une insertion (« insert_leaf ») dans deux index. pg_database dispose de deux index. Il s'agit donc de la modification des index suite à l'insertion dans la table. « DBASE » indique la création de la base.

Une création de base exécute automatiquement un CHECKPOINT, d'où l'enregistrement « checkpoint ». Remarquez que ce dernier n'est pas lié à une transaction (colonne « xid » à 0).

Enfin, nous avons le COMMIT indiquant la validation de la transaction.

Maintenant, créons une table et regardons les nouveaux enregistrements contenus dans le journal :

```
postgres=# CREATE TABLE t1 (id int4, contenu text);
CREATE TABLE
postgres=# SELECT *
postgres-# FROM xlogviewer('/home/guillaume/glmf/pg_xlog/'||
postgres-# pg_xlogfile_name(pg_current_xlog_location()));
 rmid | xid | rmname | info | len | total_len |  infoname
------+-----+--------+------+-----+-----------+-------------
[... suppression des infos sur les anciennes transactions ...]
    2 |   0 | SMGR   | 10   |  12 |        40 | create rel
   10 | 391 | HEAP   |  9   |  18 |      6802 | insert
   11 | 391 | BTREE  |  9   |  18 |      4458 | insert_leaf
   11 | 391 | BTREE  |  9   |  18 |      7386 | insert_leaf
[... coupé ...]
   10 | 391 | HEAP   |  0   |  49 |        77 | insert
   11 | 391 | BTREE  |  0   |  38 |        66 | insert_leaf
   11 | 391 | BTREE  |  0   |  38 |        66 | insert_leaf
   10 | 391 | HEAP   | 70   | 146 |       174 | inplace
   10 | 391 | HEAP   | 70   | 146 |       174 | inplace
   10 | 391 | HEAP   | 40   | 168 |       196 |
   10 | 391 | HEAP   |  0   |  49 |        77 | insert
   11 | 391 | BTREE  |  0   |  38 |        66 | insert_leaf
   11 | 391 | BTREE  |  0   |  38 |        66 | insert_leaf
    1 | 391 | XACT   |  0   |  16 |        44 | commit
(113 rows)
```

Ajouter une table réalise un grand nombre de modifications dans les tables et index systèmes. En fait, il y en a 105 qui sont suivies de la validation de la transaction.

Insérons une ligne dans cette table :

```
postgres=# INSERT INTO t1 VALUES (1, 'un');
INSERT 0 1
postgres=# SELECT *
postgres-# FROM xlogviewer('/home/guillaume/glmf/pg_xlog/'||
postgres-# pg_xlogfile_name(pg_current_xlog_location()));
[... suppression des infos sur les anciennes transactions ...]
   10 | 392 | HEAP   | 80   |  31 |        59 | insert
    1 | 392 | XACT   |  0   |  16 |        44 | commit
(116 rows)
```

Deux enregistrements pour cette insertion : l'insertion elle-même et la validation de la transaction. L'ajout d'une nouvelle ligne contenant un entier et une chaîne de deux caractères a demandé l'ajout de 59 octets dans le journal de transactions.

Ajoutons un index et insérons une donnée :

```
postgres=# CREATE INDEX i1 ON t1 (id);
CREATE INDEX
postgres=# INSERT INTO t1 VALUES (2, 'deux');
INSERT 0 1
postgres=# SELECT *
postgres-# FROM xlogviewer('/home/guillaume/glmf/pg_xlog/'||
postgres-# pg_xlogfile_name(pg_current_xlog_location()));
 rmid | xid | rmname | info | len | total_len |  infoname
------+-----+--------+------+-----+-----------+-------------
[... suppression des infos sur les anciennes transactions ...]
   10 | 395 | HEAP   |  0   |  33 |        61 | insert
   11 | 395 | BTREE  |  9   |  18 |       138 | insert_leaf
    1 | 395 | XACT   |  0   |  16 |        44 | commit
(138 rows)
```

Cette fois, il y a trois enregistrements : l'insertion de la donnée dans la table (« insert »), l'insertion dans l'index (« insert_leaf ») et la validation de la transaction.

Exécutons maintenant une opération CHECKPOINT :

```
postgres=# CHECKPOINT;
CHECKPOINT
postgres=# SELECT *
postgres-# FROM xlogviewer('/home/guillaume/glmf/pg_xlog/'||
postgres-# pg_xlogfile_name(pg_current_xlog_location()));
 rmid | xid | rmname | info | len | total_len |  infoname
------+-----+--------+------+-----+-----------+-------------
[... suppression des infos sur les anciennes transactions ...]
    0 |   0 | XLOG   | 10   |  36 |        64 | checkpoint
(139 rows)
```

Un seul enregistrement est ajouté dans le journal des transactions pour indiquer qu'un CHECKPOINT a été réalisé.

Tous ces exemples ont pour but de montrer que chaque modification de schéma et que chaque modification de données est tracée dans les journaux de transactions.

Le journal de transactions ne contient pas la requête en clair, mais seulement les octets modifiés de chaque bloc de chaque fichier. Le fait d'avoir les octets modifiés permet justement de restaurer une sauvegarde. Pour plus de sécurité, PostgreSQL sauvegarde chaque bloc complet dans le journal de transactions avant d'indiquer les octets à modifier de ce bloc. Cette sauvegarde n'a lieu qu'à la première modification d'un bloc après un checkpoint. Du coup, moins vous avez de checkpoint, moins les journaux grossiront et plus vous gagnerez en rapidité. Il est même possible de désactiver l'enregistrement des pages complètes en positionnant à off le paramètre full_page_writes.

