Table des matières



# Nouveautés de PostgreSQL 9.1, partie 2

*Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 146 (Février 2012). Il est disponible maintenant sous licence Creative Commons.*


Dans la première partie, nous avons couvert les nouveautés en terme de réplication, de sécurité et de performances. Cependant, il reste plein de fonctionnalités à découvrir, ainsi qu'un changement important dont il faut tout particulièrement se méfier. Tout à la fin, nous aborderons ce qu'il faut attendre de la prochaine version de PostgreSQL.

## Fonctionnalités SQL et applicatives

### Support complet du collationnement

L'ordre de tri des données de type texte a toujours été complexe dans PostgreSQL.

Avant la 8.4, cet ordre était figé au moment de la création de l'instance (lors de l'initdb pour être précis). Il était impossible de spécifier un ordre différent par base de données. Du coup, en 8.4, un patch a permis d'avoir un ordre de tri différent par base de données. La 9.1 va bien plus loin en proposant un ordre de tri configurable par colonne. Il est donc possible d'avoir une table dont les différentes colonnes texte n'ont pas le même comportement pour le tri des données de type texte.

Le tri est toujours fait par la bibliothèque C pour la locale sélectionnée. PostgreSQL n'accepte donc, pour un collationnement particulier, que les locales actuellement installées sur le système. La liste des locales est disponible depuis PostgreSQL à partie de la vue système pg_collation:

```
b1=# SELECT * FROM pg_collation
b1-# WHERE collname IN ('de_DE.utf8', 'fr_FR.utf8', 'en_GB.utf8');
-[ RECORD 1 ]-+-----------
collname      | de_DE.utf8
collnamespace | 11
collowner     | 10
collencoding  | 6
collcollate   | de_DE.utf8
collctype     | de_DE.utf8
-[ RECORD 2 ]-+-----------
collname      | en_GB.utf8
collnamespace | 11
collowner     | 10
collencoding  | 6
collcollate   | en_GB.utf8
collctype     | en_GB.utf8
-[ RECORD 3 ]-+-----------
collname      | fr_FR.utf8
collnamespace | 11
collowner     | 10
collencoding  | 6
collcollate   | fr_FR.utf8
collctype     | fr_FR.utf8
```

L'ajout de locale dépend évidemment du système utilisé.

La locale peut être indiquée lors de l'ajout d'une colonne (il peut même être différent pour les colonnes d'une même table) :

```
b1=# CREATE TABLE t2 (c1 serial, c2 text COLLATE "fr_FR");
NOTICE:  CREATE TABLE will create implicit sequence "t2_c1_seq" for serial column "t2.c1"
CREATE TABLE
```

N'oubliez pas les doubles guillemets si le nom de la locale contient des majuscules (le nom de la locale est un nom d'objet). Maintenant, insérons quelques données:

```
b1=# INSERT INTO t2 (c2)
b1-# VALUES ('élève'),('élevé'),('élever'),('Élève');
INSERT 0 4
```

Essayons maintenant de lire les données en les triant sur la colonne c2:

```
b1=# SELECT * FROM t2 ORDER BY c2;
 c1 |   c2
----+--------
  1 | élève
  4 | Élève
  2 | élevé
  3 | élever
(4 rows)
```

L'ordre observé est l'ordre français. Mais supposons que nous avons un utilisateur allemand qui veut trier les données suivant l'ordre allemand. Il est possible de le spécifier en ajoutant la clause COLLATE à la clause ORDER BY, ainsi:

```
b1=# SELECT * FROM t2 ORDER BY c2 COLLATE "de_DE.utf8";
 c1 |   c2
----+--------
  2 | élevé
  1 | élève
  4 | Élève
  3 | élever
(4 rows)
```

Et en effet, l'ordre est différent.

