−Table des matières

Nouveautés de PostgreSQL 9.1, partie 1

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 145 (Janvier 2012). Il est disponible maintenant sous licence Creative Commons.

Sortie le 12 septembre, la version 9.1 constitue la version à utiliser en priorité pour les nouvelles installations de PostgreSQL. Elle comporte de nombreuses nouvelles fonctionnalités allant de la réplication à la sécurité, en passant par les fonctionnalités SQL et applicatives. Le tout en gagnant en performance.

Pour cette première partie, nous allons aborder les nouveautés sur la réplication, la sécurité et les performances.

Réplication

Pour la majorité des utilisateurs, les deux grosses nouveautés de la version 9.0 étaient la réplication en flux (en VO Streaming Replication) et les esclaves en lecture seule (Hot Standby). Cependant, la 9.0 étant la première version à avoir ces fonctionnalités, elles étaient simples, sans fioritures. La version 9.1 essaye donc de rattraper cela en proposant un ensemble d'améliorations aux fonctionnalités de réplication.

Attribut utilisateur pour la réplication

Le réplication se fait grâce à une connexion de l'esclave vers le maître. En 9.0, la chaîne de connexion vers le maître devait indiquer un rôle de type superutilisateur. Autrement dit, un rôle qui a tous les droits sur le serveur. Ceci n'est guère souhaitable pour des raisons de sécurité. La version 9.1 propose un nouvel attribut qui permet à un rôle standard (standard dans le sens « non superutilisateur ») de se connecter au serveur maître pour réaliser une connexion de réplication.

Cet attribut s'appelle réplication. Le nouveau rôle se crée ainsi :

```
CREATE ROLE repli LOGIN REPLICATION;
```

Cela porte donc à quatre le nombre d'attributs greffables à un rôle, les trois autres permettant de créer une base de données (CREATEDB), de créer un rôle (CREATEROLE) et d'avoir tous les droits (SUPERUSER).

L'option LOGIN indique que ce rôle a le droit de se connecter. Cependant, cela ne l'empêche pas d'avoir à respecter les règles explicitées dans le fichier pg_hba.conf.

Même si ce n'est pas indiqué sur cet exemple, il est toujours préférable de donner un mot de passe à ce rôle, même s'il n'est utilisé que pour la réplication.

Enfin, tout superutilisateur a par défaut les attributs REPLICATION, CREATEDB et CREATEROLE. Il est tout à fait possible de lui supprimer l'attribut REPLICATION, contrairement aux attributs CREATEDB et CREATEROLE.

Supervision de la réplication

L'un des soucis principaux de la version 9.0, en ce qui concerne la réplication, était la supervision. Il était possible de connaître le retard entre le maître et l'esclave en exécutant deux requêtes :

- sur le maître, pour connaître la position actuelle dans le journal de transactions courant
  ```
  SELECT pg_current_xlog_location
  ```
- sur l'esclave, pour connaître la dernière position reçue du journal de transactions
  ```
  SELECT pg_last_xlog_receive_location()
  ```
- sur l'esclave, pour connaître la dernière position rejouée du journal de transactions
  ```
  SELECT pg_last_xlog_replay_location()
  ```

Cela permettait de récupérer des informations indispensables mais encore fallait-il faire un petit calcul pour connaître le retard entre le maître et l'esclave, sans compter qu'avoir à ouvrir une connexion sur le maître et une sur l'esclave, simplement pour connaître le retard de cet esclave est assez gênant.

La version 9.1 améliore cela avec un catalogue système appelé pg_stat_replication. Elle n'est renseignée que sur le maître et permet de connaître l'état de tous les esclaves connectés. Voici un exemple du contenu de cette table :

```
postgres=# SELECT * FROM pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
procpid          | 5762
```

```
usesysid         | 10
usename          | guillaume
application_name | walreceiver
client_addr      | ::1
client_hostname  |
client_port      | 54781
backend_start    | 2011-11-05 10:51:16.072243+01
state            | streaming
sent_location    | 0/A000000
write_location   |
flush_location   |
replay_location  | 0/A000000
sync_priority    | 0
sync_state       | async
```

Le calcul est toujours à faire mais il n'est plus nécessaire de se connecter sur les deux serveurs.

La plupart des colonnes se comprennent aisément (ou sont déjà connus car elles font déjà partie du catalogue système pg_stat_activity). La colonne state indique l'état du flux de réplication : streaming indique que le flux est fonctionnel, catchup indique que le serveur esclave essaie de rattraper son retard, etc. sync_state peut avoir trois valeurs : async dans le cas d'une réplication asynchrone, sync et potential dans le cas d'une réplication synchrone.

Le retard d'un esclave sur son maître est toujours exprimé en octets, ce qui n'est pas simple à appréhender si on veut savoir si un esclave est trop en retard par rapport au maître. La version 9.1 propose une nouvelle procédure stockée, appelée pg_last_xact_replay_timestamp(), indiquant la date et l'heure de la dernière transaction rejouée. Du coup, soustraire la date et l'heure actuelle à cette fonction permet d'avoir une estimation sur le retard d'un esclave sous la forme d'une durée. Attention, cela ne fonctionne que si le maître est très actif. En effet, si aucun requête ne s'exécute sur le maître, le calcul now()-pg_last_xact_replay_timestamp() aura pour résultat une durée de plus en plus importante, même si l'esclave est identique au maître (un peu comme Slony et son pseudo lag quand les événements de synchronisation ne passent plus).

Supervision des conflits

Un autre catalogue système, pg_stat_database_conflicts, a été ajouté pour avoir des statistiques précises sur les conflits détectés sur un esclave. Pour rappel, il peut exister un conflit entre l'application des modifications de la réplication et la connexion sur une base d'un serveur esclave ou sur l'exécution d'une requête en lecture seule. Comme les modifications de la réplication doivent s'enregistrer dans l'ordre de leur émission, si une requête bloque l'application d'une modification, elle bloque en fait l'application de toutes les modifications suivantes pour cet esclave. Un exemple simple de conflit est l'exécution d'une requête sur une base que la réplication veut supprimer. PostgreSQL attend un peu avant de forcer l'application des modifications. S'il doit forcer, il sera contraint d'annuler les requêtes en cours, voire de déconnecter les utilisateurs. Évidemment, cela ne concerne que les requêtes et/ou les utilisateurs gênants.

