−Table des matières

pgPool-II, la réplication par duplication des requêtes

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.

Un moyen extrêmement simple, voire même simpliste diront certains, de faire de la réplication est d'envoyer toutes les requêtes à tous les serveurs en réplication. Évidemment, cela donnera lieu à des limitations assez fortes, mais le résultat peut être intéressant.

Installation de pgPool-II

Comme vu dans l'article sur le mode de pooling de connexions de pgPool-II, l'installation de pgPool-II est très simple. Nous n'allons reprendre ici que les commandes de base:

```
debian1:~# aptitude install postgresql-server-dev-8.4
debian1:~# wget -q http://pgfoundry.org/frs/download.php/2362/pgpool-II-2.2.4.tar.gz
debian1:~# tar xfz pgpool-II-2.2.4.tar.gz
debian1:~# cd pgpool-II-2.2.4
debian1:~/pgpool-II-2.2.4# ./configure –prefix=/opt/pgpool-II-2.2
[... messages de progression …]
debian1:~/pgpool-II-2.2.4# make
[... messages de progression ...]
debian1:~/pgpool-II-2.2.4# make install
[... messages de progression …]
```

Configuration

La configuration a toujours lieu dans le fichier pgpool.conf.

La première chose à faire est d'activer la réplication. Pour cela, il faut mettre à 'on' le paramètre replication_mode.

```
replication_mode = true
```

Ceci fait, pgPool-II enverra une copie de chaque requête qu'il reçoit à tous les nœuds de base de données. Néanmoins, par défaut, les SELECT ne sont pas répliqués. Si vous souhaitez faire exécuter les SELECT sur les différents nœuds, c'est possible. Il faut pour cela activer le paramètre replicate_select. Nous n'allons pas le faire dans notre cas. Ensuite, il faut configurer les différents nœuds. Nous avions déjà configuré le nœud 0 lors du précédent article sur pgPool-II, nous allons maintenant faire le nœud 1:

```
backend_hostname1 = 'debian2'
backend_port1 = 5432
```

pgPool-II va donc se connecter aux deux nœuds. Pour cela, il faut évidemment corriger la configuration des accès sur debian1 et debian2. La première chose à faire est d'autoriser les accès distants. Nous passons donc le paramètre listen_addresses à '*' dans le fichier /etc/postgresql/8.4/main/postgresql.conf. Ensuite, nous devons autoriser les adresses IP des deux serveurs. Cela se fait dans le fichier /etc/postgresql/8.4/main/pg_hba.conf. La configuration retenue est la suivante:

```
# Database administrative login by UNIX sockets
local   all         postgres                          ident

# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD

# "local" is for Unix domain socket connections only
local   all         all                               trust
# IPv4 local connections:
host    all         all         127.0.0.1/32          trust
host    all         all         192.168.10.66/32      trust
host    all         all         192.168.10.67/32      trust
# IPv6 local connections:
host    all         all         ::1/128               trust
```

De cette façon, nous autorisons les connexions de n'importe quel utilisateur PostgreSQL sur n'importe quelle base de données, à condition qu'elle provienne des adresses IP 192.168.10.66 et 192.168.10.67 (adresses IP des serveurs debian1 et debian2). Remarquez la méthode trust. Seules les méthodes trust, password et pam sont acceptées. Password est certainement un peu meilleur que trust car il est quand même nécessaire d'avoir un mot de passe. Cependant, ce dernier circule en clair sur le réseau. pam est probablement le meilleur choix, à défaut de mieux. Néanmoins, cela vous contraint à créer autant d'utilisateurs Unix que de personnes capables de se connecter à l'instance PostgreSQL.

Maintenant que la configuration est faite, nous devons faire en sorte que PostgreSQL la prenne en compte. Si vous avez dû modifier le paramètre listen_addresses, vous devez redémarrer PostgreSQL. Dans le cas contraire, un simple rechargement de la configuration suffit.

```
debian1:/opt/pgpool-II-2.2.4/etc# /etc/init.d/postgresql-8.4 restart
Reloading PostgreSQL 8.4 database server: main.
```

Tout ce travail fait sur debian1 doit aussi être fait sur debian2.