Un tri peut facilement se faire en utilisant un index. Mais que se passe-t-il si nous voulons trier par un autre collationnement que celui par défaut ? Testons ça:

```
b1=# CREATE INDEX ON t2(c2);
CREATE INDEX
b1=# EXPLAIN ANALYZE SELECT * FROM t2 ORDER BY c2;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Sort  (cost=1.08..1.09 rows=4 width=36) (actual time=0.032..0.033 rows=4 loops=1)
   Sort Key: c2
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t2  (cost=0.00..1.04 rows=4 width=36) (actual time=0.006..0.009 rows=4 loops=1)
 Total runtime: 0.063 ms
(5 rows)
```

Par défaut, PostgreSQL n'utilisera pas l'index ici car la quantité de données est trop faible et, du coup, un parcours séquentiel suivi d'un tri sera forcément plus rapide que l'utilisation de l'index. Désavantageons fortement l'utilisation des parcours séquentiels en désactivant le paramètre enable_seqscan:

```
b1=# SET enable_seqscan TO off;
SET
b1=# EXPLAIN ANALYZE SELECT * FROM t2 ORDER BY c2;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Scan using t2_c2_idx on t2  (cost=0.00..12.31 rows=4 width=36) (actual time=0.069..0.073 rows=4 loops=1)
 Total runtime: 0.113 ms
(2 rows)
```

Cette fois, il passe bien par l'index. Changeons maintenant le collationnement.

```
b1=# EXPLAIN ANALYZE SELECT * FROM t2 ORDER BY c2 COLLATE "de_DE.utf8";
                                           QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=10000000001.08..10000000001.09 rows=4 width=36) (actual time=0.041..0.041 rows=4 loops=1)
```

```
   Sort Key: ((c2)::text)
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on t2  (cost=10000000000.00..10000000001.04 rows=4 width=36) (actual time=0.011..0.013 rows=4 loops=1)
 Total runtime: 0.078 ms
(5 rows)
```

Il ne passe plus par l'index. En effet, l'index est spécifique à la colonne et au collationnement choisi. Cependant, il est possible de créer un index pour un collationnement différent:

```
b1=# CREATE INDEX idx_allemand ON t2(c2 collate "de_DE.utf8");
CREATE INDEX
b1=# EXPLAIN ANALYZE SELECT * FROM t2 ORDER BY c2 COLLATE "de_DE.utf8";
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx_allemand on t2  (cost=0.00..12.31 rows=4 width=36) (actual time=2.758..2.765 rows=4 loops=1)
 Total runtime: 2.812 ms
(2 rows)
```

Du coup, le tri peut se faire avec ce nouvel index utilisant le bon collationnement.

## Requêtes CTE en écriture

PostgreSQL dispose des requêtes CTE depuis la version 8.4. Cependant, ces requêtes étaient seulement disponibles en lecture. Avec la 9.1, il est possible d'utiliser les commandes INSERT, UPDATE et DELETE.

Disons que nous avons un outil du type forum web, et que ce dernier utilise une base PostgreSQL pour y stocker les messages. Il y une table messages, contenant plusieurs colonnes dont une colonne id_utilisateur et une colonne cree_le. Supposons maintenant que nous voulons supprimer tous les messages datant de plus de six mois. La requête est simple :

```
DELETE FROM messages WHERE cree_le<now()-'6 months'::interval;
```

Parfait. Seulement, nous n'avons aucune information sur ce que nous avons réellement supprimé. Or, il serait intéressant d'avoir des statistiques sur la suppression. Par exemple, nous souhaitons savoir quels ont été les utilisateurs qui se sont vus supprimer des messages, ainsi que le nombre de messages qui a été supprimé par utilisateur. Cette information correspond à cette pseudo-requête:

```
SELECT id_utilisateur, count(*)
FROM “les éléments supprimées”
GROUP BY id_utilisateur;
```

Il est impossible de coller l'instruction DELETE derrière le FROM. De toute façon, l'instruction DELETE montrée ci-dessus ne renvoie pas les lignes supprimées. Cependant, il est possible de corriger ce dernier problème avec PostgreSQL en utilisant la clause RETURNING:

```
b1=# DELETE FROM messages
b1-# WHERE cree_le<now()-'6 months'::interval
b1-# RETURNING *;
 id_utilisateur |  cree_le
----------------+------------
              1 | 2011-02-01
              2 | 2011-03-01
              3 | 2011-04-01
              4 | 2011-05-01
              5 | 2011-06-01
             11 | 2011-01-01
              1 | 2011-01-01
              6 | 2011-03-01
             17 | 2011-04-01
[...]
```