L'utilisation de XlogViewer n'est pas simple, notamment dû au fait qu'il n'est plus maintenu. Dans le cas où la compilation de PostgreSQL est possible, il est intéressant pour mieux comprendre le système des journaux de transactions de demander au serveur de tracer son utilisation des journaux. C'est même beaucoup plus parlant que XlogViewer.

Pour cela, après avoir lancé l'étape configure mais avant la compilation de PostgreSQL, vous devez activer la fonctionnalité WAL_DEBUG. Voici le moyen le plus simple (à exécuter dans le répertoire des sources de PostgreSQL) :

```
guillaume@laptop$ echo "#define WAL_DEBUG" >> src/include/pg_config.h
```

Ceci fait, vous pouvez lancer la compilation avec make. Une fois le serveur compilé et installé, la configuration doit disposer de la ligne suivante :

```
wal_debug = true
```

Recommençons une partie de l'exemple ci-dessus. Créons une base de données :

```
guillaume@laptop:~$ psql -q postgres
postgres=# SET client_min_messages TO log;
postgres=# CREATE DATABASE db5;
LOG:  INSERT @ 0/104EA44: prev 0/104EA04; xid 417; bkpb1: Heap - insert: rel 1664/0/1262; tid 0/8
LOG:  INSERT @ 0/104EF00: prev 0/104EA44; xid 417; bkpb1: Btree - insert: rel 1664/0/2671; tid 1/5
LOG:  INSERT @ 0/104F1CC: prev 0/104EF00; xid 417; bkpb1: Btree - insert: rel 1664/0/2672; tid 1/8
LOG:  INSERT @ 0/104F2B8: prev 0/104F1CC; xid 417: Database - create db: copy dir 1/1663 to 16414/1663
LOG:  INSERT @ 0/104F324: prev 0/104F2E4; xid 417: Transaction - commit: 2008-07-22 23:15:12.806781+02
LOG:  xlog flush request 0/104F350; write 0/104EA04; flush 0/104EA04
```

Les informations sont assez simples à décoder. Contrairement à XlogViewer, les traces nous indiquent l'objet concerné grâce aux nombres situés après rel. Le première nombre correspond à l'OID du tablespace, le deuxième à l'OID de la base (0 dans le cas d'un catalogue partagé) et le dernier au relfilenode de l'objet.

Nous avions dit tout à l'heure que l'insertion se faisait dans la table pg_database et dans ses index. Vérifions-ça :

```
postgres=# SELECT reltablespace, relfilenode, relname, relkind
postgres-# FROM pg_class WHERE relname LIKE 'pg_database%';
 reltablespace | relfilenode |          relname          | relkind
---------------+-------------+---------------------------+---------
          1664 |        2671 | pg_database_datname_index | i
          1664 |        2672 | pg_database_oid_index     | i
          1664 |        1262 | pg_database               | r
(3 rows)
```

Le premier enregistrement dans le journal de transactions indique « rel 1664/0/1262 », ce qui signifie tablespace 1664, base 0 (donc objet partagé entre toutes les bases) et table 1262. D'après le résultat de la requête ci-dessus, cela concerne bien la table pg_database. De la même façon, nous voyons que le second enregistrement concerne l'index pg_stat_database_datname_index et le troisième l'index pg_database_oid_index. Le quatrième enregistrement indique l'ordre de création d'une base et le cinquième précise l'heure de validation.

Maintenant, insérons une donnée dans notre table t1 :

```
postgres=# INSERT INTO t1 VALUES (10, 'dix');
LOG:  INSERT @ 0/104F390: prev 0/104F350; xid 418; bkpb1: Heap - insert: rel 1663/11511/16390; tid 0/6
LOG:  INSERT @ 0/104F4D0: prev 0/104F390; xid 418; bkpb1: Btree - insert: rel 1663/11511/16396; tid 1/5
LOG:  INSERT @ 0/104F58C: prev 0/104F4D0; xid 418: Transaction - commit: 2008-07-22 23:21:09.92561+02
LOG:  xlog flush request 0/104F5B8; write 0/104F350; flush 0/104F350
```

Nous vérifions ainsi que le premier enregistrement concerne l'insertion dans la table (le relfilenode 16390 pointe bien vers la table t1), le second s'occupe des modifications de l'index et le dernier concerne la validation.

Ce qu'il faut tirer de tout ceci se résume en quelques points :

- les journaux de transactions enregistrent toutes les modifications des fichiers de données avant que ceux-ci ne soient modifiés ;
- d'autres événements, comme les CHECKPOINT, sont aussi enregistrés dans les journaux ;
- les journaux concernent toutes les bases de données.

Fonctionnalités avancées

Gestion des performances

La première façon d'améliorer les performances, c'est de faire en sorte de diminuer les écritures. Le meilleur moyen, c'est de configurer wal_buffers pour que cette partie de mémoire soit capable de conserver une transaction complète par processus exécuté en parallèle. En effet, une transaction va faire un certain nombre de modifications au niveau des fichiers correspondant aux tables et/ou index de la base. Tant que la transaction n'est pas terminée, l'écrire sur disque n'a aucun intérêt. Dans le cas d'un crash du serveur, ce qui est en mémoire sera perdue, mais comme la transaction n'est pas terminée, qu'elle soit ou non sur disque, les modifications ne devront pas être prises en compte. Donc autant commencer par ne pas les écrire sur disque tant qu'un COMMIT n'a pas eu lieu. Le vrai problème n'est pas de comprendre l'intérêt de ce paramètre, mais plutôt de savoir le configurer. Certains pensent qu'un mégaoctet est une bonne valeur quand le serveur comprend plusieurs processeurs (car plusieurs processus peuvent s'exécuter réellement en parallèle, ce qui augmente le nombre de transactions en parallèle, et de ce fait la mémoire utilisée pour conserver en cache leurs modifications). D'autres réfléchissent plutôt en terme de taille d'un journal de transactions. 8 Mo semble une valeur assez populaire car ça permet de faire tenir la moitié d'un journal de transactions en mémoire. Quelque soit la valeur que vous choisirez, souvenez-vous que l'écriture du journal se fait toujours au plus tard au moment du COMMIT, que la mémoire soit pleine ou non. Donc n'hésitez pas à augmenter ce paramètre car 64 Ko est une valeur vraiment basse.