Avant de lancer pgPool-II, il est de bon ton de s'assurer que les connexions se font bien. Essayez de vous connecter sur les bases de debian1 à partir de debian1 et de debian2. Puis faites de même sur les bases de debian2.

Utilisation de la réplication

Commençons par lancer le démon pgpool:

```
debian1:~# cd /opt/pgpool-II-2.2.4/bin
debian1:/opt/pgpool-II-2.2.4/bin# ./pgpool -n
2009-08-27 02:43:38 LOG:   pid 1469: pgpool successfully started
```

Maintenant, nous allons créer une base sur debian1.

```
postgres@debian1:~$ createdb -p 9999 b1
```

Remarquez que nous utilisons toujours l'option -p pour indiquer le numéro de port de pgpool et donc bien nous assurer de passer par pgPool-II. Pour éviter d'avoir à saisir cette option, vous pouvez initialiser la variable d'environnement PGPORT de cette façon:

```
postgres@debian1:~$ export PGPORT=9999
```

Évidemment, un moyen encore plus simple est de faire en sorte que pgPool-II écoute sur le port 5432.

Revenons-en à notre base de données. Nous l'avons créé sur debian1. Voyons si elle existe bien:

```
postgres@debian1:~$ psql -l
                        Liste des bases de données
   Nom    | Propriétaire | Encodage |     Tri     | Type caract. |    Droits d'accès
-----------+--------------+----------+-------------+--------------+-----------------------
 b1        | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 postgres  | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 template0 | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/postgres
                                                                 : postgres=CTc/postgres
 template1 | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/postgres
                                                                 : postgres=CTc/postgres
(12 lignes)
```

Oui, c'est bien le cas. Maintenant, voyons si elle existe sur debian2:

```
postgres@debian1:~$ psql -h debian2 -l
                        Liste des bases de données
   Nom    | Propriétaire | Encodage |     Tri     | Type caract. |    Droits d'accès
-----------+--------------+----------+-------------+--------------+-----------------------
 b1        | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 postgres  | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 template0 | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/postgres
                                                                 : postgres=CTc/postgres
 template1 | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/postgres
                                                                 : postgres=CTc/postgres
(4 lignes)
```

C'est bien le cas aussi. La requête de création de base générée par l'outil createdb a bien été envoyée aux deux serveurs par l'outil pgPool-II. Maintenant, connectons-nous à la base b1 de debian1 et ajoutons une table:

```
postgres@debian1:~$ psql -p 9999 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# CREATE TABLE t1 (id integer);
CREATE TABLE
```

Ajoutons quelques données:

```
b1=# INSERT INTO t1 SELECT i FROM generate_series(1, 1000) AS i;
INSERT 0 1000
```

Parfait. Voyons ce qui s'est passé sur debian2:

```
postgres@debian2:~$ psql b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# \d
        Liste des relations
 Schéma | Nom | Type  | Propriétaire
--------+-----+-------+--------------
 public | t1  | table | postgres
(1 ligne)

b1=# SELECT count(*) FROM t1;
 count
-------
  1000
(1 ligne)
```

La table est bien là. Elle contient bien les nouvelles lignes. Avouez que c'est difficile de faire plus simple.

Voyons maintenant une autre limite de la réplication par les instructions SQL. Créons une table avec un champ de type timestamp (horodatage, avec date, heure et fuseau horaire) et un champ de type float4:

```
postgres@debian1:~$ psql -p 9999 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# CREATE TABLE t2 (id serial, dateheure timestamp, nombre float4);
NOTICE:  CREATE TABLE créera des séquences implicites « t2_id_seq » pour la colonne serial « t2.id »
CREATE TABLE
b1=# INSERT INTO t2 (dateheure, nombre) VALUES (now(), '12.3'::float4);
INSERT 0 1
b1=# INSERT INTO t2 (dateheure, nombre) VALUES ('2009-08-03 22:36:00', random());
INSERT 0 1
b1=# SELECT * FROM t2;
 id |         dateheure          |  nombre
----+----------------------------+----------
  1 | 2009-08-28 15:28:35.292307 |     12.3
  2 | 2009-08-03 22:36:00        | 0.232582
(2 lignes)
```