Il ne nous reste plus qu'à placer le DELETE et le SELECT dans une requête CTE pour obtenir ce que l'on souhaite:

```
b1=# WITH messages_supprimes AS
b1-# (DELETE FROM messages
b1-#  WHERE cree_le<now()-'6 months'::interval
b1-#  RETURNING *)
b1-# SELECT id_utilisateur, count(*)
b1-# FROM messages_supprimes
b1-# GROUP BY id_utilisateur;
 id_utilisateur | count
----------------+-------
              8 |    23
             16 |     6
             15 |     6
              4 |    27
              1 |    39
             13 |    13
              5 |    36
             11 |    33
              3 |    28
             14 |    10
             17 |    10
              0 |    29
             12 |    16
             10 |    26
             18 |     2
              9 |    27
              6 |    29
              2 |    23
              7 |    31
(19 rows)
```

Et voilà !

En une requête, les messages sont supprimés et comptabilisés. Notez qu'il n'y a pas d'autres moyens de le faire réellement. Il est évidemment possible d'exécuter les deux requêtes séparément, dans deux transactions ou dans la même transaction. Mais même dans ce cas, il est possible que la statistique ne corresponde pas exactement au résultat de la suppression (un autre utilisateur peut avoir ajouté, modifié ou supprimé des messages entre temps).

Disons maintenant que nous voulons en plus mettre à jour une table de statistiques. Prenons donc une table stats contenant au moins deux colonnes, id_utilisateur et nombre_messages. Nous voulons mettre à jour nombre_messages en y retirant le nombre de messages supprimés. En utilisant la même technique que tout à l'heure, nous obtenons la requête suivante:

```
b1=# WITH messages_supprimes AS
b1-# (DELETE FROM messages
b1-#  WHERE cree_le<now()-'6 months'::interval
b1-#  RETURNING *) ,
b1-# par_utilisateur AS
b1-# (SELECT id_utilisateur, count(*)
b1-#  FROM messages_supprimes
b1-#  GROUP BY id_utilisateur)
b1-# UPDATE stats
b1-# SET nombre_messages = nombre_messages-par_utilisateur.count
b1-# FROM par_utilisateur
b1-# WHERE par_utilisateur.id_utilisateur=stats.id_utilisateur;
UPDATE 19
```

Les possibilités offertes par les requêtes CTE sont impressionnantes, sans même parler des performances qu'elles procurent.

## Extensions

PostgreSQL est un système très ouvert. Si un type de données n'existe pas, en créer un nouveau, répondant à votre contexte métier, est simple. Tout comme lui ajouter des fonctions, des opérateurs, etc. PostgreSQL est d'ailleurs livré depuis longtemps avec des extensions, appelées modules contrib. Ces modules couvrent de nombreux besoins qui dépendent vraiment de l'utilisation de la base. Par exemple, le module contrib adminpack fournit des fonctionnalités utiles aux outils d'administration (et est utilisé par pgAdmin) alors que le module contrib pgbench est un petit outil de benchmarks pour PostgreSQL et que le module contrib earthdistance sert à calculer la distance séparant deux points sur le globe. PostgreSQL fournit donc des modules contrib et il est aussi possible d'en télécharger ailleurs.

Il existe néanmoins des problèmes à l'utilisation des modules contrib. Certains demandent l'exécution d'un script SQL pour créer des objets dans la base où ils seront activés. Les objets en question dépendent souvent de l'installation préalable d'une bibliothèque partagée. Ils dépendent aussi de la version de cette bibliothèque et de la version de PostgreSQL. Lorsque la base contenant ces objets est sauvegardée, la sauvegarde contient les ordres de création de ces objets. Lors de la restauration, ils ne seront restaurés que si la bibliothèque partagée est déjà présente et d'une version compatible. En fait, les objets ajoutés par un module se comportent comme des objets utilisateurs alors qu'ils ont tout de l'objet système.

