DALIBO

L'Expertise PostgreSQL

PostgreSQL 9.0 et la réplication

Table des matières

PostgreSQL 9.0 et la réplication

1 À propos des auteurs...

» Auteur : Guillaume Lelarge
» Société : DALIBO
» Date : Avril 2011
» URL : https://support.dalibo.com/kb/conferences/postgresql_9.0_haute_dispo/

2 Licence

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

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libre de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Ceci est un résumé explicatif du Code Juridique. La version intégrale du contrat est disponible ici : http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

3 Version 9.0

- Sortie le 20 septembre 2010
- Dernière version corrective : 9.0.4 (18/04/2011)
- 15 mois de développement
- 4 versions beta, 1 version RC
- Au menu :
  - Réplication
  - Nouvelles fonctionnalités
  - Amélioration des performances

3.1 Réplication intégrée

Hot Standby
+ Streaming Replication
= Oracle Active Data Guard

La réplication est LA grande nouveauté de PostgreSQL 9.0 !

Attendue depuis des années, ces deux nouveautés (Hot Standby et Streaming Replication) sont celles qui ont justifié à elles seules le renommage de la version 8.5 en 9.0.

La réplication en question est de la réplication asynchrone asymétrique. Autrement dit, un maître associé à un ou plusieurs esclaves qui reçoivent les données un peu après que la validation des modifications soit envoyée au client.

Nous détaillerons cette réplication dans la partie « Réplication par les journaux de transactions ».

3.2 Autres nouveautés

- Version 64 bits pour Windows
- Contraintes d'exclusion
- Triggers avec conditions
- Contraintes uniques différables
- Fonctions anonymes
- Meilleure gestion des droits

3.2.1 64 bits sous Windows

Il y a maintenant une version 64 bits native pour Windows.

Pour l'instant, peu de mesures de performance ont été effectuée pour en connaître les gains. Néanmoins, le gain attendu se trouve dans la quantité de mémoire accessible. Cela concerne assez peu le cache disque de PostgreSQL. Ce dernier étant en mémoire partagée et Windows ayant une gestion particulière de celle-ci, des interrogations sérieuses subsistent toujours concernant les performances lorsque le paramètre shared_buffers (indiquant la taille du cache disque de PostgreSQ) est élevé au delà de 500 Mo sous Windows. Par contre, d'autres paramètres, comme work_mem et maintenance_work_mem, vont pouvoir être augmentés sérieusement.

Notons aussi qu'il faut faire très attention à la version de Windows pour s'assurer de pouvoir profiter du maximum de mémoire. Cet article du MSDN (http://msdn.microsoft.com/en-us/library/aa366778%28VS.85%29.aspx) explique les différentes tailles utilisables suivant la version de Windows et la license.

Malgré cette avancée, pour un matériel équivalent, PostgreSQL reste plus performant sous Linux que sous Windows Server.

3.2.2 Contraintes d'exclusion

Il est maintenant possible de déclarer des contraintes d'unicité plus complexes que celles s'appuyant sur l'opérateur '='. Une contrainte d'unicité est une contrainte sur un jeu de colonnes ne pouvant être identiques. Les contraintes d'exclusion permettent de faire varier les opérateurs.

Nous allons, pour l'illustrer, utiliser l'exemple de l'auteur, Jeff Davis, en utilisant le type 'temporal' qu'il a aussi développé. Ce type de données permet de définir des plages de temps, c'est-à-dire par exemple la plage de 12h15 à 13h15. Il faut donc récupérer le module temporal à l'adresse suivante : http://pgfoundry.org/projects/temporal/ , le compiler et l'installer comme tout module contrib (notamment en exécutant le script SQL fourni).

```
CREATE TABLE reservation
(
  salle      TEXT,
  professeur TEXT,
```

```
  periode    PERIOD
);

ALTER TABLE reservation
  ADD CONSTRAINT exclusion1
  EXCLUDE USING gist (salle WITH =, periode WITH &&);
```

Par ceci, nous disons qu'un enregistrement doit être refusé (contrainte d'exclusion) s'il en existe déjà un vérifiant les deux conditions (même salle et intersection au niveau de l'intervalle de temps).

```
marc=> INSERT INTO reservation (professeur, salle, periode)
       VALUES ( 'marc', 'salle techno', period('2010-06-16 09:00:00', '2010-06-16 10:00:00'));
INSERT 0 1
marc=> INSERT INTO reservation (professeur, salle, periode)
       VALUES ( 'jean', 'salle chimie', period('2010-06-16 09:00:00', '2010-06-16 11:00:00'));
INSERT 0 1
marc=> INSERT INTO reservation (professor, room, periode)
       VALUES ( 'marc', 'salle chimie', period('2010-06-16 10:00:00', '2010-06-16 11:00:00'));
ERROR:  conflicting KEY value violates exclusion constraint "test_exclude"
DETAIL:  KEY (salle, periode)=(salle chimie, [2010-06-16 10:00:00+02, 2010-06-16 11:00:00+02))
conflicts WITH existing KEY (salle, periode)=(salle chimie, [2010-06-16 09:00:00+02, 2010-06-16
11:00:00+02)).
```

L'insertion est interdite puisque la salle de chimie est déjà prise de 9h à 11h.

3.2.3 Triggers

Les développeurs de PostgreSQL ont ajouté la possibilité de définir des trigger avec des conditions plus évoluées, comme le déclenchement suite à la mise à jour d'une colonne spécifique ou un déclenchement à condition qu'une expression soit vraie.

Voici d'abord un trigger par colonne.

```
CREATE TRIGGER toto
  BEFORE UPDATE OF c ON t
  FOR EACH ROW
  EXECUTE PROCEDURE mon_trigger();
```

Ce trigger ne se déclenche que si la colonne c de la table t a été modifiée.