Ce catalogue n'est renseigné que sur les esclaves d'une réplication. Il contient le nombre de conflits détectés sur cet esclave par type de conflit (conflit sur un tablespace, conflit sur un verrou, etc.). Le catalogue contient une ligne par base de données. Il est à noter que pg_stat_database contient une nouvelle colonne contenant le décompte total des conflits. Cela nous donne ce résultat :

```
postgres=# SELECT * FROM pg_stat_database WHERE datname='postgres' ;
-[ RECORD 1 ]-+------------------------------
datid         | 12857
datname       | postgres
numbackends   | 1
xact_commit   | 10370
xact_rollback | 32
blks_read     | 3090
blks_hit      | 387737
tup_returned  | 3236826
tup_fetched   | 110887
tup_inserted  | 12
tup_updated   | 1
tup_deleted   | 0
conflicts     | 5
stats_reset   | 2011-11-13 22:51:31.669638+01

postgres=# SELECT * FROM pg_stat_database_conflicts
postgres-# WHERE datname='postgres' ;
-[ RECORD 1 ]----+---------
datid            | 12857
datname          | postgres
confl_tablespace | 0
confl_lock       | 0
confl_snapshot   | 3
confl_bufferpin  | 2
confl_deadlock   | 0
```

Les informations disponibles dans ce catalogue permettent aux administrateurs de mieux configurer les paramètres vacuum_defer_cleanup_age, max_standby_archive_delay et max_standby_streaming_delay.

Concernant les conflits, il est aussi à savoir que les esclaves peuvent maintenant envoyer des informations au maître sur les requêtes en cours d'exécution pour tenter de prévenir les conflits de requêtes lors du nettoyage des enregistrements (action effectuée par le VACUUM). Il faut pour cela activer le paramètre hot_standby_feedback. L'esclave envoie des informations au maître à une certaine fréquence, configurée par le paramètre wal_receiver_status_interval. Il va sans dire que ces deux paramètres doivent être maniés avec précaution, notamment si les tables du serveur ont tendance à se fragmenter facilement. L'inconvénient est que cela peut causer une fragmentation des tables plus importantes sur le maître.

Grâce à cet envoi d'informations, PostgreSQL peut savoir si un esclave est indisponible, par exemple suite à une coupure réseau ou à un arrêt brutal de l'esclave. Si jamais l'esclave est indisponible pendant un certain temps (dépendant du nouveau paramètre replication_timeout), il coupe la connexion entre l'esclave et lui-même. Pour éviter des coupures intempestives, il faut donc configurer wal_receiver_status_interval avec une valeur inférieure à celle de replication_timeout.

Sauvegarde en flux

La création d'un esclave est une suite d'opérations simples. Cependant, comme il s'agit d'un ensemble d'opérations, il est aussi simple d'oublier une opération ou de se tromper sur une opération. Voici les étapes en question :

- exécution de la procédure stockée pg_start_backup() ;
- envoi sur l'esclave des fichiers du répertoire des données de PostgreSQL ainsi que des répertoires correspondant aux tablespaces ;
- exécution de la procédure stockée pg_stop_backup().

L'envoi se fait généralement en deux temps : création d'une archive tar, puis envoi (par scp, ftp ou autre) de cette archive sur l'esclave. Il faut par la suite se connecter sur l'esclave et y déballer l'archive pour pouvoir l'utiliser.

La version 9.1 propose un outil pour faire tout ça : pg_basebackup. Il s'utilise sur l'esclave. Il a besoin d'une connexion de réplication au serveur maître. Autrement dit, il n'est pas nécessaire d'ouvrir un port supplémentaire, le port de PostgreSQL (par défaut 5432) sera utilisé.

Voici comment utiliser pg_basebackup :

```
[guillaume@esclave glmf]$ pg_basebackup -D /home/guillaume/glmf/s2 -c fast -P -v
19016/19016 kB (100%), 1/1 tablespace
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
pg_basebackup: base backup completed
```

Le rôle utilisé dans cet exemple était guillaume. Il dispose de l'attribut REPLICATION pour avoir le droit d'établir une connexion de réplication avec le serveur maître, action indispensable pour utiliser correctement pg_basebackup.

Il ne reste plus qu'à créer le fichier recovery.conf, puis à démarrer PostgreSQL. Et l'esclave sera disponible.

Il est à noter que pg_basebackup peut être utilisé dans un autre cadre que la réplication. Il est utilisable pour faire des sauvegardes autonomes. Autrement dit, il s'agit d'une sauvegarde des fichiers de données, avec les journaux de transactions permettant un rejeu jusqu'au point de cohérence. Pour cela, il faut utiliser l'option -x. Il est aussi possible de récupérer un fichier tar avec l'option -Ft. Il est important de savoir que pg_basebackup() ne sauvegarde les journaux de transactions qu'à la fin de la copie des fichiers. Il est donc important que PostgreSQL n'ait pas recyclé les journaux de transactions nécessaires à cette sauvegarde autonome. Néanmoins, pour que cela fonctionne bien, pensez à configurer correctement le paramètre wal_keep_segments. Tout le travail du DBA se résume dans ce « correctement ». Il dépend de l'activité du maître, il n'y a donc pas de règles miraculeuses.