Apparaît donc en 9.1 le système des extensions. Ce système a pour but de faciliter la gestion des modules contrib, maintenant appelés extensions. PostgreSQL gère un nouvel objet (non standard) appelé extension. Après installation de l'extension sur le système d'exploitation, il ne reste plus à l'administrateur qu'à utiliser la commande CREATE EXTENSION. Il n'est plus nécessaire de savoir où se trouve le script SQL de l'extension, tout est géré par cette commande. Il suffit même d'une connexion à une base pour connaître la liste des extensions installées sur le système d'exploitation. Pour cela, il suffit d'interroger la vue système pg_available_extensions, par exemple avec la requête suivante:

```
postgres=# SELECT * FROM pg_available_extensions ORDER BY name;
-[ RECORD 1 ]-----+--------------------------------------------------------------------------
name              | adminpack
default_version   | 1.0
installed_version |
```

```
comment           | administrative functions for PostgreSQL
-[ RECORD 2 ]-----+-----------------------------------------------------------------------
name              | autoinc
default_version   | 1.0
installed_version |
comment           | functions for autoincrementing fields
-[ RECORD 3 ]-----+-----------------------------------------------------------------------
name              | btree_gin
default_version   | 1.0
installed_version |
comment           | support for indexing common datatypes in GIN
[... coupé car trop long ...]
```

La colonne name indique le nom de l'extension. La colonne default_version précise la version qui sera installée par défaut alors que installed_version indique la version déjà installée (elle vaut NULL si aucune version n'est installée). Enfin, la colonne comment est, comme son nom l'indique, un commentaire sur l'extension.

Adminpack n'étant pas activé, il suffit de la requête suivante pour le faire:

```
postgres=# CREATE EXTENSION adminpack;
CREATE EXTENSION
postgres=# SELECT * FROM pg_available_extensions WHERE name='adminpack';
-[ RECORD 1 ]-----+-----------------------------------------
name              | adminpack
default_version   | 1.0
installed_version | 1.0
comment           | administrative functions for PostgreSQL
```

À partir de là, les objets nécessaires à cette extension sont installés, comme le montre la commande \dx+ de psql:

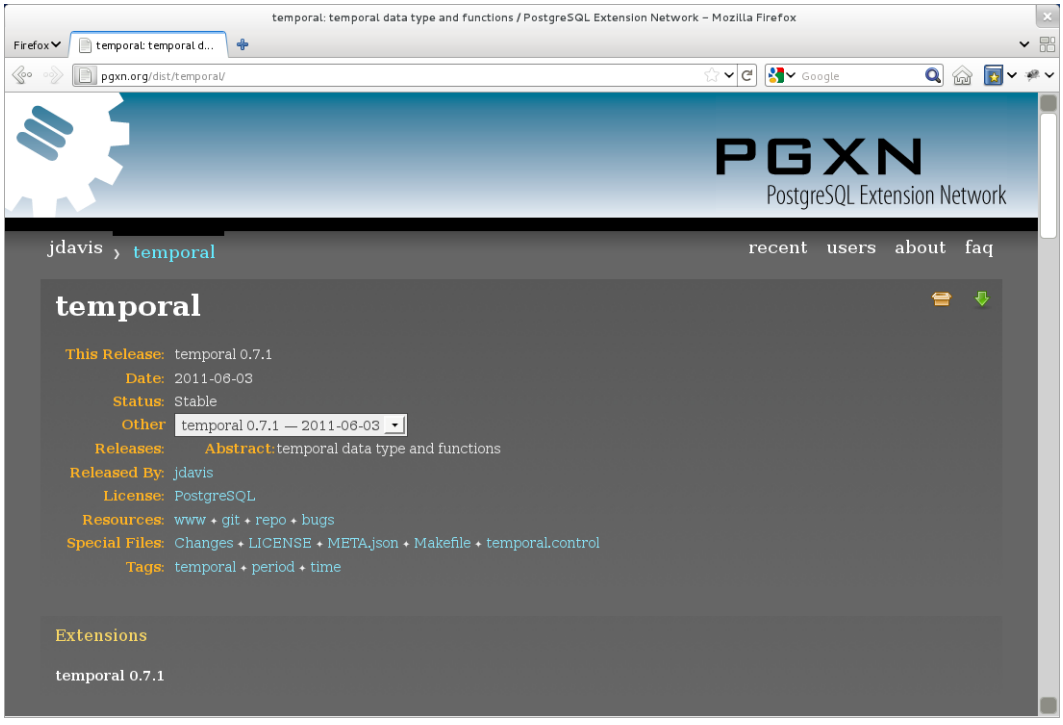
```
postgres=# \dx+ adminpack
  Objects in extension "adminpack"
        Object Description
-------------------------------------------
 function pg_file_length(text)
 function pg_file_read(text,bigint,bigint)
 function pg_file_rename(text,text)
 function pg_file_rename(text,text,text)
 function pg_file_unlink(text)
 function pg_file_write(text,text,boolean)
 function pg_logdir_ls()
 function pg_logfile_rotate()
(8 rows)
```