Par ailleurs, les triggers disposent d'une autre clause conditionnelle. La clause WHEN permet justement de placer une condition d'activation du trigger. Les mots-clefs `new` et `old` sont utilisables dans l'expression indiquée par la clause WHEN. Ils permettent d'accéder respectivement à la nouvelle et l'ancienne version de la ligne impactée.

Voici maintenant des exemples tirés de la documentation officielle pour la clause WHEN des triggers:

```
CREATE TRIGGER verification_mise_a_jour
  BEFORE UPDATE ON comptes
  FOR EACH ROW
  WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
  EXECUTE PROCEDURE verification_mise_a_jour_compte();
```

Ici, le trigger est positionné sur l'UPDATE de la table `compte`. Il ne se déclenche que si la colonne `balance` est modifiée.

Le caractère joker * permet de cibler l'ensemble des colonnes de la table :

```
CREATE TRIGGER trace_mise_a_jour
  AFTER UPDATE ON comptes
  FOR EACH ROW
  WHEN (OLD.* IS DISTINCT FROM NEW.*)
  EXECUTE PROCEDURE trace_mise_a_jour_compte();
```

3.2.4 Contraintes UNIQUE différée

Cette fonctionnalité existe déjà dans les versions précédentes pour les contraintes de type clé étrangère. Elle a été étendue aux clés primaires et aux contraintes d'unicité.

Voici un exemple avec une clé primaire au lieu d'une simple clé unique, mais le concept reste le même :

```
marc=> CREATE TABLE test (a int PRIMARY KEY);
marc=> INSERT INTO test VALUES (1), (2);
marc=> UPDATE test SET a = a+1;
ERROR:  duplicate KEY value violates UNIQUE constraint "test_pkey"
DETAIL:  KEY (a)=(2) already EXISTS.
```

C'est tout à fait normal, mais bien dommage : à la fin de la transaction, les données auraient été cohérentes (valeurs 2 et 3). D'autant plus que si la table avait été triée physiquement par ordre descendant, la requête aurait fonctionné !

Avec PostgreSQL 8.4, il n'y avait pas d'échappatoire simple, il fallait trouver une astuce pour mettre à jour les enregistrements dans le bon ordre.

Avec PostgreSQL 9.0, nous pouvons maintenant faire ceci :

```
marc=> CREATE TABLE test (a int PRIMARY KEY DEFERRABLE);
marc=> INSERT INTO test VALUES (2),(1);
marc=> UPDATE test SET a = a+1;
ERROR:  duplicate KEY value violates UNIQUE constraint "test_pkey"
DETAIL:  KEY (a)=(2) already EXISTS.
```

Ah zut, ça ne marche pas 

Profitons pour faire un petit rappel sur les contraintes deferrable/deferred, une contrainte 'deferrable' PEUT être vérifiée en fin de transaction (sa vérification peut être repoussée à la fin de la transaction). Il faut toutefois dire à PostgreSQL expressément qu'on veut vraiment faire ce contrôle en fin de transaction.

On peut, pour la session en cours, demander à passer toutes les contraintes en 'DEFERRED' :

```
marc=> SET CONSTRAINTS ALL DEFERRED;
SET CONSTRAINTS
marc=> UPDATE test SET a = a+1;
UPDATE 2
```

Si on veut ne pas avoir à effectuer le SET CONSTRAINTS à chaque fois, il est aussi possible de déclarer la contrainte comme INITIALLY DEFERRED:

```
CREATE TABLE test (a int PRIMARY KEY DEFERRABLE INITIALLY DEFERRED);
```

Un autre rappel s'impose : les contraintes DEFERRED sont plus lentes que les contraintes IMMEDIATE. Par ailleurs, il faut bien stocker la liste des enregistrements à vérifier en fin de transaction quelque part, ce qui consomme de la mémoire. Attention à ne pas le faire sur des millions d'enregistrements d'un coup. C'est la raison pour laquelle les contraintes 'DEFERRABLE' ne sont pas 'INITIALLY DEFERRED' par défaut.

3.2.5 Fonctions anonymes

Cette nouvelle fonctionnalité permet de créer des fonctions à usage unique. Elles seront très pratiques dans des scripts de livraison de version applicative par exemple.

Voici une version un peu différente du GRANT SELECT ON ALL TABLES qui sera présenté plus loin dans ce document, qui donne le droit de sélection à tout un jeu de tables, en fonction du propriétaire des tables, et en ignorant deux schémas :

```
DO LANGUAGE plpgsql $$
DECLARE
  vr record;
BEGIN
  FOR vr IN SELECT tablename FROM pg_tables WHERE tableowner = 'marc' AND schemaname NOT IN
('pg_catalog','information_schema')
  LOOP
    EXECUTE 'GRANT SELECT ON ' || vr.tablename || ' TO toto';
  END LOOP;
END
$$
;
```

Avec PostgreSQL 8.4, il aurait fallu créer une fonction (via CREATE FUNCTION), l'exécuter puis la supprimer (avec DROP FUNCTION). Le tout demandant d'avoir les droits pour ça. La version 9.0 facilite donc ce type d'exécution rapide.

3.2.6 Gestion des droits

Tout d'abord, voici la fin d'un problème idiot et un peu frustrant, qui est déjà arrivé à beaucoup d'administrateurs de base de données : créer 400 tables, puis devoir attribuer des droits à un utilisateur sur ces 400 tables. Jusque là, on en était quitte pour créer un script. Plus maintenant :

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO toto;
```

Et la marche arrière :

```
REVOKE SELECT ON ALL TABLES IN SCHEMA public FROM toto;
```

Bien sûr, cette commande ne vaut que pour les tables en place au moment de la commande. Il faudra toujours faire de nouveaux GRANT pour les futures tables du schéma. Pour cela, une nouvelle commande permet de gagner du temps dans la gestion des droits en définissant les droits par défaut d'un utilisateur :

```
ALTER DEFAULT PRIVILEGES FOR ROLE marc GRANT SELECT ON TABLES TO PUBLIC ;
CREATE TABLE test_priv (a int);
\z test_priv
```