Contrôle de la réplication

Avec la version 9.0, il pouvait être très difficile de lancer un pg_dump sur un serveur esclave. En effet, l'exécution d'un pg_dump peut durer très longtemps et comme ce dernier travaille en exécutant des requêtes, certaines des requêtes se faisaient très facilement annuler (même après configuration des paramètres comme max_standby_streaming_delay). Il faudrait donc pouvoir mettre en pause l'application de la réplication. Un patch avait circulé pendant le développement de la version 9.0 et avait été réfusé à l'époque. Il a été proposé de nouveau pendant le cycle de développement de la version 9.1, et a été accepté. Il propose trois fonctions :

- pg_xlog_replay_pause(), pour mettre en pause la réplication sur l'esclave où est exécutée cette commande ;
- pg_xlog_replay_resume(), pour relancer la réplication sur un esclave où la réplication avait été précédemment mise en pause ;
- pg_is_xlog_replay_paused(), pour savoir si la réplication est en pause sur l'esclave où est exécutée cette commande.

Ces fonctions s'exécutent uniquement sur les esclaves et la réplication n'est en pause que sur l'esclave où la fonction est exécutée. Donc il est possible de laisser la réplication en exécution sur certains esclaves et de la mettre en pause sur d'autres.

Réplication synchrone

La réplication synchrone est très fréquemment demandée sur tous les moteurs de bases de données. En effet, lorsqu'une base est répliquée de façon asynchrone, cela signifie que, lorsqu'un utilisateur enregistre une donnée sur le maître, ce dernier indique à l'utilisateur que l'enregistrement s'est bien passé lorsqu'il a enregistré les données dans les journaux de transactions du maître. Autrement dit, il n'attend pas de savoir si l'esclave a reçu et encore moins enregistré les données sur disque. Le problème survient quand le maître meurt soudainement et qu'il faut basculer l'esclave en maître. Les dernières données enregistrées sur le maître n'ont peut-être pas eu le temps d'arriver sur l'esclave. Et du coup, on peut se trouver dans une situation où le serveur a indiqué la donnée comme enregistrée mais qu'après le failover, la donnée ne soit plus disponible. Utiliser une réplication synchrone évite ce problème en faisant en sorte que le maître ne renvoie la confirmation de la réussite de l'enregistrement à l'utilisateur qu'à partir du moment où l'esclave synchrone a lui-aussi enregistré la donnée.

Le gros avantage de cette solution est de s'assurer que si, par malheur, il fallait faire un failover, aucune donnée ne serait perdue. L'immense inconvénient de cette solution est que cela ajoute de la latence dans les échanges entre le client et le serveur maître pour chaque écriture. En effet, il ne faut pas seulement attendre que le maître fasse l'écriture, il faut aussi attendre l'écriture sur l'esclave sans parler des interactions entre le maître et l'esclave. Même si le coût est minime, le coût est présent. Et pour des serveurs faisant beaucoup d'écritures, le coût n'en sera que plus grand.

Ce sera donc du cas par cas. Pour certains, la réplication synchrone sera obligatoire (dû à un cahier des charges réclamant aucune perte de données en cas de failover). Pour d'autres, malgré l'intérêt de la réplication synchrone, la pénalité à payer sera trop importante pour se le permettre.

Avec PostgreSQL, passer d'une réplication asynchrone à une réplication synchrone est très simple : il suffit simplement de configurer la variable synchronous_standby_names. Ce paramètre doit contenir la liste des esclaves utilisant la réplication synchrone, en les séparant par des virgules. L'ordre des esclaves a un sens. Le premier esclave cité sera de priorité 1, le deuxième de priorité 2, etc.

Reste à savoir comment indiquer le nom d'un esclave. Le nom dépend d'un paramètre de connexion appelé application_name. Ce paramètre est apparu avec la version 9.0 et est utilisé ici pour distinguer les esclaves. La connexion de l'esclave au maître pour la réplication se configure avec le paramètre primary_conninfo dans le fichier recovery.conf.

Il est à noter que le statut asynchrone/synchrone peut se changer grâce à un paramètre nommé synchronous_commit. Pour les utilisateurs d'anciennes versions de PostgreSQL, ce paramètre était déjà utilisé pour permettre des insertions plus rapides en acceptant un délai (généralement très court, inférieur à trois fois la valeur du paramètre wal_writer_delay, soit 600ms par défaut) dans l'écriture et la synchronisation des journaux de transactions sur disque. Il conserve ce comportement dans le cadre de la réplication asynchrone (et évidemment sans réplication). Dans le cas de la réplication synchrone, ce paramètre contrôle aussi le fait que les écritures sont faites ou non sur l'esclave. Il peut donc avoir trois valeurs : on, off et local. Indiquer la valeur local permet de fonctionner, pour les prochaines requêtes de la session, en mode asynchrone. Cela peut se révéler utile quand on cherche à insérer des données éphémères.

Pratiquons un peu

Le plus simple étant de faire un exemple, allons-y.

Nous allons dans un premier temps mettre en place le serveur maître, sachant que son répertoire des données sera sur /home/guillaume/glmf/s1 et que son port de connexion sera le port par défaut (donc 5432). Voici les étapes à suivre :

```
$ export PGDATA=/home/guillaume/glmf/s1
$ initdb
```

Si tout se passe bien, vous devriez obtenir un message du style « Success. You can now start the database server…». Il nous faut maintenant configurer le serveur en tant que serveur maître. Pour cela, les modifications se font dans le fichier de configuration postgresql.conf. Voici la liste des modifications à réaliser :

- wal_level = 'hot_standby' (pour pouvoir exécuter des requêtes en lecture seule sur les esclaves) ;
- archive_mode = on (pour activer l'archivage des journaux de transactions) ;
- archive_command = 'cp %p /home/guillaume/glmf/archives/%f' (pour indiquer la commande d'archivage) ;
- max_wal_senders = 5 (pour accepter plusieurs esclaves sur ce serveur maître, le nombre indiqué est un peu trop haut par rapport à ce que nous voulons faire) ;
- hot_standby = on (inutile sur le maître mais ainsi nous n'aurons pas à modifier ce paramètre sur l'esclave) ;
- logging_collector = on (pour se faciliter la vie avec les traces) ;
- log_filename = 'postgresql-%Y-%m-%d.log' (là-aussi, pour se faciliter la vie avec les traces).