Ces objets sont bien créés dans la base mais sont liés à cette extension. Si jamais nous souhaitons supprimer cette extension, il nous suffit d'exécuter la commande DROP EXTENSION et nous n'avons pas à nous préoccuper de supprimer manuellement les différents objets ou à trouver un script de désinstallation. De même, nous pouvons mettre à jour une extension avec la commande ALTER EXTENSION.

De même, la sauvegarde, via pg_dump, ne contiendra que l'ordre CREATE EXTENSION pour chaque extension activée sur la base sauvegardée.

Bref, ce système permet de faciliter la vie des administrateurs en ce qui concerne les extensions. Il suffit de penser à PostGIS et à ses plus de 700 procédures stockées. La sauvegarde et la restauration était un problème permanent. La version 2.0 de PostGIS devrait permettre l'utilisation d'une extension avec PostgreSQL 9.1.

Ce système d'extension en a fait réfléchir plus d'un sur la possibilité de récupérer des extensions sur Internet. Il existe évidemment pgfoundry.org mais ce dernier est vieux, pratiquement non maintenu et beaucoup réclame même sa suppression. David Wheeler a travaillé un bon moment sur un équivalent CPAN pour PostgreSQL. Le résultat de son travail s'appelle PGXN pour PostGresql Extension Network. Le site web associé, http://pgxn.org, permet à tout un chacun de récupérer des extensions sur PostgreSQL et de les installer facilement.




Un outil, baptisé pgxnclient, permet de rechercher les extensions, de les télécharger et de les installer. Son installation est très simple vu qu'il suffit d'utiliser l'outil easy_install:

```
[guillaume@laptop ~]$ sudo easy_install pgxnclient
Searching for pgxnclient
Reading http://pypi.python.org/simple/pgxnclient/
Reading http://pgxnclient.projects.postgresql.org/
Reading https://github.com/dvarrazzo/pgxn-client/
Best match: pgxnclient 1.0.1
Downloading http://pypi.python.org/packages/source/p/pgxnclient/pgxnclient-1.0.1.tar.gz#md5=a27fecfa720ec7435465327a695798bf
Processing pgxnclient-1.0.1.tar.gz
Running pgxnclient-1.0.1/setup.py -q bdist_egg --dist-dir /tmp/easy_install-kxvRJM/pgxnclient-1.0.1/egg-dist-tmp-X2B_IH
Adding pgxnclient 1.0.1 to easy-install.pth file
Installing pgxn script to /usr/bin
Installing pgxnclient script to /usr/bin

Installed /usr/lib/python2.7/site-packages/pgxnclient-1.0.1-py2.7.egg
Processing dependencies for pgxnclient
Finished processing dependencies for pgxnclient
```

Disons que nous voulons installer l'extension contenant le type de données period. Commençons par chercher cette extension :

```
[guillaume@laptop ~]$ pgxn search period
temporal 0.7.1
    ... != *period*→ nequals(*period*, *period*) *period* -
[...]
omnipitr 0.2.1
    To make it happen, it will *periodically* run pg_controldata program,
[...]
pgmp 1.0.0
    The least significant bit will have a *period* no more than 2, and the
[...]
Slony-I 2.0.999
    After a *period* of time, Slony removes old confirmed events from both
```

```
[...]
```