L'autre façon d'améliorer les performances revient à éviter de déplacer la tête de lecture du disque. Pour cela, il est préférable de mettre les journaux de transactions sur leur propre disque et le reste des données sur un autre (voire sur plusieurs autres). La mise en place est assez simple mais demande l'arrêt du serveur le temps du déplacement des journaux :

```
guillaume@laptop$ pg_ctl stop
[... messages coupés ...]
guillaume@laptop$ mv $PGDATA/pg_xlog /opt/nouveau/disque
guillaume@laptop$ ln -s /opt/nouveau/disque/pg_xlog
guillaume@laptop$ pg_ctl start
[... messages coupés ...]
```

L'outil initdb de la version 8.3 dispose d'une nouvelle option, -X, permettant de placer immédiatement les journaux de transactions à un autre emplacement. Elle fait exactement la même chose : création du répertoire à l'emplacement indiqué puis création d'un lien symbolique.

Installer un système RAID1 est généralement une bonne idée pour les journaux de transactions. Par contre, le RAID5 est déconseillé car il est très lent pour les écritures et que les journaux de transactions ne sont, en temps normal, que écrits.

Reprise après arrêt brutal

Le gros intérêt des journaux de transactions tient du fait que PostgreSQL peut les rejouer. Par rejouer, on veut dire que le serveur est capable de les lire et de récupérer les éléments des journaux de transactions, un par un, pour les stocker dans les fichiers de données et ainsi obtenir des fichiers de données cohérents.

Si une transaction s'est bien terminée, que les modifications qu'elle a réalisé ont bien été enregistrées dans les journaux de transactions, mais que les fichiers de données n'ont pas eu le temps d'être mis à jour, la première étape du prochain redémarrage du serveur PostgreSQL consistera à relire les journaux de transactions pour enregistrer les données dans les fichiers de données. Pendant cette période de restauration, les connexions ne seront pas possible. Une fois la restauration terminée, les utilisateurs pourront de nouveau lancer des requêtes.

Voici un exemple des traces associées à la reprise après un crash :

```
LOG:  database system was interrupted at 2008-07-15 15:26:03 CEST
LOG:  checkpoint record is at 9/E43B3B00
LOG:  redo record is at 9/E43B3B00; undo record is at 0/0; shutdown FALSE
LOG:  next transaction ID: 109920594; next OID: 9355796
LOG:  next MultiXactId: 34; next MultiXactOffset: 67
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  record with zero length at 9/E43B3B44
LOG:  redo is not required
LOG:  database system is ready
```

PostgreSQL se rend compte qu'il a été arrêté brutalement (« database system was interrupted »). L'enregistrement à recommencer commence à l'emplacement 9/E43B3B00, et PostgreSQL lance la restauration (« automatic recovery in progress »). Il n'y a qu'un enregistrement de longueur nul après l'emplacement cité. Du coup, la restauration n'est pas nécessaire et le système est prêt.

PITR

Comme il est possible de restaurer les fichiers de données à partir des journaux de transactions, certains développeurs se sont dit qu'ils seraient possibles de restaurer un serveur à partir d'une sauvegarde des fichiers à un instant t et des journaux de transactions créés à partir de cet instant. La difficulté rencontrée à l'époque était de récupérer le journal de transactions une fois que ce dernier est terminé mais avant qu'il ne soit renommé pour être réutilisé. Pour cela, un nouveau paramètre à vu le jour : archive_command.

Ce paramètre doit indiquer la commande à exécuter pour archiver un journal de transactions. Cette commande peut être cp, scp, rsync ou toute autre commande, voire même un script ou un programme développé par l'administrateur. Il y a une seule obligation : la commande doit renvoyer 0 en cas de succès de l'archivage. Tant que le serveur ne récupère pas ce code de statut 0, il conserve le journal de transactions et ré-exécute la commande d'archivage.

Le cycle de vie d'un journal de transactions devient donc :

- tant que le journal n'est pas plein ou que le délai archive_timeout n'est pas dépassé ou que pg_switch_xlog() n'est pas exécuté ;
- utilisation du journal pour y stocker les données et métadonnées ;
- si archive_command est renseigné et tant que le journal n'est pas archivé :
- archivage du journal par la commande archive_command ;
- renommage du journal pour ré-utilisation (ou suppression si le nombre de journaux actuellement présents est trop important).

La sauvegarde

Voici comment mettre en place l'archivage des journaux de transactions. Dans cet exemple, l'archivage se fait sur le même disque mais, dans la vraie vie, il faut qu'il soit au minimum sur un disque séparé, voire même sur un autre serveur.