Vérifions maintenant ce que cela a donné sur le deuxième nœud:

```
postgres@debian1:~$ psql -h debian2 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# SELECT * FROM t2;
 id |         dateheure          |  nombre
----+----------------------------+----------
  1 | 2009-08-28 15:57:30.987926 |     12.3
  2 | 2009-08-03 22:36:00        | 0.0219182
(2 lignes)
```

Premier constat, la table est bien là. Deuxième constat, nous avons bien les deux lignes. Troisième constat, la colonne serial est bien gérée. Ce n'est pas très étonnant vu que nous n'avons pas des dizaines de clients en train de faire des insertions. De plus, par défaut, nous sommes en mode insert_lock. Ce paramètre du fichier pgpool.conf permet de verrouiller la table pendant l'insertion. PgPool-II fait cela en remplaçant l'instruction INSERT par la série d'instructions suivantes: BEGIN; LOCK TABLE…; INSERT…; COMMIT;. Cela peut rapidement poser des problèmes de performance. Ces problèmes sont cependant moins présents à partir de la version 2.2. En effet, cette version ne réalise réellement le verrouillage que si elle détecte une donnée de type serial dans la table. Sur les versions précédentes, c'était fait en permanence à partir du moment où ce paramètre était activé.

Revenons-en à notre table t2 et à ses données. Si nous regardons un peu plus précisément les données, nous nous rendons compte de différences:

- l'heure est différente sur la première ligne;

- le nombre est différent sur la seconde.

En effet, pgPool-II envoie les requêtes sur les deux serveurs. Du coup, si cette requête contient un appel à une fonction dont le résultat n'est pas reproductible dans les mêmes conditions, il est possible de voir des différences. Les fonctions now() et random() sont des cas typiques pouvant révéler ce problème. Le moment où now() est exécuté sur le serveur 1 a de très fortes chances d'être différent du moment où cette même fonction est exécutée sur le serveur 2. Quant à random(), son but est d'être justement différent à chaque exécution.

Au passage, on remarquera que les heures sur nos deux serveurs sont vraiment très différentes. Il aurait été bon de mettre en place une synchronisation de l'heure avec le protocole NTP. Cependant, ça ne change rien au problème actuel.

Tout ça pour dire qu'une des limitations de pgPool-II est d'avoir à éviter ce genre de cas. Une solution possible est de récupérer la valeur de now() ou de random() avec un premier appel à un SELECT, puis de construire le INSERT à partir de la valeur renvoyée. En voici un exemple:

```
postgres@debian1:~$ psql -p 9999 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# SELECT now();
              now
-------------------------------
 2009-08-28 15:37:00.69745+02
(1 ligne)

b1=# SELECT random();
       random
--------------------
 0.0262860944494605
(1 ligne)

b1=# INSERT INTO t2 (dateheure, nombre) VALUES ('2009-08-28 15:37:00.69745+02', '0.0262860944494605');
INSERT 0 1
b1=# SELECT * FROM t2;
 id |         dateheure          |  nombre
----+----------------------------+----------
  1 | 2009-08-28 15:28:35.292307 |      12.3
  2 | 2009-08-03 22:36:00        |  0.232582
  3 | 2009-08-28 15:37:00.69745  | 0.0262861
(3 lignes)

b1=# \q
postgres@debian1:~$ psql -h debian2 b1
psql (8.4.0)
Saisissez « help » pour l'aide.

b1=# SELECT * FROM t2;
 id |         dateheure          |  nombre
----+----------------------------+----------
  1 | 2009-08-28 15:57:30.987926 |      12.3
  2 | 2009-08-03 22:36:00        | 0.0219182
  3 | 2009-08-28 15:37:00.69745  | 0.0262861
(3 lignes)
```

Autre inconvénient majeur de cette forme de réplication, c'est que vous devez absolument passer par le pooler pour que la réplication se fasse. Dans notre exemple, si nous n'indiquons pas le port 9999, nous nous connectons directement à PostgreSQL et nous pouvons faire toutes les modifications voulues sans qu'elles soient répliquées. Ceci est vrai aussi sur l'esclave. Il n'est pas bloqué en lecture seule. Donc il faut faire particulièrement attention à la façon dont le système est configuré pour ne pas avoir une réplication qui laisse à désirer. Le meilleur moyen de se protéger est d'interdire tous les accès ne provenant pas du serveur pgPool-II. Le fichier pg_hba.conf devra donc n'accepter que l'adresse IP du serveur exécutant le démon pgpool.