```
                          Access privileges
 Schema |     Name     | Type  | Access privileges | Column access privileges
--------+--------------+-------+--------------------+--------------------------
 public | test_priv    | table | =r/marc           +|
        |              |       | marc=arwdDxt/marc  |
```

Les informations sur les droits par défaut sont stockées dans le catalogue système `pg_default_acl`.

3.3 Améliorations

- Planificateur optimisé
- VACUUM plus efficace
- EXPLAIN plus souple
- Davantages de statistiques

3.3.1 Planificateur

Le planificateur de requête a reçu un grand nombre d'améliorations dans cette version. Nous allons donc commencer par lui:

```
marc=> CREATE TABLE t1 (a int);
CREATE TABLE
marc=> CREATE TABLE t2 (b int);
CREATE TABLE
marc=> CREATE TABLE t3 (c int);
CREATE TABLE
```

On insère quelques données avec la procédure stockée interne `generate_series()`…

```
marc=> EXPLAIN SELECT t1.a,t2.b FROM t1 JOIN t2 ON (t1.a=t2.b) LEFT JOIN t3 ON (t1.a=t3.c);
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Merge RIGHT JOIN  (cost=506.24..6146.24 rows=345600 width=8)
   Merge Cond: (t3.c = t1.a)
   ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
         Sort KEY: t3.c
         ->  Seq Scan ON t3  (cost=0.00..34.00 rows=2400 width=4)
   ->  Materialize  (cost=337.49..853.49 rows=28800 width=8)
         ->  Merge JOIN  (cost=337.49..781.49 rows=28800 width=8)
               Merge Cond: (t1.a = t2.b)
               ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
                     Sort KEY: t1.a
                     ->  Seq Scan ON t1  (cost=0.00..34.00 rows=2400 width=4)
               ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
                     Sort KEY: t2.b
                     ->  Seq Scan ON t2  (cost=0.00..34.00 rows=2400 width=4)
```

On a le même comportement qu'en 8.4, c'est normal. Mais imaginons que sur la table t3, on ait une contrainte UNIQUE sur la colonne c. Dans ce cas, théoriquement, la jointure sur la table t3 ne sert à rien : le nombre d'enregistrements du résultat ne sera pas modifié, pas plus, bien sûr,

que leur contenu. C'est lié au fait que la colonne est UNIQUE, que la jointure est un LEFT JOIN et qu'aucune colonne de t3 n'est récupérée. Si la colonne n'était pas UNIQUE, la jointure pourrait augmenter le nombre d'enregistrements du résultat. Si ce n'était pas un LEFT JOIN, la jointure pourrait diminuer le nombre d'enregistrements du résultat.

Avec PostgreSQL 9.0 :

```
marc=> ALTER TABLE t3 ADD UNIQUE (c);
NOTICE:  ALTER TABLE / ADD UNIQUE will CREATE implicit INDEX "t3_c_key" FOR TABLE "t3"
ALTER TABLE
marc=> EXPLAIN SELECT t1.a,t2.b FROM t1 JOIN t2 ON (t1.a=t2.b) LEFT JOIN t3 ON (t1.a=t3.c);
                            QUERY PLAN
------------------------------------------------------------------
 Merge JOIN  (cost=337.49..781.49 rows=28800 width=8)
   Merge Cond: (t1.a = t2.b)
   ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
         Sort KEY: t1.a
         ->  Seq Scan ON t1  (cost=0.00..34.00 rows=2400 width=4)
   ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
         Sort KEY: t2.b
         ->  Seq Scan ON t2  (cost=0.00..34.00 rows=2400 width=4)
(8 rows)
```

Cette optimisation devrait pouvoir être très rentable, entre autres quand les requêtes sont générées par un ORM (mapping objet-relationnel). Ces outils ont la fâcheuse tendance à exécuter des jointures inutiles. Ici on a réussi à diviser le coût estimé de la requête par 10.

C'est aussi une optimisation qui pourra être très utile pour les applications utilisant beaucoup de jointures et de vues imbriquées.

Cela constitue encore une raison supplémentaire de déclarer les contraintes dans la base : sans ces contraintes, impossible pour le moteur d'être sûr que ces réécritures sont possibles.

3.3.2 Des VACUUMs plus efficaces

La commande VACUUM FULL était jusque ici très lente. Cette commande permet de récupérer l'espace perdu dans une table, principalement quand la commande VACUUM n'a pas été passée très régulièrement. Ceci était du à son mode de fonctionnement : les enregistrements étaient lus et déplacés un par un d'un bloc de la table vers un bloc plus proche du début de la table. Une fois que la fin de la table était vide, l'enveloppe était réduite à sa taille minimale.

Le problème était donc que ce mécanisme était très inefficace : le déplacement des enregistrements un à un entraîne beaucoup d'entrées/sorties aléatoires (non contigues). Par ailleurs, durant cette réorganisation, les index doivent être maintenus, ce qui rend l'opération encore plus coûteuse, et fait qu'à la fin d'un VACUUM FULL, les index sont fortement désorganisés. Il est d'ailleurs conseillé de réindexer une table juste après y avoir appliqué un VACUUM FULL.

La commande VACUUM FULL, dans cette nouvelle version, crée une nouvelle table à partir de la table actuelle, en y recopiant tous les enregistrements de façon séquentielle. Une fois tous les enregistrements recopiés, les index sont recréés, et l'ancienne table détruite.

Cette méthode présente l'avantage d'être très largement plus rapide. Toutefois, VACUUM FULL demande toujours un verrou complet sur la table durant le temps de son exécution. Le seul défaut de cette méthode par rapport à l'ancienne, c'est que pendant le temps de son exécution, le nouveau VACUUM FULL peut consommer jusqu'à 2 fois l'espace disque de la table, puisqu'il en crée une nouvelle version.

Mesurons maintenant le temps d'exécution suivant les deux méthodes. Dans les deux cas, on prépare le jeu de test comme suit (en 8.4 et en 9.0):