Il faut commencer par créer le répertoire d'archivage :

```
guillaume@laptop:~$ mkdir /tmp/pg_xlog_archives
```

Ensuite, il faut modifier la configuration de PostgreSQL. Plutôt que de modifier le fichier principal, on va lui ajouter un simple include vers le fichier de configuration local nommé archivage.conf :

```
guillaume@laptop:~$ cat >> $PGDATA/postgresql.conf <<fin_config
> include 'archivage.conf'
> fin_config
guillaume@laptop:~$ cat > $PGDATA/archivage.conf <<fin_config
> archive_mode = on
> archive_command = 'cp %p /tmp/pg_xlog_archives/%f'
> fin_config
```

La modification de archive_command se satisfait très bien d'un simple rechargement de la configuration. Par contre, la modification de archive_mode nécessite le redémarrage. Nous allons donc redémarrer le serveur :

```
guillaume@laptop:~$ pg_ctl restart
LOG:  shutting down
LOG:  database system is shut down
 done
server stopped
server starting
LOG:  database system was shut down at 2008-07-12 17:40:30 CEST
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
```

Connectons-nous au serveur et essayons de générer des journaux de transactions :

```
guillaume@laptop:~$ psql -q postgres
postgres=# \! ls -lhG /tmp/pg_xlog_archives
total 0
```

Le nouveau répertoire ne contient aucun fichier.

```
postgres=# SELECT pg_xlogfile_name(pg_current_xlog_location());
     pg_xlogfile_name
--------------------------
 000000010000000000000007
(1 row)
```

Nous en sommes au journal de transactions 7. Ce fichier sera archivé une fois qu'il sera rempli. Lançons un ensemble d'INSERT :

```
postgres=# INSERT INTO t1 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# SELECT pg_xlogfile_name(pg_current_xlog_location());
     pg_xlogfile_name
--------------------------
 000000010000000000000008
(1 row)
```

Nous sommes bien passés au nouveau journal de transactions…

```
postgres=# \! ls -lhG /tmp/pg_xlog_archives
total 17M
-rw------- 1 guillaume 16M 2008-07-13 16:33 000000010000000000000007
```

… et le journal 7 a bien été archivé.

L'archivage étant configuré, nous pouvons passer à la sauvegarde des fichiers. Cela se passe en trois étapes :

1. prévenir PostgreSQL du lancement de la sauvegarde fichiers ;
2. sauvegarder les fichiers ;
3. et prévenir PostgreSQL de la fin de la sauvegarde.

La fonction pg_start_backup() est utilisée pour la première étape. Cette fonction commence par un CHECKPOINT pour s'assurer que toutes les données modifiées en mémoire sont bien enregistrées sur disque, puis crée un fichier backup_label donnant quelques informations sur le moment de la sauvegarde. Enfin, elle renvoie l'emplacement actuel dans le journal de transactions :

```
postgres=# SELECT pg_start_backup('sauve_20080713_1634');
 pg_start_backup
-----------------
 0/84C5654
(1 row)

postgres=# \q
guillaume@laptop:~$ cat $PGDATA/backup_label
START WAL LOCATION: 0/84C5654 (file 000000010000000000000008)
CHECKPOINT LOCATION: 0/84C65C4
START TIME: 2008-07-13 16:36:55 CEST
LABEL: sauve_20080713_1634
```

« START WAL LOCATION » indique le journal de transactions courant au début de la sauvegarde ainsi que la position au sein de ce fichier. « CHECKPOINT LOCATION » indique la position du dernier CHECKPOINT (les deux positions doivent être identiques vu que le CHECKPOINT est exécuté par la fonction pg_start_backup()). « START_TIME » indique le jour et l'heure de la sauvegarde. Enfin « LABEL » correspond à la valeur de l'argument lors de l'appel de la fonction pg_start_backup().

Il ne nous reste plus qu'à faire la sauvegarde au niveau fichiers. N'importe quel outil habituel ira bien : tar, rsync, cp, etc. Peu importe la durée de la sauvegarde, les utilisateurs peuvent toujours se connecter, faire les modifications qu'ils souhaitent… la base de données peut continuer à vivre normalement, sans aucune restriction, pendant la sauvegarde des fichiers.

```
guillaume@laptop:~$ tar cjf glmf.tar.bz2 glmf
```

Pendant qu'il fait la sauvegarde, créons une base de données et peuplons-la :

```
guillaume@laptop:~$ createdb db1
guillaume@laptop:~$ psql -q db1
db1=# CREATE TABLE t2 (id2 int4);
db1=# INSERT INTO t2 SELECT x FROM generate_series(1,1000) AS x;
db1=# \q
```

Aucun message d'erreur de la base de données. Les utilisateurs peuvent donc vraiment modifier le schéma comme les données de l'instance. Par contre, tar pourrait se plaindre que des fichiers sont modifiés pendant leur sauvegarde.

Peu importe pour nous car toutes les modifications réalisées en ce moment sont enregistrées dans les journaux de transactions qui sont archivés. C'est

d'ailleurs pour cela qu'il faut commencer par mettre en place l'archivage des journaux avant de réaliser la sauvegarde des fichiers.

Une fois la sauvegarde terminée, nous prévenons PostgreSQL de la fin de la sauvegarde en exécutant la fonction pg_stop_backup().

```
guillaume@laptop:~$ psql -q postgres
postgres=# SELECT pg_stop_backup();
 pg_stop_backup
----------------
 0/84E7980
(1 row)
```

Cette dernière renvoie l'emplacement actuel dans le journal de transactions. Créons une nouvelle table et insérons-y des données :

```
postgres=# CREATE TABLE t3 (id int4);
postgres=# INSERT INTO t3 SELECT x FROM generate_series(1, 1000000) AS x;
postgres=# SELECT now();
              now
-------------------------------
 2008-07-13 16:39:34.954131+02
(1 row)
[... un peu plus tard ...]
postgres=# SELECT now();
              now
-------------------------------
 2008-07-13 16:40:09.445774+02
(1 row)

postgres=# INSERT INTO t3 SELECT x FROM generate_series(4000000,5000000) AS x;
postgres=# SELECT now();
              now
-------------------------------
 2008-07-13 16:40:33.375076+02
(1 row)
```

Pour nous assurer que les dernières modifications sont bien archivées, nous allons nous-même demander un changement de journal de transactions :

```
postgres=# SELECT pg_switch_xlog();
 pg_switch_xlog
----------------
 0/FB20BF0
(1 row)
postgres=# \q
```

Une restauration complète

Elle se passe en trois grosses étapes :

- restauration de la sauvegarde des fichiers ;
- configuration de la restauration des journaux de transactions ;
- lancement du serveur.