Configuration de la répartition de charge

La réplication est un système intéressant, surtout dans le cas d'outils permettant d'accéder aux esclaves en lecture seule. Ça permet par exemple de déporter les sauvegardes sur l'esclave. Ça permet aussi de mettre en place la génération de rapports (activité généralement lourde pour l'activité CPU et l'utilisation de la mémoire et des disques) sur l'esclave.

Autre intérêt des esclaves accessibles en lecture seule, la possibilité de gérer une répartition de la charge entre le maître et le (ou les) esclave(s). pgPool-II peut s'en charger.

Pour activer ce mode, il faut passer à 'on' le paramètre load_balance_mode:

```
load_balance_mode = true
```

Ce mode activé, les requêtes SELECT seront distribuées parmi les différents nœuds de la base de données. Nous avons déjà expliqué comment définir un nœud. La seule différence est la possibilité de configurer un poids pour chacun des nœuds. Ce poids permet de connaître le pourcentage des requêtes qui seront envoyées sur un nœud par rapport aux autres nœuds. Disons que nous avons trois nœuds, le premier avec un poids de 0,2, le deuxième avec un poids de 0,3 et le dernier avec un poids de 0,5. Le premier recevra 20% des requêtes de lecture, le second 30% de ces mêmes requêtes et le troisième 50% des requêtes en lecture. L'intérêt est surtout de différencier le taux des requêtes en lecture sur le maître par rapport à celui des esclaves.

Il faut aussi savoir que toutes les requêtes SELECT ne sont pas bonnes à répartir. Il existe quelques exceptions que voici:

- les SELECT compris dans une transaction explicite (car au moment de l'ouverture de la transaction, il est impossible de savoir si la transaction sera en lecture seule ou non);
- SELECT nextval et SELECT setval;
- SELECT INTO;
- SELECT FOR UPDATE et SELECT FOR SHARE.

Les requêtes qui sont considérées bonnes pour la répartition sont:

- les requêtes qui commencent par SELECT à l'exception des quatre variantes ci-dessus;
- les requêtes COPY TO STDOUT;
- les requêtes DECLARE … SELECT;
- les requêtes FETCH;
- les requêtes CLOSE.

Attention, n'importe quelle caractère avant le mot SELECT fera que l'instruction ne sera pas envoyée sur un autre nœud que le maître. Quelque fois, le mot SELECT est précédé d'un commentaire ou d'un ou plusieurs espaces. Pour se protéger contre la non répartition des requêtes SELECT commençant par des espaces, il existe un paramètre. En activant ignore_leading_white_space, vous évitez ce comportement.

Il existe un excellent graphique montrant le processus de décision de la répartition de charge à partir du contenu d'une requête. Il est disponible sur http://pgpool.projects.postgresql.org/pgpool-II/doc/load_balance.png.

Il est à noter que si vous préférez utiliser un outil de réplication autre que pgPool-II, vous pouvez malgré tout bénéficier de la répartition de charge. L'autre outil de réplication peut être Slony, Londiste, bref tous ceux qui permettent un accès à l'esclave en lecture seule. Vous devez désactiver load_balance_mode (qui n'est pas activable si replication_mode est désactivé), et activer le paramètre master_slave_mode.

```
replication_mode = false
load_balance_mode = false
master_slave_mode = true
```

Après avoir relancé pgPool-II, vous pourrez voir vos requêtes de lecture réparties entre les différents nœuds définis.

Utilisation de la répartition de charge

Considérons deux serveurs, debian1 et debian2. Nous allons les mettre en réplication via pgPool-II et en répartition de charge pour les requêtes en lecture. Voici la partie la plus intéressante de la configuration de pgPool-II:

```
replication_mode = true
load_balance_mode = true
connection_cache = true
backend_hostname0 = '192.168.10.66'
backend_port0 = 5432
backend_weight0 = 0.5
backend_hostname1 = '192.168.10.67'
backend_port1 = 5432
backend_weight1 = 0.5
```

La configuration des accès aux serveurs PostgreSQL est identique à celle qui a été faite au début de cet article. Pour ce qui est de la configuration générale de PostgreSQL, nous allons configurer un paramètre supplémentaire. Nous allons passer log_statement à 'all' pour tracer toutes les requêtes SQL sur les deux serveurs.