À noter que la commande d'archivage va placer les journaux archivés dans un répertoire /home/guillaume/glmf/archives. Ce répertoire doit être créé et l'utilisateur qui exécute PostgreSQL doit être autorisé à créer des fichiers à l'intérieur de ce répertoire :

```
$ mkdir /home/guillaume/glmf/archives
```

Il faut ensuite configurer les accès au serveur maître. Comme tous les serveurs PostgreSQL seront sur le même serveur physique, cela facilite énormément la configuration car il suffit d'ajouter cette ligne en fin du fichier pg_hba.conf :

```
local   replication   guillaume   trust
```

Comme dit plus haut, il faut que l'utilisateur guillaume dispose de l'attribut REPLICATION pour que cela fonctionne. Utiliser la méthode trust est une très mauvaise idée. Il est préférable de passer par la méthode md5 ou par une autre méthode sécurisée. Nous utilisons trust ici par simplification.

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

```
$ pg_ctl start
```

Si tout s'est bien passé, vous devriez voir ceci dans les traces :

```
LOG:  database system was shut down at 2011-11-05 11:40:16 CET
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
```

Le serveur maître étant prêt, nous pouvons passer à la construction du premier esclave. Utilisons pg_basebackup car il facilite grandement la vie :

```
$ pg_basebackup -D /home/guillaume/glmf/s2 -c fast -P -v
19016/19016 kB (100%), 1/1 tablespace
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
pg_basebackup: base backup completed
```

Il ne nous reste plus qu'à configurer l'esclave. Nous devons changer le port de connexion car nous sommes sur le même serveur physique. Donc, dans le fichier postgresql.conf, il faut simplement modifier le paramètre port en indiquant une autre valeur que 5432 (nous prendrons ici 5433).

Ensuite nous devons créer le fichier recovery.conf pour que PostgreSQL sache qu'il est un esclave en mode standby. Voici le contenu du fichier :

- restore_command = 'cp /home/guillaume/glmf/archives/%f %p' (pour qu'il utilise les journaux de transactions déjà archivés pour se mettre à jour) ;
- standby_mode = on (pour que, une fois qu'il n'y a plus de journaux à rejouer dans le répertoire d'archivage, il passe en streaming replication) ;
- primary_conninfo = 'port=5432 application_name=serveur2' (pour savoir où se trouve le serveur maître ; notez l'utilisation du paramètre application_name permettant de donner un nom à l'esclave).

Et c'est tout. Il ne reste plus qu'à démarrer le serveur esclave.

```
$ pg_ctl start
```

Les traces enregistrées dans les journaux applicatifs devraient indiquer ceci :

```
LOG:  database system was interrupted; last known up at 2011-11-05 11:44:33 CET
LOG:  creating missing WAL directory "pg_xlog/archive_status"
LOG:  entering standby mode
LOG:  restored log file "000000010000000000000003" from archive
LOG:  redo starts at 0/3000020
LOG:  consistent recovery state reached at 0/4000000
LOG:  database system is ready to accept read only connections
cp: cannot stat `/home/guillaume/glmf/archives/000000010000000000000004': No such file or directory
LOG:  streaming replication successfully connected to primary
```

L'esclave s'est connecté au serveur maître après avoir rejoué le seul journal nécessaire pour parvenir à un état cohérent. De plus, les connexions sont possibles en lecture seule.

Voyons cela sur le maître grâce à une des nouvelles vues système :

```
postgres=# select * from pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
procpid          | 6883
usesysid         | 10
usename          | guillaume
```

```
application_name | serveur2
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2011-11-05 11:50:04.595305+01
state            | streaming
sent_location    | 0/40000B0
write_location   | 0/40000B0
flush_location   | 0/40000B0
replay_location  | 0/40000B0
sync_priority    | 0
sync_state       | async
```

Nous avons donc un esclave, nommé serveur2 (information donnée par la colonne application_name), connecté en asynchrone (colonne sync_state), pour lequel la réplication est à jour (les colonnes *_location ont toutes la même valeur).

Passons maintenant ce serveur esclave en synchrone. Cette configuration se fait sur le maître avec le paramètre synchronous_standby_names du fichier postgresql.conf :

```
synchronous_standby_names = 'serveur2'
```

Après rechargement de la configuration (action reload de l'outil pg_ctl), les traces ajoutent ces deux lignes :

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "synchronous_standby_names" changed to "serveur2"
```

Le maître a donc bien pris en compte notre modification. Voyons comment cela se traduit dans la vue pg_stat_replication :

```
postgres=# select * from pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
procpid          | 6883
usesysid         | 10
usename          | guillaume
application_name | serveur2
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2011-11-05 11:50:04.595305+01
state            | streaming
sent_location    | 0/4000140
write_location   | 0/4000140
flush_location   | 0/4000140
replay_location  | 0/4000140
sync_priority    | 1
sync_state       | sync
```

La colonne sync_state est passé à sync. Le serveur serveur2 est bien en synchrone.

Créons une table sur le maître :

```
postgres=# CREATE TABLE t1();
CREATE TABLE
```

Aucun problème, la table se crée bien sur le maître comme sur l'esclave. Il est difficile de montrer que la réponse à l'utilisateur ne survient que quand les deux serveurs ont enregistré les modifications. Un moyen de le faire est d'arrêter l'esclave quand on a qu'un seul esclave :

```
$ pg_ctl -D /home/guillaume/glmf/s2 -mf stop
$ psql postgres
psql (9.1.1)
Type "help" for help.
postgres=# CREATE TABLE t2();
```

Et aucune réponse. La requête reste bloquée là en attente du serveur esclave. Ce dernier ne répondant pas, le maître est en attente. Supposons que nous redémarrons l'esclave à partir d'un autre terminal :

```
$ pg_ctl -D /home/guillaume/glmf/s2 start
```

Et tout d'un coup, la création de la table sur le maître se termine correctement. Il est évident qu'on ne cherche généralement pas à éteindre un esclave quand on veut de la réplication synchrone mais supposons que nous avons fait une mise à jour du noyau Linux et qu'il faille redémarrer pour avoir ce nouveau noyau, le maître reste bloqué le temps du redémarrage de l'esclave. Plus gênant, si vous avez fréquemment des coupures réseau entre le maître et l'esclave, les écritures vont en pâtir immédiatement à cause des blocages intermittents de la réplication

Maintenant, disons que le serveur serveur2 est arrêté pour un temps indéfini. Et un autre CREATE TABLE est lancé sur le maître. La requête est en attente de l'esclave. Que se passe-t-il si j'annule la requête ?