```
marc=> CREATE TABLE test (a int);
CREATE TABLE
marc=> CREATE INDEX idxtsta ON test (a);
CREATE INDEX
marc=> INSERT INTO test SELECT generate_series(1, 1000000);
INSERT 0 1000000
marc=> DELETE FROM test WHERE a%3=0;
DELETE 333333
marc=> VACUUM test;
VACUUM
```

En 8.4 :

```
marc=> \timing
Timing IS ON.
marc=> VACUUM FULL test;
VACUUM
Time: 6306,603 ms
marc=> REINDEX TABLE test ;
REINDEX
Time: 1799,998 ms
```

Soit environ 8 secondes.

En 9.0 :

```
marc=> \timing
Timing IS ON.
marc=> VACUUM FULL test;
VACUUM
Time: 2563,467 ms
```

Soit pratiquement quatre fois plus rapide.

Pour autant, cela ne veut toujours pas dire que VACUUM FULL est une bonne idée en production. Si vous en avez besoin, c'est probablement que votre politique de VACUUM n'est pas appropriée.

Enfin une autre amélioration est à signaler, la commande système vacuumdb acquiert une nouvelle option permettant de faire seulement un ANALYZE.

```
vacuumdb --analyze-only
```

Comme son nom l'indique, on peut maintenant utiliser vacuumdb pour passer des analyses uniquement. Cela peut être pratique dans une crontab.

3.3.3 EXPLAIN

Voici un EXPLAIN ANALYZE dans les versions précédentes :

```
marc=> EXPLAIN ANALYZE SELECT a, sum(c) FROM pere JOIN fils ON (pere.a = fils.b) WHERE b BETWEEN
1000 AND 300000 GROUP BY a;                                                                  QUERY PLAN

-------------------------------------------------------------------------------------------------------------
------------------------------
```

```
 HashAggregate  (cost=905.48..905.86 rows=31 width=8) (actual time=0.444..0.453 rows=6 loops=1)
   ->  Nested Loop  (cost=10.70..905.32 rows=31 width=8) (actual time=0.104..0.423 rows=6 loops=1)
         ->  Bitmap Heap Scan ON fils  (cost=10.70..295.78 rows=31 width=8) (actual
time=0.040..0.154 rows=30 loops=1)
               Recheck Cond: ((b >= 1000) AND (b <= 300000))
               ->  Bitmap INDEX Scan ON fils_pkey  (cost=0.00..10.69 rows=31 width=0) (actual
time=0.023..0.023 rows=30 loops=1)
                     INDEX Cond: ((b >= 1000) AND (b <= 300000))
         ->  INDEX Scan USING pere_pkey ON pere  (cost=0.00..19.65 rows=1 width=4) (actual
time=0.005..0.005 rows=0 loops=30)
               INDEX Cond: (pere.a = fils.b)
 Total runtime: 0.560 ms
(9 rows)
```

Si vous voulez avoir accès aux nouvelles informations, il faut opter pour la nouvelle syntaxe :

```
EXPLAIN [ ( { ANALYZE BOOLEAN | VERBOSE BOOLEAN | COSTS BOOLEAN | BUFFERS BOOLEAN | FORMAT { TEXT |
XML | JSON | YAML } } [, ...] ) ] instruction
```

Par exemple :

```
marc=> EXPLAIN (ANALYZE true, VERBOSE true, BUFFERS true) SELECT a, sum(c) FROM pere JOIN fils ON
(pere.a = fils.b) WHERE b BETWEEN 1000 AND 300000 GROUP BY a;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------
-------------------------------
 HashAggregate  (cost=905.48..905.86 rows=31 width=8) (actual time=1.326..1.336 rows=6 loops=1)
   Output: pere.a, sum(fils.c)
   Buffers: shared hit=58 READ=40
   ->  Nested Loop  (cost=10.70..905.32 rows=31 width=8) (actual time=0.278..1.288 rows=6 loops=1)
         Output: pere.a, fils.c
         Buffers: shared hit=58 READ=40
         ->  Bitmap Heap Scan ON public.fils  (cost=10.70..295.78 rows=31 width=8) (actual
time=0.073..0.737 rows=30 loops=1)
               Output: fils.b, fils.c
               Recheck Cond: ((fils.b >= 1000) AND (fils.b <= 300000))
               Buffers: shared hit=4 READ=28
               ->  Bitmap INDEX Scan ON fils_pkey  (cost=0.00..10.69 rows=31 width=0) (actual
time=0.030..0.030 rows=30 loops=1)
                     INDEX Cond: ((fils.b >= 1000) AND (fils.b <= 300000))
                     Buffers: shared hit=3
         ->  INDEX Scan USING pere_pkey ON public.pere  (cost=0.00..19.65 rows=1 width=4) (actual
time=0.013..0.014 rows=0 loops=30)
               Output: pere.a
               INDEX Cond: (pere.a = fils.b)
               Buffers: shared hit=54 READ=12
 Total runtime: 1.526 ms
(18 rows)
```

- VERBOSE apporte les lignes 'Output' (l'option existait déjà en 8.4)
- BUFFERS indique les opérations sur les buffers (les entrées sorties de la requête): hit correspond aux données lues en cache, read aux données demandées au système d'exploitation. Ici, peu de données étaient en cache.

Vous pouvez aussi demander une sortie dans un autre format que le texte. Pour un utilisateur, cela n'a aucune importance. Pour les développeurs d'interfaces graphiques présentant le résultat d'EXPLAIN, cela permettra de faire l'économie d'un analyseur sur le texte du EXPLAIN (et des bugs qui vont avec).

On peut aussi désactiver l'affichage des coûts en positionnant COSTS à `false`.

3.3.4 Statistiques d'activité

À la connexion, une application peut indiquer son nom. Dans ce cas, il est visible dans les traces et dans la vue système `pg_stat_activity`.

Dans la session de supervision :