Nous allons maintenant créer une nouvelle base, y ajouter une table et insérer quelques données dans cette table:

```
postgres@debian1:~$ createdb -p 9999 repli1
postgres@debian1:~$ psql -p 9999 repli1
psql (8.4.0)
Saisissez « help » pour l'aide.

repli1=# CREATE TABLE t1 (id integer);
CREATE TABLE
repli1=# INSERT INTO t1 SELECT i FROM generate_series(1, 1000) AS i;
INSERT 0 1000
```

Les traces de debian1 indiquent:

```
2009-08-28 20:55:06 CEST LOG:  instruction : CREATE DATABASE repli1;
2009-08-28 20:59:00 CEST LOG:  instruction : BEGIN
2009-08-28 20:59:00 CEST LOG:  instruction : CREATE TABLE t1 (id integer);
2009-08-28 20:59:00 CEST LOG:  instruction : COMMIT
2009-08-28 20:59:26 CEST LOG:  instruction : BEGIN
2009-08-28 20:59:26 CEST LOG:  instruction : SELECT count(*) FROM pg_catalog.pg_attrdef AS d, pg_catalog.pg_class AS c WHERE d.adrelid = c.oid AND d.adsrc ~ 'nextval' AND c.relname = 't1'
2009-08-28 20:59:26 CEST LOG:  instruction : INSERT INTO t1 SELECT i FROM generate_series(1, 1000) AS i;
2009-08-28 20:59:26 CEST LOG:  instruction : COMMIT
```

Alors que celles de debian2 montrent:

```
2009-08-28 21:53:17 CEST LOG:  instruction : CREATE DATABASE repli1;
2009-08-28 21:57:30 CEST LOG:  instruction : BEGIN
2009-08-28 21:57:30 CEST LOG:  instruction : CREATE TABLE t1 (id integer);
2009-08-28 21:57:30 CEST LOG:  instruction : COMMIT
2009-08-28 21:57:58 CEST LOG:  instruction : BEGIN
2009-08-28 21:57:58 CEST LOG:  instruction :  INSERT INTO t1 SELECT i FROM generate_series(1, 1000) AS i;
2009-08-28 21:57:58 CEST LOG:  instruction : COMMIT
```

La requête de création de la base est bien arrivée aux deux serveurs PostgreSQL. L'ordre de création de la table a été modifiée pour être intégré dans une transaction explicite. L'ordre d'insertion a subi le même sort. Néanmoins, une autre instruction SQL s'est glissé dans la transaction pour vérifier qu'il n'y avait pas colonnes avec une valeur par défaut utilisant la fonction nextval (typique pour les colonnes auto-incrémentées). Il faut aussi remarquer que cette recherche ne se fait que sur le maître. Le nœud 2 n'a que l'ordre INSERT dans sa transaction explicite. Maintenant, exécutons plusieurs SELECT:

```
repli1=# SELECT * FROM t1 WHERE id=1;
 id
----
  1
(1 ligne)
```

Les traces nous indiquent que debian2 a pris la main:

```
2009-08-28 22:52:11 CEST LOG:  instruction : select * from t1 where id=1;
```

Laissons cette session tranquille et ouvrons-en une autre:

```
postgres@debian1:~$ psql -p 9999 repli1
psql (8.4.0)
Saisissez « help » pour l'aide.

repli1=# SELECT * FROM t1 WHERE id=50;
 id
----
 50
(1 ligne)
```

Et cette fois les traces nous disent que c'est debian2 qui s'en est occupé:

```
2009-08-28 21:11:44 CEST LOG:  instruction : SELECT * FROM t1 WHERE id=50;
```

Pour avoir une idée plus complète d'une répartition de charge pour un grand nombre de requêtes, installons pgbench:

```
debian1:/opt/pgpool-II-2.2.4/bin# aptitude install postgresql-contrib-8.4
[… messages de progression …]
```