```
postgres=# CREATE TABLE t3();
^CCancel request sent
WARNING:  canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been replicated to the standby.
CREATE TABLE
```

Lors de l'annulation de la requête, les modifications sont déjà enregistrées sur le maître et il n'est pas possible de l'annuler. L'information n'a pas été envoyée à l'esclave mais ce dernier se rattrapera une fois qu'il aura été redémarré. Cela ne veut pas dire pour autant que le serveur esclave est sorti des serveurs synchrones. La prochaine requête en écriture bloquera de nouveau. Les deux seuls moyens pour sortir de cet état est de changer la configuration du paramètre synchronous_standby_names ou de redémarrer l'esclave.

Créons maintenant un deuxième esclave :

```
$ pg_basebackup -D /home/guillaume/glmf/s3 -c fast -P -v
19162/19162 kB (100%), 1/1 tablespace
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
```

```
pg_basebackup: base backup completed
```

La configuration est identique à celle du premier esclave. Le port doit être changé par le port 5434 dans le fichier postgresql.conf. Le fichier recovery.conf doit contenir les lignes suivantes :

```
restore_command = 'cp /home/guillaume/glmf/archives/%f %p'
standby_mode = on
primary_conninfo = 'port=5432 application_name=serveur3'
```

Et il ne reste plus qu'à démarrer le serveur esclave. Que nous indique le maître dans la vue pg_stat_replication ?

```
postgres=# SELECT application_name, sync_state
postgres-# FROM pg_stat_replication;
 application_name | sync_state
------------------+------------
 serveur3         | async
 serveur2         | sync
(2 rows)
```

Nous avons donc un serveur synchrone (serveur2) et un serveur asynchrone (serveur3). Plaçons maintenant le serveur 3 dans la liste des serveurs synchrones. Après rechargement de la configuration, PostgreSQL nous indique ceci dans les journaux applicatifs :

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "synchronous_standby_names" changed to "serveur2,serveur3"
```

La vue pg_stat_replication a changé en accordance :

```
postgres=# SELECT application_name, sync_state, sync_priority
postgres-# FROM pg_stat_replication;
 application_name | sync_state | sync_priority
------------------+------------+---------------
 serveur3         | potential  |             2
 serveur2         | sync       |             1
(2 rows)
```

serveur2 est le serveur synchrone. serveur3 le remplacera si serveur2 vient à ne plus être disponible. Donc, arrêtons serveur2 :

```
$ pg_ctl -mf stop
```

Le serveur maître s'en aperçoit comme l'indique les traces :

```
LOG:  standby "serveur3" is now the synchronous standby with priority 2
```

Du coup, la vue pg_stat_replication n'indique plus que le serveur serveur3 et ce dernier est en mode synchrone.

```
postgres=# SELECT application_name, sync_state, sync_priority
postgres-# FROM pg_stat_replication;
 application_name | sync_state | sync_priority
------------------+------------+---------------
 serveur3         | sync       |             2
(1 row)
```

Du coup, nous n'allons plus être bloqué en cas d'écritures :

```
postgres=# CREATE TABLE t4();
CREATE TABLE
```

Dès que serveur2 sera de nouveau disponible, il sera synchronisé, puis deviendra le serveur synchrone :

```
postgres=# SELECT application_name, sync_state, sync_priority
postgres-# FROM pg_stat_replication;
 application_name | sync_state | sync_priority
------------------+------------+---------------
 serveur3         | potential  |             2
 serveur2         | sync       |             1
(2 rows)
```

Évidemment, si les deux serveurs esclaves sont arrêtés, nous arrivons au même problème que précédemment, à savoir un blocage des écritures.

En mode synchrone, il est toujours possible d'utiliser les commandes de contrôle de la réplication. Autrement dit, il est possible de mettre en pause l'application de la réplication. Ce n'est pas du tout un problème car le mode synchrone garantit seulement que les données sont enregistrées sur l'esclave, pas qu'elles sont immédiatement visibles. C'est un mode suffisant pour s'assurer qu'un failover ne perd pas de données, c'est un mode insuffisant si on veut pouvoir faire de la répartition de charge pour laquelle les données visibles sur un serveur le sont aussi sur un autre. Pour ce dernier cas, il est possible, en étant rapide, de voir des différences entre les données d'un maître et d'un esclave.