Trouvé ! Il s'agit de l'extension temporal. Installons-là:

```
[guillaume@laptop ~]$ pgxn install temporal
INFO: best version: temporal 0.7.1
INFO: saving /tmp/tmpNwcgWs/temporal-0.7.1.zip
INFO: unpacking: /tmp/tmpNwcgWs/temporal-0.7.1.zip
INFO: building extension
cp temporal.sql.in sql/temporal--0.7.1.sql
sed 's,MODULE_PATHNAME,$libdir/temporal,g' temporal.sql.in >temporal.sql
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-labels -Wformat-security -fno-strict-aliasing -fwrapv -fpic -I. -I. -I/opt/postgresql-9.1/include/server -I/opt/postgresql-9.1/include/internal -D_GNU_SOURCE   -c -o s
src/temporal.c:53:16: warning: 'period_dup' defined but not used [-Wunused-function]
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-labels -Wformat-security -fno-strict-aliasing -fwrapv -fpic -L/opt/postgresql-9.1/lib -Wl,--as-needed -Wl,-rpath,'/opt/postgresql-9.1/lib',--enable-new-dtags  -shared -
rm src/temporal.o
INFO: installing extension
/bin/mkdir -p '/opt/postgresql-9.1/share/extension'
/bin/mkdir -p '/opt/postgresql-9.1/lib'
/bin/mkdir -p '/opt/postgresql-9.1/share/doc/extension'
/bin/sh /opt/postgresql-9.1/lib/pgxs/src/makefiles/../../config/install-sh -c -m 644 ./temporal.control '/opt/postgresql-9.1/share/extension/'
/bin/sh /opt/postgresql-9.1/lib/pgxs/src/makefiles/../../config/install-sh -c -m 644 ./sql/temporal--0.7.1.sql ./sql/temporal--0.7.0.sql ./sql/temporal--0.7.1.sql temporal.sql '/opt/postgresql-9.1/share/extension/'
/bin/sh /opt/postgresql-9.1/lib/pgxs/src/makefiles/../../config/install-sh -c -m 755  src/temporal.so '/opt/postgresql-9.1/lib/'
/bin/sh /opt/postgresql-9.1/lib/pgxs/src/makefiles/../../config/install-sh -c -m 644 ./doc/html/reference.html ./doc/html/index.html ./doc/html/tutorial.html ./doc/html/index.dist.html ./doc/html/abstract.html '/opt/postgresql-9.1/share/doc/extension/'
```

Téléchargé, décompressé, compilé, installé… prête à l'utilisation. Il est à noter que sur la majorité des distributions, il faudra avoir installé préalablement le paquet de développement de PostgreSQL et/ou de sa bibliothèque libpq, sans parler d'un environnement de développement.

Maintenant, il ne reste plus qu'à activer ce type dans une base:

```
[guillaume@laptop ~]$ pgxn load -d b1 temporal
INFO: best version: temporal 0.7.1
CREATE EXTENSION
```

Vérifions que l'extension est bien installée avec ses objets:

```
[guillaume@laptop ~]$ psql b1
psql (9.1.2)
Type "help" for help.

b1=# \dx+ temporal
                    Objects in extension "temporal"
                         Object Description
---------------------------------------------------------------------------
 function adjacent(period,period)
 function after(period,period)
 function before(period,period)
 function btree_period_compare(period,period)
 function contained_by(period,period)
[... et une soixantaine d'autres objets ...]
```

Créons une table avec le nouveau type:

```
b1=# CREATE TABLE t3 (id serial PRIMARY KEY, p period);
NOTICE:  CREATE TABLE will create implicit sequence "t3_id_seq" for serial column "t3.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t3_pkey" for table "t3"
CREATE TABLE
```

Tout va bien.