Nous allons stopper le serveur, déplacer l'ancien répertoire de l'instance pour pouvoir déballer le contenu de l'archive tar :

```
guillaume@laptop:~$ pg_ctl stop
waiting for server to shut down....LOG:  received smart shutdown request
LOG:  autovacuum launcher shutting down
LOG:  shutting down
LOG:  database system is shut down
 done
server stopped
guillaume@laptop:~$ mv glmf glmf.old
guillaume@laptop:~$ tar xjf glmf.tar.bz2
```

La restauration se configure dans le fichier recovery.conf. Un seul paramètre nous intéresse actuellement, celui qui va indiquer la commande permettant de récupérer le journal de transactions :

```
guillaume@laptop:~$ cat >> $PGDATA/recovery.conf <<fin_config
> restore_command = 'cp /tmp/pg_xlog_archives/%f %p'
> fin_config
```

Il nous faut aussi supprimer l'archivage (ou archiver ailleurs que dans l'ancien répertoire) pour éviter que PostgreSQL n'archive de nouveau les journaux qu'il restaure.

```
guillaume@laptop:~$ > $PGDATA/archivage.conf
```

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

```
guillaume@laptop:~$ pg_ctl start
pg_ctl: another server might be running; trying to start server anyway
LOG:  database system was interrupted; last known up at 2008-07-13 16:38:17 CEST
LOG:  starting archive recovery
LOG:  restore_command = 'cp /tmp/pg_xlog_archives/%f %p'
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
LOG:  restored log file "000000010000000000000008.004C5654.backup" from archive
LOG:  restored log file "000000010000000000000008" from archive
LOG:  automatic recovery in progress
server starting
LOG:  redo starts at 0/84C5654
LOG:  restored log file "000000010000000000000009" from archive
LOG:  restored log file "00000001000000000000000A" from archive
LOG:  restored log file "00000001000000000000000B" from archive
LOG:  restored log file "00000001000000000000000C" from archive
LOG:  restored log file "00000001000000000000000D" from archive
LOG:  restored log file "00000001000000000000000E" from archive
LOG:  restored log file "00000001000000000000000F" from archive
cp: cannot stat `/tmp/pg_xlog_archives/000000010000000000000010': No such file or directory
LOG:  could not open file "pg_xlog/000000010000000000000010" (log file 0, segment 16): No such file or directory
LOG:  redo done at 0/FB20BD4
LOG:  last completed transaction was at log time 2008-07-13 16:40:30.632653+02
LOG:  restored log file "00000001000000000000000F" from archive
cp: cannot stat `/tmp/pg_xlog_archives/00000002.history': No such file or directory
LOG:  selected new timeline ID: 2
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
LOG:  archive recovery complete
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
guillaume@laptop:~$
```

Quelques explications sur les traces… « database system was interrupted » indique que la base a été arrêté de façon anormale. C'est tout à fait normale, la sauvegarde s'est faite alors que le serveur était toujours en cours d'exécution. Ce message n'est donc pas du tout alarmant dans ce cadre.

« starting archive recovery » indique que le serveur entre dans son mode de restauration. Il indique d'ailleurs la commande de copie des journaux de transactions sur la ligne suivante. Il tente de récupérer un fichier historique sans y parvenir, ce qui n'est pas alarmant en soi. Enfin, il se met à restaurer les journaux de transactions (message du type « restored log file “000000010000000000000009” »).

Une fois la restauration terminée, PostgreSQL va basculer sur une nouvelle TimeLineID (« selected new timeline ID: 2 »). Il indique la fin de la restauration avec le message « archive recovery complete » et le système est prêt à accepter des connexions des utilisateurs (« database system is ready to accept connections »).

Dans ce cas, connectons-nous et regardons l'état de la base de données :

```
guillaume@laptop:~$ psql -q postgres
postgres=# \d
        List of relations
 Schema | Name | Type  |   Owner
--------+------+-------+-----------
 public | t1   | table | guillaume
 public | t3   | table | guillaume
(2 rows)
```

Nous avons nos deux tables… donc nous avons bien la table créée après la sauvegarde de base.

```
postgres=# SELECT count(*) FROM t1;
  count
---------
 2400003
(1 row)
```

```
postgres=# SELECT count(*) FROM t3;
  count
---------
 2000000
(1 row)
```

Et nous avons aussi les données. Concernant la base de données créées pendant la sauvegarde des fichiers :

```
postgres=# \l
        List of databases
   Name    |   Owner   | Encoding
-----------+-----------+----------
 db1       | guillaume | UTF8
 postgres  | guillaume | UTF8
 template0 | guillaume | UTF8
 template1 | guillaume | UTF8
(4 rows)
```

Elle est bien présente.

```
postgres=# \c db1
db1=# \d
         List of relations
 Schema | Name | Type |   Owner
--------+------+-------+-----------
 public | t2   | table | guillaume
(1 row)
```

La table t2 est aussi là.

```
db1=# SELECT count(*) FROM t2;
 count
-------
  1000
(1 row)
```

Les 1000 lignes insérées sont aussi disponibles.