Arrêtons maintenant les serveurs esclaves (donc serveur2 et serveur3). Le test précédent a montré qu'au cas où on se trouve avec un maître et un esclave lié par une réplication synchrone, la moindre modification sur le maître bloque en attente de la réponse du maître. C'est vrai à condition que synchronous_commit soit à on. En lui donnant la valeur local, la modification est autorisée sans attendre le retour de l'esclave :

```
postgres=# SET synchronous_commit TO local;
SET
postgres=# CREATE TABLE t10();
CREATE TABLE
```

C'est vrai aussi quand synchronous_commit est configuré à off mais dans ce cas, PostgreSQL n'attend même pas que le serveur maître ait enregistré et synchronisé ses journaux de transactions sur disque. Le gros intérêt de ce paramètre est de gagner en performance lorsque les données à enregistrer sont peu importantes. Redémarrons serveur2 et testons la vitesse d'écriture d'un million de lignes :

```
postgres=# CREATE TABLE t11 (c1 serial primary key, c2 text);
CREATE TABLE
Time: 708.430 ms
postgres=# INSERT INTO t11
```

```
postgres-# SELECT i,'Ligne '|| i FROM generate_series(1,1000000) AS i;
INSERT 0 1000000
Time: 12685.799 ms
postgres=# TRUNCATE t11;
TRUNCATE TABLE
Time: 140.474 ms
postgres=# SET synchronous_commit TO local;
SET
Time: 0.258 ms
postgres=# INSERT INTO t11
postgres-# SELECT i,'Ligne '|| i FROM generate_series(1,1000000) AS i;
INSERT 0 1000000
Time: 9643.279 ms
postgres=# TRUNCATE t11;
TRUNCATE TABLE
Time: 211.300 ms
postgres=# SET synchronous_commit TO off;
SET
Time: 0.236 ms
postgres=# INSERT INTO t11
postgres-# SELECT i,'Ligne '|| i FROM generate_series(1,1000000) AS i;
INSERT 0 1000000
Time: 8992.683 ms
```

La possibilité de modifier le statut synchrone des transactions permet d'amoindrir l'impact sur les performances dû à la réplication synchrone.

Sécurité

Authentification serveur sur les sockets unix

Le contrôle de l'accès au serveur se fait par vérification de l'utilisateur qui se connecte. L'utilisateur peut aussi vérifier qu'il se connecte bien au bon serveur en utilisant une connexion TCP/IP SSL avec la mise en place de certificats. Cependant, cette vérification n'était pas possible au niveau des connexions sur les sockets Unix.

La version 9.1 permet cela en ajoutant un nouveau paramètre de connexion : requirepeer. La valeur de cet argument correspond au nom de l'utilisateur qui lance PostgreSQL. Il s'agit habituellement de l'utilisateur postgres mais il est tout à fait possible que l'utilisateur sélectionné ait un autre nom. Il est aussi possible qu'un attaquant réussisse à récupérer les connexions PostgreSQL avec un autre serveur utilisant un autre utilisateur système. Là, cette nouvelle option protège l'utilisateur qui essaie de se connecter au serveur PostgreSQL :

```
[guillaume@laptop glmf]$ psql "dbname=postgres requirepeer=postgres"
psql: requirepeer specifies "postgres", but actual peer user name is "guillaume"
[guillaume@laptop glmf]$ psql "dbname=postgres requirepeer=guillaume"
psql (9.1.1)
Type "help" for help.

postgres=#
```

Ce paramètre, comme la méthode d'authentification peer, ne fonctionne que sur les systèmes Linux, la majorité des BSD (Mac OS X y compris) et Solaris.

Labels de sécurité

SELinux est un patch intégré à Linux ajoutant une surcouche très avancée de gestion des droits. Un développeur japonais a travaillé sur l'intégration de SELinux dans PostgreSQL. Ce travail a donné lieu tout d'abord à une version de dérivée de PostgreSQL, appelée SEPostgres. Après avoir montré l'intérêt de ces fonctionnalités, il a travaillé à coder des patchs intégrables dans la version en cours de développement de PostgreSQL (il s'agissait à l'époque de la 9.0). Ce travail a permis de nombreuses améliorations sur la gestion des droits dans PostgreSQL (c'est ainsi par exemple que la gestion des droits a été ajoutée sur les Large Objects en version 9.0), mais elle a surtout permis l'intégration d'une méthode simple et puissante de gestion des droits sur la connexion, l'accès aux objets, la modification des métadonnées d'un objet, l'utilisation de commandes utilitaires (comme la commande LOAD),

En fait, pour ne pas gérer qu'un seul type de gestion des droits, l'implémentation a été faite de façon à ce qu'un module de gestion de droits externes soit intégrable dans PostgreSQL. La version 9.1 en propose deux, disponibles sous la forme de modules à installer en plus de PostgreSQL. Les objets d'une base se voient affectés un label de sécurité grâce à une nouvelle commande (SECURITY LABEL). Quand PostgreSQL détecte qu'il a besoin d'accéder à tel ou tel objet avec tel label de sécurité, il demande au module si l'accès est autorisé. Si c'est le cas, l'exécution de la requête continue. Dans le cas contraire, la requête est annulée.

Le premier module, appelé dummy_seclabel, est principalement un exemple de module d'autorisation d'accès. Il propose quatre labels possibles : unclassified, classified, secret, top secret. Il autorise la mise en place de label sur les objets à condition que le label est d'un des quatre ci-dessus. Les deux derniers ne peuvent être placés que par un superutilisateur. Par contre, cela ne change pas les droits d'accès, ce qui fait de ce module un module d'exemple mais en aucun cas un module utilisable ainsi en production.

Le module sepgsql est un vrai module, utilisable en production. Il se base sur SELinux pour la gestion des autorisations d'accès aux objets. Un article complet ne suffirait pas à expliquer les tenants et aboutissants de ce module. Le plus intéressant dans ce système est qu'il est possible d'attacher un module externe de gestion des droits. Un module existe déjà pour SELinux mais il est tout à fait possible de coder le sien facilement.