```
marc=> SELECT * FROM pg_stat_activity WHERE procpid= 5991;

datid | datname | procpid | usesysid | usename | application_name | client_addr | client_port |
backend_start         | xact_start | query_start | waiting | current_query
------+---------+---------+----------+---------+------------------+-------------+-------------
+-------------------------------+------------+-------------+---------+---------------
16384 | marc    |    5991 |       10 | marc    | psql             |             |          -1 |
2010-05-16 13:48:10.154113+02 |            |             | f       | <IDLE>
(1 row)
```

Dans la session de PID 5991 :

```
marc=> SET application_name TO 'mon_appli';
SET
```

Dans la session de supervision :

```
marc=> SELECT * FROM pg_stat_activity WHERE procpid= 5991;

datid | datname | procpid | usesysid | usename | application_name | client_addr | client_port |
backend_start         | xact_start |          query_start          | waiting | current_query
------+---------+---------+----------+---------+------------------+-------------+-------------
+-------------------------------+------------+-------------------------------+---------+---------------
16384 | marc    |    5991 |       10 | marc    | mon_appli        |             |          -1 |
2010-05-16 13:48:10.154113+02 |            | 2010-05-16 13:49:13.107413+02 | f       | <IDLE>
(1 row)
```

À vous de le positionner correctement dans votre application ou vos sessions. Votre DBA vous dira merci, sachant enfin qui lance quelle requête sur son serveur.

3.4 Modules

- Nouveaux modules
  - pg_upgrade
  - passwordcheck
- Amélioration de certains anciens modules
  - hstore
  - pg_stat_statements
  - auto_explain

3.4.1 Ajout du module contrib pg_upgrade

Le module pg_upgrade a pour but de faciliter les mises à jour de PostgreSQL 8.4 vers PostgreSQL 9.0. Encore mieux, il permet une mise à jour extrêmement rapide.

Il y a trop de choses à dire sur ce module, le mieux est de se reporter à sa documentation.

3.4.2 Ajout du module contrib passwordcheck

Ce module contrib permet de vérifier la force des mots de passe, dans le but d'empêcher les plus mauvais d'être acceptés. Après l'avoir installé comme décrit dans la documentation, voici un exemple d'utilisation :

```
marc=> ALTER USER marc PASSWORD 'marc12';
<marc%marc> ERROR:  password IS too short
<marc%marc> STATEMENT:  ALTER USER marc PASSWORD 'marc12';
ERROR:  password IS too short
marc=> ALTER USER marc PASSWORD 'marc123456';
<marc%marc> ERROR:  password must NOT contain user name
<marc%marc> STATEMENT:  ALTER USER marc PASSWORD 'marc123456';
ERROR:  password must NOT contain user name
```

Ce module souffre de limitations, principalement dues au fait que PostgreSQL permet l'envoi d'un mot de passe déjà chiffré à la base au moment de la déclaration, ce qui l'empêche de le vérifier correctement. Néanmoins, c'est une avancée dans la bonne direction.

Par ailleurs, le code du module contrib est bien documenté, ce qui permet de l'adapter à vos besoins (entre autres, il est très facile d'y activer la cracklib, afin d'effectuer des contrôles plus complexes).

3.4.3 Amélioration du module contrib hstore

Ce module contrib, déjà très pratique, devient encore plus puissant :

- La limite de taille sur les clés et valeurs a été supprimée.
- Il est maintenant possible d'utiliser GROUP BY et DISTINCT.
- De nombreux opérateurs et fonctions ont été ajoutés.

Un exemple serait trop long, tellement ce module est riche. Lisez la documentation sans perdre de temps !

3.4.4 Compteurs sur buffers dans pg_stat_statements

Ce module contrib ajoute quelques compteurs. Pour rappel, son intérêt est de stocker des statistiques sur les requêtes exécutées par le moteur. Jusque là, il donnait la requête, le nombre d'exécution, le temps cumulé et le nombre d'enregistrements cumulés. Maintenant, il collecte aussi des informations sur les entrées sorties (dans le cache, et hors du cache).

```
marc=> SELECT * FROM pg_stat_statements ORDER BY total_time DESC LIMIT 2;
-[ RECORD 1 ]-------+---------------------
userid              | 10
dbid                | 16485
query               | SELECT * FROM fils ;
calls               | 2
```