Créons une base bench, et initialisons-la grâce au mode -i de pgbench:

```
postgres@debian1:~$ createdb -p 9999 bench
postgres@debian1:~$ /usr/lib/postgresql/8.4/bin/pgbench -i -p 9999 bench -s 10
NOTICE:  la table « pgbench_branches » n'existe pas, poursuite du traitement
NOTICE:  la table « pgbench_tellers » n'existe pas, poursuite du traitement
NOTICE:  la table « pgbench_accounts » n'existe pas, poursuite du traitement
NOTICE:  la table « pgbench_history » n'existe pas, poursuite du traitement
creating tables...
10000 tuples done.
[… messages de progression …]
990000 tuples done.
1000000 tuples done.
set primary key...
NOTICE:  ALTER TABLE / ADD PRIMARY KEY créera un index implicite « pgbench_branches_pkey » pour la table « pgbench_branches »
NOTICE:  ALTER TABLE / ADD PRIMARY KEY créera un index implicite « pgbench_tellers_pkey » pour la table « pgbench_tellers »
NOTICE:  ALTER TABLE / ADD PRIMARY KEY créera un index implicite « pgbench_accounts_pkey » pour la table « pgbench_accounts »
vacuum...done.
```

Parfait. Avant de lancer les tests de pgbench, nous allons vider les journaux applicatifs de PostgreSQL:

```
debian1:/var/log/postgresql# /etc/init.d/postgresql-8.4 stop
Stopping PostgreSQL 8.4 database server: main.
debian1:/var/log/postgresql# >postgresql-8.4-main.log
debian1:/var/log/postgresql# /etc/init.d/postgresql-8.4 start
Starting PostgreSQL 8.4 database server: main.
```

Il faut faire de même sur debian2.

Nous allons maintenant utiliser pgbench en lui demandant de ne faire que des SELECT (mode -S). pgbench va simuler huit clients. Il faut donc s'assurer que le paramètre num_init_children de la configuration de pgPool-II vaut au minimum 8. Pour être sûr de ne pas avoir de concurrence entre les différents clients, nous avons doublé cette valeur. Relancez le démon pgpool si vous avez dû modifier la configuration.

Maintenant, nous pouvons lancer pgbench:

```
postgres@debian1:~$ /usr/lib/postgresql/8.4/bin/pgbench -p 9999 -c 8 -t 1000 -S bench
starting vacuum...end.
transaction type: SELECT only
scaling factor: 10
query mode: simple
number of clients: 8
number of transactions per client: 1000
number of transactions actually processed: 8000/8000
tps = 563.613149 (including connections establishing)
tps = 574.091165 (excluding connections establishing)
```

Regardons le nombre de SELECT dans les journaux de debian1:

```
debian1:/var/log/postgresql# grep "SELECT abalance" postgresql-8.4-main.log | wc -l
4000
```

Et dans ceux de debian2:

```
debian1:/var/log/postgresql# grep "SELECT abalance" postgresql-8.4-main.log | wc -l
4000
```

Nous avons bien du 50/50. Maintenant, changeons le poids. Disons 0,3 pour debian1 et 0,7 pour debian2, vidons les journaux applicatifs sur les deux serveurs, puis relançons le test.

On obtient cette fois 2000 transactions par debian1 et 6000 par debian2. Ce n'est pas exactement du 30/70, mais plusieurs facteurs ont pu jouer pour que cela ne corresponde pas exactement.

Petit récapitulatif

Avantages majeurs

- extrêmement simple à mettre en place
- très facile à comprendre
- réplication des changements de schéma
- les esclaves en lecture seule

Inconvénients majeurs

- aucune garantie que les données soient identiques entre les nœuds si l'applicatif utilise des fonctions comme now(), random(), ou des types comme les OID, TID

les tables temporaires ne sont pas supprimées automatiquement à la fin d'une session

- les clients doivent obligatoirement passer par le pooler

Conclusion

pgPool-II n'est pas le seul outil proposant ce type de réplication. On pourra notamment citer sequoia/tungsten et ha-jdbc (http://ha-jdbc.sourceforge.net/). Lequel utiliser dépend de votre contexte. Le bon point de pgPool-II est qu'il est capable de faire d'autres choses, comme du pooling de connexions ou de la répartition de charge.

Il est aussi à noter que nous n'avons pas tout vu de pgPool-II. Notamment, il resterait à voir les outils PCP et la base de données système.