Un petit exemple pour montrer la puissance de ce système. Disons que nous voulons empêcher toute connexion à PostgreSQL si un fichier particulier existe. Pour cela, nous avons besoin de créer un module de sécurité qui, à chaque connexion, va tester l'existence de ce fichier. Il faut donc s'attacher au code d'authentification. Au lancement de PostgreSQL, notre module doit définir la variable de type ClientAuthentication_hook avec le nom de notre fonction de sécurité. Dans notre fonction de sécurité, si l'authentification se passe bien, nous vérifions l'existence du fichier. Voici le code complet de ce module :

```
#include "postgres.h"
#include "commands/seclabel.h"
#include "libpq/auth.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
PG_MODULE_MAGIC;

static ClientAuthentication_hook_type next_client_auth_hook = NULL;

/*
 * my_client_auth
 * Point d'entrée du hook d'authentification.
 */
static void
my_client_auth(Port *port, int status)
{
    struct stat buf;

    if (next_client_auth_hook)
        (*next_client_auth_hook) (port, status);

    /*
     * Si l'authentification a échoué, pas la peine d'aller plus loin.
     */
    if (status != STATUS_OK)
        return;

    /*
     * Là se trouve tout le travail spécifique de notre module :)
     */
    if(!stat("/tmp/connection.stopped", &buf))
        ereport(FATAL, (errcode(ERRCODE_INTERNAL_ERROR),
            errmsg("Connection not authorized!!")));
}

/* Point d'entrée du module */
void
_PG_init(void)
{
    next_client_auth_hook = ClientAuthentication_hook;
    ClientAuthentication_hook = my_client_auth;
}
```

Pour le compiler, nous avons besoin du Makefile suivant :

```
MODULES = dummy_auth
ifdef USE_PGXS
PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
else
subdir = contrib/dummy_auth
top_builddir = ../..
include $(top_builddir)/src/Makefile.global
include $(top_srcdir)/contrib/contrib-global.mk
endif
```

Le fichier source doit avoir pour nom dummy_auth.c. La compilation et son installation se fait avec la commande suivante :

```
$ make USE_PGXS=1 install
```

Il est possible d'avoir à installer les paquets de développement de la bibliothèque libpq et du serveur PostgreSQL pour que la compilation se passe bien.

Ceci fait, il faut activer le module. Pour cela, il faut configurer la variable shared_preload_libraries avec le nom 'dummy_auth' puis redémarrer PostgreSQL. Ceci fait, le module est chargé et le test peut commencer :

```
[guillaume@laptop glmf]$ ll /tmp/connection.stopped
ls: cannot access /tmp/connection.stopped: No such file or directory
[guillaume@laptop glmf]$ psql postgres
psql (9.1.1)
Type "help" for help.

postgres=# \q
[guillaume@laptop glmf]$ touch /tmp/connection.stopped
[guillaume@laptop glmf]$ psql postgres
psql: FATAL:  Connection not authorized!!
```

Ajouter une gestion particulière des droits est faisable simplement. Par exemple, il est tout à fait plausible de créer un module qui utilise un annuaire LDAP pour savoir ce qu'un utilisateur a le droit d'utiliser ou non.

Performances

Supervision

Au niveau supervision, en dehors de la partie réplication, il n'y pas de nouvelles tables statistiques. Par contre, de nouvelles informations sont disponibles.

Sur pg_stat_all_tables, le nombre de VACUUM et d'ANALYZE est maintenant précisé, toujours avec la différentiation sur les opérations manuelles et automatiques (autrement dit, dues au démon autovacuum). Voici ce que cela donne :

```
postgres=# VACUUM t11;
VACUUM
postgres=# SELECT * FROM pg_stat_user_tables WHERE relname='t11';;
-[ RECORD 1 ]-----+------------------------------
relid             | 16422
```

```
schemaname        | public
relname           | t11
[... quelques autres colonnes ...]
last_vacuum       | 2011-11-20 14:08:50.236131+01
last_autovacuum   |
last_analyze      |
last_autoanalyze  | 2011-11-20 11:33:27.947532+01
vacuum_count      | 1
autovacuum_count  | 0
analyze_count     | 0
autoanalyze_count | 2
```

De même, la table pg_stat_bgwriter dispose de deux nouvelles informations :

```
postgres=# SELECT * FROM pg_stat_bgwriter;
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 65
checkpoints_req       | 16
buffers_checkpoint    | 25575
buffers_clean         | 5028
maxwritten_clean      | 46
buffers_backend       | 59204
buffers_backend_fsync | 0
buffers_alloc         | 41758
stats_reset           | 2011-11-05 11:42:55.472923+01
```

La colonne buffers_backend_fsync permet de savoir le nombre d'appels système fsync réalisés par les processus postgres. Plus ce nombre augmente, plus cela indique un problème au niveau de la configuration du bgwriter (le processus en charge des écritures des fichiers de données). Avoir cet indicateur est très important car tout travail de ce type réalisé par les processus postgres est en soi un gros soucis pour les performances du système.

Quant à la colonne stats_reset, elle indique la dernière réinitialisation de cette table statistique. Ce n'est pas la seule table statistique à disposer d'une telle colonne. pg_stat_database est aussi concernée.

Tables non journalisées

Au niveau de PostgreSQL, toute action réalisée est enregistrée deux fois : une première fois dans les journaux de transactions et une deuxième fois dans les fichiers de données. Seule exception à la règle : les index hash. Certaines optimisations permettent aussi de ne pas payer le prix de ce double enregistrement quand cela est possible. Ces optimisations sont désactivées dès que PostgreSQL a absolument besoin de l'information (par exemple dans le cas de la réplication). Ce comportement est important pour s'assurer de pouvoir avoir des performances et une grande fiabilité. Cependant, dans certains cas, il est intéressant de pouvoir enregistrer très rapidement des données, quitte à les perdre en cas de crashs. L'exemple le plus fréquent est une table de sessions web. L'utilisation de tables temporaires pourrait aider dans certains cas. En fait, dès que ces tables doivent pouvoir être utilisées par plusieurs connexions, les tables temporaires ne seront pas la solution (une table temporaire n'est visible que pour la session qui l'a créé).