```
total_time          | 0.491229
rows                | 420000
shared_blks_hit     | 61
shared_blks_read    | 2251
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
-[ RECORD 2 ]-------+--------------------
userid              | 10
dbid                | 16485
query               | SELECT * FROM pere;
calls               | 2
total_time          | 0.141445
rows                | 200000
shared_blks_hit     | 443
shared_blks_read    | 443
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
```

Une fois ce contrib installé, on peut donc répondre aux questions suivantes :

- Quelle est la requête la plus gourmande en temps d'exécution cumulé ?
- Quelle est la requête qui génère le plus d'entrées sorties ? (attention, les données peuvent être tout de même dans le cache système)
- Quelles requêtes utilisent principalement le cache (et ne gagneront donc pas à le voir augmenté)
- Qui effectue beaucoup de mises à jour de bloc ?
- Qui génère des tris ?

`local` et `temp` correspondent aux buffers et entrées des tables temporaires et autres opérations locales (tris, hachages) à un backend.

3.4.5 Amélioration du module contrib auto_explain

Le module contrib auto_explain affiche maintenant le code de la requête en même temps que son plan, ce qui devrait en augmenter la lisibilité.

3.5 Bilan

- La réplication est simple et efficace.
- Plus de 200 nouvelles fonctionnalités et autres améliorations.
- L'administration est beaucoup plus souple.
- Les performances sont meilleures.

4 Solutions de réplication

- But fréquent : pouvoir basculer sur un autre serveur si le premier tombait
- Autres buts
  - Disposer d'un serveur de tests
  - Disposer d'un serveur de génération de rapports
- Types de réplication
- Diffusion des modifications

La réplication a principalement pour but de disposer d'un deuxième serveur, copie du premier, où travailler au cas où le premier tomberait en panne. L'idée est que l'activité ne doit pas être impactée par la perte d'un serveur.

Mais ce n'est pas la seule raison pour laquelle un administrateur veut de la réplication. Le serveur répliqué peut aussi servir à y déporter une partie du travail, histoire d'alléger la charge du serveur principal.

Dans ce cadre, plusieurs types de réplications vont apparaître, chacune permettant d'apporter une solution à un type de problème. Chaque type sera couvert par un ou plusieurs outils de réplication.

4.1 Asynchrone Asymétrique

- Écritures uniquement sur le maître
- Mise à jour différée des tables sur l'autre serveur
- Outils pour PostgreSQL : Slony, Londiste, Bucardo, Replicator, rubyrep

C'est le mode de réplication le plus simple.

Un serveur est en lecture/écriture, il est généralement appelé serveur maître. Le ou les autres serveurs ne sont pas disponibles en écriture. Ils peuvent cependant être disponibles en lecture.

Tout enregistrement fait sur le maître n'est pas immédiatement reporté sur l'esclave. Il existe généralement un petit délai avant application sur l'esclave. Cela sous-entend que, si le serveur maître est tombé en panne avant d'avoir eu le temps de transférer les derniers transactions au serveur esclave, il manquera les données de ces transactions sur l'esclave. Il faut donc être prêt à accepter une certaine perte, généralement petite. De plus, si l'esclave sert à répartir la charge, il est possible qu'une lecture du maître et qu'une lecture de l'esclave ne donnent pas le même résultat.

Étant la solution la plus simple à mettre à œuvre, il existe de nombreux outils pour ce type de réplication : la réplication interne de PostgreSQL par exemple, mais aussi Slony, Londiste, etc.

4.2 Asynchrone Symétrique

- Écritures sur les deux « maîtres »
- Mise à jour différée des tables sur l'autre serveur
- Difficile d'avoir un respect d'ACID
- Outils pour PostgreSQL : Bucardo, rubyrep

Ce mode est plus complexe. Les serveurs sont tous en lecture/écriture. Il faut donc pouvoir gérer les conflits causés par la mise à jour des mêmes objets sur plusieurs serveurs en même temps. Cela complexifie de beaucoup le respect de la norme ACID.

Bucardo implémente néanmoins ce type de système sur deux serveurs uniquement.

4.3 Synchrone Asymétrique

- Écritures uniquement sur le maître
- Mise à jour immédiate des tables sur l'autre serveur
- Source de lenteurs
- Outils pour PostgreSQL : PostgreSQL 9.1, pgPool-II

Ce mode de réplication est plus simple que le précédent, mais plus lent. Il n'y a qu'un seul maître mais chaque modification réalisée sur le maître doit être enregistrée sur l'esclave avant de redonner la main à l'utilisateur. Ce qui est source de lenteurs (deux systèmes doivent avoir enregistrés l'information au lieu d'un seul sans compter les lags réseau possibles).

C'est la meilleure solution s'il est inconcevable de perdre des données suite à l'arrêt inopiné du maître. Par contre, cela ne résout pas complètement le problème de la répartition de charge. En effet, cette solution garantit seulement que la donnée est enregistrée sur les esclaves, pas qu'elle soit visible. Donc, encore une fois, si l'esclave sert à répartir la charge, il est possible qu'une lecture du maître et qu'une lecture de l'esclave ne donnent pas le même résultat, même si le risque est minimisé par rapport aux autres solutions.

La réplication interne de PostgreSQL proposera cette méthode dès la version 9.1. pgPool-II le propose dès maintenant via son mode de réplication. À noter qu'il dispose aussi d'un mode de pooling de connexions et d'un mode de répartition de charge.

4.4 Synchrone Symétrique

- Écritures sur les deux « maîtres »
- Mise à jour immédiate des tables sur l'autre serveur

Ce mode de réplication est le plus complexe et le plus lente. La complexité est due à la gestion des conflits, inévitable quand il y a plusieurs serveurs maîtres. La lenteur est due au côté synchrone du système.

Il n'existe pas d'outils compatibles PostgreSQL pour cette méthode.

5 Réplication par triggers

- Récupération des modifications par des triggers
- Ajout de triggers sur chaque table à répliquer
- Différents outils :
  - Slony
  - Londiste
  - Bucardo

C'est la plus ancienne méthode de réplication disponible avec PostgreSQL.

Par l'ajout de triggers sur les tables à répliquer, il est possible de connaître les données modifiées. Ces données sont stockées temporairement (généralement dans une table spécifique à la réplication), puis envoyées sur l'esclave où elles sont enregistrées.

Il existe différents outils qui utilisent ce moyen de réplication : Slony (certainement l'outil de réplication le plus ancien sur PostgreSQL), Londiste (développé par Skype, mais libre), Bucardo sont les principaux exemples.