Nous sommes passés un peu rapidement sur le message concernant la TimeLineID. Nous sommes passés en TimeLineID 2 et cela se réflète sur les noms des nouveaux journaux de transactions :

```
guillaume@laptop:~$ ls -lhG $PGDATA/pg_xlog
total 145M
-rw------- 1 guillaume  16M 2008-07-13 16:36 000000010000000000000008
-rw------- 1 guillaume  16M 2008-07-13 16:22 000000010000000000000009
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000A
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000B
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000C
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000D
-rw------- 1 guillaume  16M 2008-07-13 16:23 00000001000000000000000E
-rw------- 1 guillaume  16M 2008-07-13 16:43 00000002000000000000000F
-rw------- 1 guillaume  16M 2008-07-13 16:43 000000020000000000000010
-rw------- 1 guillaume   74 2008-07-13 16:43 00000002.history
drwx------ 2 guillaume 4,0K 2008-07-13 16:43 archive_status
```

De même, nous avons maintenant un fichier historique pour cette TimeLineID.

Restauration partielle

Et si nous avions voulu restaurer jusqu'à un certain moment ? Il est tout à fait possible d'indiquer jusqu'à quand vous voulez restaurer grâce à deux paramètres :

- recovery_target_time permet de préciser une heure d'arrêt de la restauration.
- recovery_target_xid permet d'indiquer la dernière transaction à restaurer en la désignant par son identifiant. Cette méthode est pratiquement inutilisable actuellement car il n'existe pas de moyens pour lier une transaction à son identifiant en dehors de tracer toutes les requêtes avec l'identifiant de transaction associé (caractère joker %x pour le paramètre log_line_prefix).

Si nous reprenons notre sauvegarde de base et que nous cherchions à la restaurer juste entre les deux insertions dans t3. La deuxième insertion s'étant faite après 16h40, nous devrions pouvoir restaurer juste avant.

Le début est identique à la restauration précédente :

```
guillaume@laptop:~$ pg_ctl stop
waiting for server to shut down...LOG:  received smart shutdown request
.LOG:  autovacuum launcher shutting down
LOG:  shutting down
LOG:  database system is shut down
... done
server stopped
guillaume@laptop:~$ rm -r glmf
guillaume@laptop:~$ tar xjf glmf.tar.bz2
guillaume@laptop:~$ cat >> $PGDATA/recovery.conf <<fin_config
> restore_command = 'cp /tmp/pg_xlog_archives/%f %p'
> recovery_target_time = '2008-07-13 16:40:00'
> fin_config
guillaume@laptop:~$ > /home/guillaume/glmf/archivage.conf
```

Remarquez l'ajout du paramètre recovery_target_time précisant la date et l'heure de fin de la restauration.

Enfin, nous lançons PostgreSQL. Les messages du serveur vont légèrement changer.

```
guillaume@laptop:~$ pg_ctl start
pg_ctl: another server might be running; trying to start server anyway
LOG:  database system was interrupted; last known up at 2008-07-13 16:38:17 CEST
LOG:  starting archive recovery
LOG:  restore_command = 'cp /tmp/pg_xlog_archives/%f %p'
LOG:  recovery_target_time = '2008-07-13 16:40:00+02'
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
LOG:  restored log file "000000010000000000000008.004C5654.backup" from archive
LOG:  restored log file "000000010000000000000008" from archive
LOG:  automatic recovery in progress
server starting
guillaume@laptop:~$ LOG:  redo starts at 0/84C5654
LOG:  restored log file "000000010000000000000009" from archive
LOG:  restored log file "00000001000000000000000A" from archive
LOG:  restored log file "00000001000000000000000B" from archive
LOG:  restored log file "00000001000000000000000C" from archive
LOG:  recovery stopping before commit of transaction 408, time 2008-07-13 16:40:01.887323+02
LOG:  redo done at 0/C59D288
LOG:  last completed transaction was at log time 2008-07-13 16:40:01.887323+02
cp: cannot stat `/tmp/pg_xlog_archives/00000002.history': No such file or directory
LOG:  selected new timeline ID: 2
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
LOG:  archive recovery complete
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections

guillaume@laptop:~$ psql -q postgres
postgres=# SELECT count(*) FROM t3;
  count
---------
 1000000
(1 row)

postgres=#
```

La ligne « recovery_target_time = '2008-07-13 16:40:00+02' » indique qu'il a bien compris que la restauration ne sera pas complète car elle devra se terminer dans l'état cohérent à 16h40.

D'ailleurs, nous voyons que la restauration s'achève au journal de transactions C (contrairement à auparavant où il avait été jusqu'au F). « recovery stopping before commit of transaction 408, time 2008-07-13 16:40:01.887323+02 » indique la fin de la restauration, juste avant la validation de la transaction 408 qui a eu lieu à 16h40 et une seconde. Trop tard pour celle-ci.

LogShipping

Restaurer une sauvegarde des fichiers avec les journaux de transactions associés peut prendre beaucoup de temps. C'est pour cela que la version 8.2 a ajouté une fonctionnalité supplémentaire appelée « Log Shipping ». L'idée est d'avoir un serveur en attente des journaux de transactions archivés pour

pouvoir les restaurer dès réception.

L'installation du serveur principal se fait comme au chapitre précédent. Sur le serveur secondaire, nous devons restaurer la sauvegarde de base, ajouter le fichier recovery.conf et lancer PostgreSQL. Cependant, la commande de restauration, celle pointée par le paramètre restore_command, doit tout d'abord attendre l'arrivée du journal de transactions avant de pouvoir le fournir à PostgreSQL.