La version 9.1 propose un nouveau type de table appelée UNLOGGED (pour « non journalisée »). Elle fonctionne de façon identique aux autres tables sauf que toute modification dans cette table n'est pas enregistrée dans les journaux de transactions. Du coup, en cas de crash du serveur, il n'est pas garanti que les données de cette table ne sont pas corrompues. La seule solution à ce problème est que, après redémarrage suite à un crash, les tables non journalisées sont vidées. Autre différence avec les tables standards, elles ne sont pas répliquées dans le cas de la réplication par journaux de transactions. Leur principal intérêt est leur rapidité :

```
postgres=# CREATE TABLE t12 (c1 serial primary key, c2 text);
[...]
CREATE TABLE
Time: 379.806 ms
postgres=# INSERT INTO t12
postgres-# SELECT i, 'Ligne '||i FROM generate_series(1,1000000) AS i;
INSERT 0 1000000
Time: 13438.877 ms
postgres=# CREATE UNLOGGED TABLE t13 (c1 serial primary key, c2 text);
[...]
CREATE TABLE
Time: 305.286 ms
postgres=# INSERT INTO t13
postgres-# SELECT i, 'Ligne '||i FROM generate_series(1,1000000) AS i;
INSERT 0 1000000
Time: 2907.262 ms
```

L'insertion prend 13 secondes dans le cas d'une table normale avec un index normal (pour la clé primaire). Elle ne prend que 3 secondes dans le cas d'une table non journalisée avec un index non journalisé. Cela se passe de commentaires.

Notez aussi qu'un index ajouté à une table non journalisée est forcément non journalisé lui-aussi.

Dernière information sur cette fonctionnalité, il n'est pas encore possible de passer une table du statut UNLOGGED au statut standard (et inversement).

KNN-GiST

KNN est l'acronyme de K-Next-Neighbour. Il s'agit d'un algorithme permettant de trouver les K plus proches voisins d'un point parmi un arbre. GiST est une méthode d'indexage de PostgreSQL.

KNN-GiST est donc la possibilité d'utiliser un index GiST pour rechercher les plus proches voisins de quelque chose. Ce « quelque chose » dépend de la recherche effectuée. Cela peut être une proximité géométrique, géographique ou autre. Pour l'instant, PostgreSQL propose ce type de recherche pour le type de données point. Des modules permettent son utilisation comme pg_trgm (recherche de mots similaires) et btree_gist. PostGIS 2.0 permettra à sa sortie son utilisation pour trouver les données géographiques proches d'un point particulier.

Prenons comme exemple le module pg_trgm. Ce module calcule un trigramme pour comparer les chaînes. Supposons une table mots contenant une seule colonne, la colonne mot de type text. Cette table fait 43 Mo pour 1100000 lignes. Ajoutons lui un index :

```
postgres=# CREATE INDEX mots_idx ON mots
postgres-# USING gist (mot gist_trgm_ops);
```

```
CREATE INDEX
```

Et maintenant, utilisant pg_trgm pour trouver les deux mots les plus proches du mot ganger en utilisant l'ancienne syntaxe :

```
postgres=# SELECT mot FROM mots
postgres-# ORDER BY similarity(mot, 'ganger') DESC LIMIT 2;
  mot
--------
 langer
 danger
(2 rows)

Time: 2485.960 ms
```

Maintenant, faisons la même recherche en utilisant le nouvel opérateur de similarité :

```
postgres =# SELECT mot FROM mots ORDER BY mot <-> 'ganger' LIMIT 2;
  mot
--------
 danger
 langer
(2 rows)

Time: 395.465 ms
```

La différence est parlante : 2,5 secondes avec l'ancienne méthode, 400 ms avec la nouvelle. Ceci est dû au fait que, contrairement à l'ancienne méthode, la nouvelle méthode peut utiliser l'index :

```
postgres=# EXPLAIN SELECT mot FROM mots
postgres-# ORDER BY similarity(mot, 'ganger') DESC LIMIT 2;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Limit  (cost=30538.50..30538.51 rows=2 width=9)
   ->  Sort  (cost=30538.50..33319.00 rows=1112200 width=9)
         Sort Key: (similarity(mot, 'ganger'::text))
         ->  Seq Scan on mots  (cost=0.00..19416.50 rows=1112200 width=9)
(4 rows)

Time: 0.648 ms
postgres=# EXPLAIN SELECT mot FROM mots
postgres-# ORDER BY mot <-> 'ganger' LIMIT 2;
                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.00..0.14 rows=2 width=9)
   ->  Index Scan using mots_idx on mots  (cost=0.00..79676.60 rows=1112200 width=9)
         Order By: (mot <-> 'ganger'::text)
(3 rows)

Time: 0.534 ms
```

Cependant, l'utilisation de cet index ne se limite pas à ça. Il est aussi utilisable dans le cas de la recherche du texte '%quelquechose%' (les précédentes versions de PostgreSQL ne pouvaient pas utiliser d'index dans ce cas) :

```
postgres=# SELECT count(*) FROM mots WHERE mot like '%gagn%';
 count
-------
   200
(1 row)

Time: 56.410 ms
postgres=# DROP index mots_idx ;
DROP INDEX
Time: 73.755 ms
postgres=# SELECT count(*) FROM mots WHERE mot like '%gagn%';
 count
-------
   200
(1 row)

Time: 183.030 ms
postgres=# CREATE INDEX mots_idx ON mots
postgres-# USING gist (mot gist_trgm_ops);
CREATE INDEX
postgres=# EXPLAIN SELECT count(*) FROM mots WHERE mot like '%gagn%';
                              QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=393.78..393.79 rows=1 width=0)
   ->  Bitmap Heap Scan on mots  (cost=5.49..393.50 rows=109 width=0)
         Recheck Cond: (mot ~~ '%gagn%'::text)
         ->  Bitmap Index Scan on mots_idx  (cost=0.00..5.46 rows=109 width=0)
               Index Cond: (mot ~~ '%gagn%'::text)
(5 rows)
```

Cette nouvelle fonctionnalité est donc bienvenue à plus d'un titre.

Conclusion

Voilà une partie des apports de cette nouvelle version. C'est déjà suffisamment alléchant pour vouloir tester cette nouvelle version. Mais il y a bien plus encore, comme nous le montrerons dans la deuxième partie.