5.1 Slony - Introduction

- Premier outil de réplication libre (PostgreSQL)
- Créé par un développeur majeur de PostgreSQL
- Licence BSD
- Dernières versions
  - 1.2.22 (6 décembre 2010) : à partir de PostgreSQL 7.3
  - 2.0.6 (6 décembre 2010) : à partir de PostgreSQL 8.3
- http://slony.info/

Slony est le premier outil à proposer de la réplication pour PostgreSQL. Il est conçu par un ancien membre de la Core Team, Jan Wieck.

Il dispose de deux branches. Certaines nouveautés de la version 8.3 ont une implication importante au niveau des outils de réplication par trigger. Seulement, gérer ces nouvelles fonctionnalités changent de beaucoup le code d'un outil comme Slony. Il a donc été décidé de créer deux branches qui continuent à vivre séparément. La branche 1.2 ne gère pas ces nouveautés de la version 8.3 de PostgreSQL mais est compatible avec cette version (et aussi les versions suivantes, comme la 8.4 et la 9.0). La branche 2.0 gère ces nouveautés. Elle a mis un peu de temps à se stabiliser. La version 2.0.6 est actuellement utilisable en production.

5.2 Slony - Techniques

- Réplication par trigger
- 1 maître sur un ensemble de tables et séquences
- Les autres nœuds sont esclaves pour cet ensemble…
- Mais peuvent être maîtres sur d'autres ensembles
- Cascade des serveurs possible

Slony récupère les modifications grâce à des triggers posés sur les tables à répliquer du serveur maître. Ces triggers ont pour but de récupérer toute modification faite sur les tables en question. Les informations sont conservées dans des tables de log. Les données des tables de log sont envoyés aux serveurs secondaires par des démons qui interrogent les tables de log à une fréquence configurable. Le serveur esclave se voit aussi posé des triggers pour empêcher toute modification des données par d'autres programmes que le démon slon.

Comme il est possible de spécifier les tables à répliquer, ces tables sont intégrées dans un « set » qui se voit attribuer un serveur maître. Du coup, il est possible d'écrire une configuration où un serveur est maître de certaines tables et un autre serveur maître pour d'autres tables.

Enfin, il est possible de configurer Slony pour qu'une réplication en cascade soit mise en place.

5.3 Slony - Avantages

- Choix des objets à répliquer
- Esclaves en lecture seule
- Mise à jour majeure de PostgreSQL facilitée
- Quelques outils simplifient la vie
  - Outils alt-Perl
  - Outils slony1-ctl

Contrairement à d'autres systèmes de réplication, il est possible de choisir les tables à répliquer. Les esclaves sont disponibles en lecture seule. Il est malgré tout possible de créer d'autres objets, comme des tables de travail ou des index permettant à un outil comme BO de travailler

sur le serveur esclave.

Slony peut être utilisé sur des serveurs PostgreSQL de versions majeures différentes. Cela permet de faire une mise à jour de version avec une immobilisation moins importante qu'avec la méthode traditionnelle (qui revient à un export / import).

Le fait d'avoir à sélectionner les objets à répliquer est souvent vu comme un problème car cela demande une configuration longue et fastidieuse s'il y a beaucoup d'objets à répliquer. Des outils ont été créés pour remédier à cette impression-là. Ils offrent de nombreux avantages mais il est nécessaire de les maîtriser pour en tirer avantage.

5.4 Slony - Inconvénients

- Pas de réplication de la structure d'une base
- Pas de réplication des Large Objects
  - Par contre, pas de soucis avec les Bytea
- Pas de réplication du TRUNCATE (en 1.2 et 2.0)
- Complexe à maîtriser
- Une documentation qui, bien que complète, laisse à désirer

Comme Slony fonctionne avec des triggers, qu'il n'existe pas de triggers DDL et qu'il n'est pas possible de poser des triggers sur les catalogues systèmes, tout changement dans la structure de la base n'est pas répliquée. Par exemple, une nouvelle table ne sera pas automatiquement répliquée. Plus gênant, une nouvelle colonne pourra faire échouer la réplication car il ne sera plus possible de répliquer les modifications sur le table impactée.

Les Large Objects faisant partie d'un catalogue système, ils ne peuvent pas être répliqués. Les données binaires stockées avec le type Bytea ne sont pas concernées vu qu'elles sont enregistrées directement dans la table utilisateur.

Il n'existait pas de trigger sur l'instruction TRUNCATE jusqu'à la version 8.4. Pour l'instant, aucune version stable de Slony n'est capable de répliquer un TRUNCATE sur une table. Cependant, cela va changer avec la version 2.1 dont la version beta 1 est sortie le 6 mai.

Comme tout outil disposant de beaucoup de fonctionnalités, il est difficile à maîtriser. La documentation, bien que complète, est un document de référence, ce qui ne facilite pas la maîtrise de cet outil.

6 Réplication par journaux de transactions

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

Chaque transaction, implicite ou explicite, réalisant des modifications sur la structure ou les données d'une base est tracée dans les journaux de transactions. Ces derniers contiennent des informations d'assez bas-niveau, comme les blocs modifiés sur un fichier suite, par exemple, à un UPDATE. La requête elle-même n'apparaît jamais. Les journaux de transactions sont valables pour toutes les bases de données de l'instance.

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

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

6.1 PITR

- Point In Time Recovery
- Possibilité de rejouer
  - Tous les journaux de transactions
  - Jusqu'à un certain point dans le temps
  - Jusqu'à un certain identifiant de transaction
- Rejeu à partir d'une sauvegarde des fichiers à un instant *t*
- Disponible à partir de PostgreSQL 8.0 (sortie en 2005)

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

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

6.2 Warm Standby

- Esclave mis à jour en permanence
- Fichier par fichier
  - Un fichier == un journal de transactions
- Esclave non disponible pendant la restauration
- En 8.3 (sortie en 2008), ajout de l'outil pg_standby

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

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

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

6.3 Hot Standby - Introduction

- Patch développé par la société 2nd Quadrant
- Financement par un grand nombre de sociétés
- Deux ans de développement
- Initialement prévue pour la version 8.4
- Permet l'accès en lecture seule aux serveurs esclave