Nous ne pouvons donc plus utiliser des commandes simples comme cp ou scp. Il est nécessaire d'écrire un script… ou de récupérer un outil tout fait, comme pg_standby disponible en tant que module contrib sur la version 8.3.

pg_standby est un outil développé par Simon Riggs. Son utilisation est très simple. Il demande en argument l'emplacement des journaux archivés, le nom du prochain journal à restaurer et l'endroit où le copier. Certaines options permettent d'aller plus loin :

- l'option -d permet d'avoir des traces supplémentaires ;
- l'option -l permet de créer un lien plutôt que de copier le fichier ;
- l'option -s permet de modifier l'attente entre deux vérifications (par défaut, cinq secondes) ;
- l'option -t permet de spécifier un fichier qui annulera le mode restauration ;
- etc.

Mettons donc en place le « Log Shipping ». Nous supposons que le serveur maître est déjà configuré pour archiver les journaux de transactions. Nous allons simplement modifier la configuration pour que les journaux soient copiés sur un serveur nommé esclave.

```
guillaume@laptop:~$ cat > $PGDATA/archivage.conf <<fin_config
> archive_mode = on
> archive_command = 'scp %p esclave:/tmp/pg_xlog_archives/%f'
> fin_config
```

Note : pour permettre la connexion à l'esclave avec SSH sans indiquer son mot de passe, il est nécessaire de paramétrer l'utilisation de clés et de générer ces clés, mais ce travail est en dehors du périmètre de cet article.

Comme nous n'avons modifié que le paramètre archive_command, nous pouvons simplement recharger la configuration sans redémarrer le serveur :

```
guillaume@laptop:~$ pg_ctl reload
```

Ensuite, nous devons mettre en place l'esclave. Commençons par restaurer la sauvegarde des fichiers :

```
guillaume@esclave:~$ tar xjf glmf.tar.bz2
guillaume@esclave:~$ mv glmf glmf.esclave
guillaume@esclave:~$ export PGDATA=/home/guillaume/glmf.esclave
```

La restauration se configure dans le fichier recovery.conf. Un seul paramètre nous intéresse actuellement, celui qui va indiquer la commande permettant de récupérer le journal de transactions :

```
guillaume@esclave:~$ cat >> $PGDATA/recovery.conf <<fin_config
> restore_command = 'pg_standby -d /tmp/pg_xlog_archives %f %p >> /home/guillaume/glmf.esclave/pg_standby.log 2>&1'
> fin_config
```

Il nous faut aussi supprimer l'archivage (ou archiver ailleurs que dans l'ancien répertoire) pour éviter que PostgreSQL n'archive de nouveau les journaux qu'il restaure.

```
guillaume@esclave:~$ > $PGDATA/archivage.conf
```

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

```
guillaume@esclave:~$ pg_ctl start
server starting
LOG:  database system was interrupted; last known up at 2008-07-14 10:53:26 CEST
LOG:  starting archive recovery
LOG:  restore_command = 'pg_standby -d /tmp/pg_xlog_archives %f %p >>/home/guillaume/glmf.esclave/recovery.log 2>&1'
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
cp: cannot stat `/tmp/pg_xlog_archives/00000001.history': No such file or directory
LOG:  restored log file "000000010000000000000002.00470154.backup" from archive
LOG:  restored log file "000000010000000000000002" from archive
LOG:  automatic recovery in progress
LOG:  restored log file "000000010000000000000003" from archive
LOG:  restored log file "000000010000000000000004" from archive
LOG:  restored log file "000000010000000000000005" from archive
```

Comme la fois précédente, le serveur comprend qu'il est en mode de restauration. Il récupère les journaux 2, 3, 4 et 5.

Maintenant, créons une nouvelle table et générons des données (sur le serveur principal) :

```
postgres=# CREATE TABLE t2 (id int4);
postgres=# INSERT INTO t2 SELECT x FROM generate_series(1, 300000) AS x;
postgres=# SELECT pg_switch_xlog();
 pg_switch_xlog
----------------
 0/704EDB0
(1 row)
```

La fonction pg_switch_xlog() nous assure que le serveur principal change de journal de transactions courant et, du coup, que les données vont être envoyées maintenant au serveur secondaire.

Que constatons-nous sur l'esclave ?

```
guillaume@esclave:~$ tail -f glmf.esclave/pg_standby.log
WAL file not present yet.
WAL file not present yet.
WAL file not present yet.
WAL file not present yet.
running restore         : OK
Trigger file            : <not set>
Waiting for WAL file    : 000000010000000000000006
WAL file path           : /tmp/pg_xlog_archives/000000010000000000000006
Restoring to...         : pg_xlog/RECOVERYXLOG
Sleep interval          : 5 seconds
Max wait interval       : 0 forever
Command for restore     : cp "/tmp/pg_xlog_archives/000000010000000000000006" "pg_xlog/RECOVERYXLOG"
Keep archive history    : No cleanup required
WAL file not present yet.
WAL file not present yet.
```

Quelques explications sur les traces de pg_standby. « WAL file not present » est le message habituel indiquant qu'il cherche le prochain journal de transactions mais qu'il ne le trouve pas. Une fois le journal découvert, il lance la restauration (message « running restore : OK »).

Il affiche quelques autres informations, comme le nom du journal attendu, le chemin complet vers ce fichier, l'endroit où il va le restaurer, la commande complète qu'il va utiliser. Puis il se retrouve de nouveau en attente.