Le fait que les esclaves n'étaient pas disponibles en lecture seule a été un sujet de complainte de nombreux utilisateurs. Simon Riggs, par l'intermédiaire de sa société, a décidé de travailler sur ce problème. Plusieurs sociétés ont permis de financer cet effort qui a duré deux ans au total. Initialement prévu pour la 8.4, le patch est arrivé trop tard pour être intégré sur cette version. De plus, certaines questions restaient en suspens, ce qui jetait un gros doute sur la qualité du patch. Il a donc été décidé de repousser l'intégration de ce patch. Il a fallu un an supplémentaire pour finaliser le patch, d'où son intégration dans la version 9.0.

6.4 Hot Standby - Configuration

- Maître
  - Configuration d'archivage habituelle
  - postgresql.conf : wal_level = 'hot_standby'
  - Redémarrage du maître
- Esclave
  - Configuration normale d'un Warm Standby
  - postgresql.conf : hot_standby = 'on'
  - Redémarrage de l'esclave

Pour une explication complète sur la mise en place d'un serveur en HotStandby, nous vous conseillons la lecture de cet article :
http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_1

6.5 Streaming Replication - Introduction

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

Avoir la possibilité de se connecter en lecture seule aux esclaves est une fonctionnalité très intéressante. Malheureusement, on constate d'autant plus que les modifications ne sont intégrées sur l'esclave que fichier par fichier. Autrement dit, le lag de réplication est très visible car il peut être assez important.

NTT avait travaillé sur un système de réplication intégré à PostgreSQL. Leur système était synchrone et fonctionnait en flux. Ils ont parlé de leur système au PGCon 2006 (les slides sont disponibles sur http://www.pgcon.org/2008/schedule/events/76.en.html). Par la suite, ils ont travaillé avec les développeurs de PostgreSQL pour intégrer leur système (ou plus exactement une partie de ce système) dans le code source de PostgreSQL. Le résultat en est le patch Streaming Replication.

Ce patch introduit deux nouveaux processus. L'esclave se connecte au maître via un processus appelé walreceiver. Cette connexion déclenche le lancement d'un processus appelé walsender sur le maître.

La partie manquante du patch de NTT concerne le caractère synchrone de la réplication. Pour la version 9.0, la réplication en flux est une réplication asynchrone.

6.6 Streaming Replication - Configuration

- Maître
  - postgresql.conf : max_wal_senders = 1
  - pg_hba.conf : autoriser connexion esclave
  - Redémarrage du maître
- Esclave
  - recovery.conf : restore_command, archive_cleanup_command, trigger_file
  - recovery.conf : standby_mode (on), primary_conninfo
  - Redémarrage de l'esclave

Pour une explication complète sur la mise en place du Streaming Replication, nous vous conseillons la lecture de cet article :
http://www.dalibo.org/glmf131_mise_en_place_replication_postgresl_9.0_2

6.7 Administration

- Failover
  - Créer le fichier indiqué par trigger_file
- Switchover
  - Arrêter le maître, créer le fichier indiqué par trigger_file, reconstruire le maître en tant qu'esclave
- Supervision
  - pg_is_in_recovery(), pg_last_xlog_receive_location(), pg_last_xlog_replay_location()
  - Plugin munin, action check_postgres.pl

La mise en place de la réplication interne de PostgreSQL est très simple. Cela se fait en très peu de temps. Là où les choses se compliquent, c'est au niveau de l'administration et de la supervision.

Faire un failover est simple car il suffit de créer un fichier sur le serveur esclave pour qu'il

devienne maître. Il faut cependant bien faire attention à arrêter l'ancien maître, sinon des clients pourraient toujours s'y connecter et continuer à modifier les données sur l'ancien maître.

Faire un switchover est plus compliqué. Il faut arrêter l'ancien maître, passer l'esclave en nouveau maître, et reconstruire l'ancien maître en esclave. Cette dernière partie est généralement gênante car cela sous-entend de renvoyer les fichiers de l'esclave vers le maître. Il existe deux astuces pour éviter que cela ne prenne trop de temps : utiliser rsync pour diminuer le volume de données à renvoyer, ou utiliser repmgr (outil créé par 2ndQuadrant, disponible sur http://projects.2ndquadrant.com/repmgr). Si évidemment il y avait plusieurs esclaves, ces derniers sont aussi à reconstruire de la même façon que l'ancien maître.

Pour la supervision, les seules options possibles sont trois procédures stockées :

- pg_is_in_recovery() pour s'assurer que le PostgreSQL est bien en mode restauration,
- pg_last_xlog_receive_location() et pg_last_xlog_replay_location() : ces deux procédures stockées permettent de connaître le lag d'un esclave par rapport au maître (lag au niveau réception des données et au niveau rejeu).

Magnus Hagander a écrit un plugin munin pour visualiser le lag. Nicolas Thauvin a écrit une action permettant d'obtenir la même chose avec le plugin check_postgres.pl.

6.8 Avantages / Inconvénients

- Avantages
  - Facile à mettre en place, ne nécessite aucune modification des applications, ne désactive aucune fonctionnalité, dispose des esclaves en lecture seule avec un lag très léger
- Inconvénients
  - Administration complexe, supervision difficile, implication sur la sécurité mal appréhendée, pas de réplication synchrone, pas de réplication maître/maître, réplication de l'instance complète, pas de réplication en cascade

7 Et les prochaines versions ?

- 9.1
  - Réplication synchrone
  - Nouvel attribut REPLICATION pour les rôles
  - Nouvelle vue pg_stat_replication
  - Nouvelle vue pg_stat_database_conflicts
  - Module contrib pg_basebackup
- 9.X ?
  - Réplication avec un lag programmé

La version 9.1 améliore l'administration et la supervision des serveurs répliqués. L'attribut REPLICATION améliore la sécurité. Les nouvelles vues offrent plus de moyens de superviser son serveur. pg_stat_replication indique la liste des esclaves connectés au maître alors que pg_stat_database_conflicts rapporte le nombre de conflits constatés sur un esclave par type de conflit.

Quant au module pg_basebackup, il facilite grandement la mise en place d'un esclave. Il s'occupe directement du pg_start_backup(), du pg_stop_backup(), de la copie des fichiers et de leur transfert.

Il y a fort à parier que la 9.2 aura aussi son lot de nouveautés sur la réplication…

8 Conclusion

- PostgreSQL est un projet vivant
- Beaucoup de nouvelles fonctionnalités sur chaque version majeure
- Différentes solutions de réplication
- Et une réplication interne très intéressante et très simple

Pour plus d'informations sur la haute-disponibilité avec PostgreSQL, nous vous conseillons la lecture de cette page du wiki PostgreSQL : Replication, Clustering, and Connection Pooling