Arrêtons l'esclave pour voir son état…

```
guillaume@esclave:~$ ps -ef | grep [p]g_standby
1000      8385  8272  0 11:00 ?        00:00:00 sh -c pg_standby -d /tmp/pg_xlog_archives 000000010000000000000008 pg_xlog/RECOVERYXLOG >>/home/guillaume/glmf.esclave/recovery.log 2>&1
1000      8386  8385  0 11:00 ?        00:00:00 pg_standby -d /tmp/pg_xlog_archives 000000010000000000000008 pg_xlog/RECOVERYXLOG
guillaume@esclave:~$ kill -INT 8386
LOG:  could not open file "pg_xlog/000000010000000000000008" (log file 0, segment 8): No such file or directory
LOG:  redo done at 0/704ED94
LOG:  last completed transaction was at log time 2008-07-14 11:00:04.022844+02
LOG:  restored log file "000000010000000000000007" from archive

LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
guillaume@esclave:~$ psql -q postgres
postgres=# \d
        List of relations
 Schema | Name | Type  |   Owner
--------+------+-------+-----------
 public | t1   | table | guillaume
 public | t2   | table | guillaume
(2 rows)
postgres=# \q
```

La nouvelle table est présente sur l'esclave. De même pour les données.

Note : il existe d'autres outils pour gérer le « Log Shipping », walmgr étant l'un des plus connus. Il fait partie de l'ensemble d'outils libres développé par Skype. Pour plus d'informations, consultez

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

Nouveautés en 8.3

En dehors du module contrib pg_standby, la version 8.3 propose d'autres améliorations concernant les journaux de transactions.

La première est l'ajout d'un processus dédié à l'écriture des informations dans le journal des transactions courant. À l'instar du « background writer », le « wal writer process » décharge les processus postgres de l'écriture sur les journaux de transactions. Ce travail réalisé, le processus s'endort pendant wal_writer_delay milli-secondes (200 par défaut).

L'autre très importante nouveauté de la 8.3 concerne la performance des journaux de transactions. Nous avons vu précédemment que désactiver fsync avait un gros intérêt pour gagner en performances si on négligeait la sécurité des données. La 8.3 dispose d'un mode permettant de gagner en rapidité tout en conservant un bon niveau de sécurité. Il s'agit du « asynchronous commit », ou autrement dit l'enregistrement asynchrone. En activant ce paramètre, PostgreSQL repousse (très légèrement, d'au plus 100 ms) le moment de l'enregistrement pour pouvoir enregistrer plusieurs transactions à la fois, ce qui fait gagner nettement en performances, principalement lors d'insertions ou de modifications massives (par exemple lors du chargement d'une sauvegarde). L'autre gros intérêt de ce mode est qu'il peut s'activer par session.

De plus, des efforts ont été réalisés pour limiter les écritures dans les journaux de transactions. Par exemple, les modifications effectuées par l'opération CLUSTER ne passent plus par les journaux de transactions. De même, un COPY suivant dans la même transaction la création de la table n'est pas tracé dans les journaux de transactions. Pour ces deux cas, les écritures des journaux de transactions sont évitées si et seulement si l'archivage est désactivé.

restore_command permet l'utilisation d'un nouveau caractère joker, %r. Ce dernier indique le premier journal de transactions réellement utile, ce qui permet aux outils comme pg_standby de supprimer les journaux de transactions antérieurs.

Le futur... vers la 8.4 et au-delà

PGCon 2008 a permis aux développeurs importants du projet de se rencontrer et de se mettre d'accord sur la mise en place d'une réplication encore plus fine. Cette fois, ce ne sont plus les journaux de transactions qui sont envoyés mais l'ensemble des modifications d'une transaction. Donc la réplication proposée sera réalisée transaction par transaction, ce qui permettra d'avoir une vrai réplication synchrone. Si les développements avancent bien, cette fonctionnalité fera partie de la version 8.4. Il s'agit tout simplement d'un LogShipping au niveau des transactions et non plus des groupes de transactions que sont les journaux.

Une autre fonctionnalité très intéressante serait d'avoir des esclaves disponibles en lecture seule. Malheureusement, les développeurs tablent plutôt sur la version 8.5 pour cette fonctionnalité.

Rappel sur la configuration

Il y a quelques paramètres de configuration à connaître. Nous en avons déjà fait le tour, ce n'est qu'un résumé avec la valeur par défaut pour chacun.

Nom du paramètre	Par défaut	Commentaires
		Configuration globale
fsync	on	active la synchronisation sur disque
synchronous_commit	on	active la synchronisation sur disque au moment du COMMIT
wal_sync_method	fsync	méthode de synchronisation
full_page_writes	on	écriture préalable de la page complète
wal_buffers	64 Ko	tampon disque des journaux de transactions
wal_writer_delay	200 ms	délai entre deux réveils du processus « wal writer »
commit_delay	0	délai en microsecondes observé après le COMMIT d'une transaction et son écriture sur disque
commit_siblings	5	nombre minimum de transactions en parallèle pour activer le délai commit_delay
		Checkpoints
checkpoint_segments	3	nombre de segments de 16 Mo
checkpoint_timeout	5 min	délai entre deux CHECKPOINT automatiques
checkpoint_completion_target	0.5	
checkpoint_warning	30 s	Enregistrement d'un message dans les traces si deux CHECKPOINT sont générés l'un à la suite de l'autre dans un délai inférieur à ce paramètre
		Archivage
archive_mode	off	active l'archivage
archive_command	''	commande à utiliser pour l'archivage
archive_timeout	0	Délai avant passage automatique à un nouveau journal des transactions

Conclusion

Les journaux de transactions sont un moyen d'assurer la sécurité de vos données. Cela a un léger coût en terme de performances sauf si vous avez la prudence de les placer sur un disque séparé. Ils vous permettent de plus une sauvegarde incrémentale d'une mise en place simplissime.