Le système des extensions et l'infrastructure PGXN forment un duo étonnant qui permettra d'aller encore plus loin que ce qu'on imagine pour l'instant. Un grand nombre d'extensions sont développées pour ajouter des fonctionnalités à PostgreSQL, fonctionnalités pour lesquelles il ne sera plus nécessaire d'attendre un an et d'avoir à installer une nouvelle version.

## SQL/MED

SQL/MED est partie de la norme ISO SQL portant sur les données externes à la base de données. MED est l'acronyme de Management of External Data, soit la gestion de données externes. La gestion de SQL/MED dans PostgreSQL a commencé dès la version 8.4 mais pour cette version, seule une partie de l'infrastructure avait été mise en place. Il manquait le plus important : la gestion des tables externes. C'est exactement ce que rajoute la version 9.1

PostgreSQL propose donc quatre objets pour SQL/MED:

- le Foreign Data Wrapper;
- le Foreign Server;
- le User Mapping;
- et le Foreign Table.

Les trois premiers étaient déjà disponibles depuis la version 8.4, le dernier n'est supporté que depuis la version 9.1.

Le Foreign Data Wrapper est en quelque sorte un connecteur permettant d'accéder à des données qui se trouvent à l'extérieur de la base de données où vous êtes connecté. Il existe déjà un certain nombre de Foreign Data Wrapper disponibles. Suivant les cas, ils pourront lire un fichier, lire des données dans une autre base de données (SQL comme noSQL), lire un flux twitter ou un autre webservice, etc. La seule limite est celle de votre imagination. Il existe une liste des différents Foreign Data Wrapper disponibles sur cette page : http://wiki.postgresql.org/wiki/Foreign_data_wrappers. À l'heure actuelle, il s'agit de MySQL, Oracle, ODBC, CouchDB, Redis, twitter, LDAP, Amazon S3 et d'un lecteur de fichiers CSV. Un grand nombre d'entre eux sont toujours en version beta, donc attention lors de leur utilisation. La version 9.1 ne propose pas de Foreign Data Wrapper pour PostgreSQL. Ce manque sera couvert normalement avec la version 9.2.

Le Foreign Server permet d'indiquer où se trouve la donnée à récupérer. Le User Mapping permet d'établir une correspondance entre l'utilisateur de connexion sur PostgreSQL et l'utilisateur du serveur distant (si un tel utilisateur est nécessaire). Enfin, le Foreign Table est une table un peu particulière.

Testons cette fonctionnalité avec le Foreign Data Wrapper file_fdw inclus dans les modules contrib de PostgreSQL. Ajoutons l'extension file_fdw dans la base de données b1:

```
b1=# CREATE EXTENSION file_fdw;
CREATE EXTENSION
```

L'ajout de l'extension a créé le Foreign Data Wrapper :

```
b1=# SELECT * FROM pg_foreign_data_wrapper ;
 fdwname  | fdwowner | fdwhandler | fdwvalidator | fdwacl | fdwoptions
----------+----------+------------+--------------+--------+------------
 file_fdw |       10 |      16569 |        16570 |        |
(1 row)
```

Nous pouvons maintenant ajouter le serveur externe:

```
b1=#  CREATE SERVER fichier FOREIGN DATA WRAPPER file_fdw ;
CREATE SERVER
```

Maintenant, ajoutons la table externe. Nous allons prendre pour exemple le fichier /etc/passwd car c'est un fichier que tout utilisateur Unix possède

```
b1=# CREATE FOREIGN TABLE passwd (pseudo text, password text,
b1-# uid integer, gid integer, nomcomplet text, repertoire text,
b1-# shell text)
b1-# SERVER fichier
b1-# OPTIONS (filename '/etc/passwd', format 'csv', delimiter ':');
CREATE FOREIGN TABLE
```

Comme pour toute création de table, il faut indiquer les colonnes (nom et type). Par contre, nous spécifions en plus le serveur, qui définit par conséquent le Foreign Data Wrapper. Nous ajoutons aussi les options spécifiques de la table, qui seront utilisées par le Foreign Data Wrapper : chemin vers le fichier (option filename), format du fichier et délimiteur. D'autres options sont disponibles mais cela suffira pour cet exemple.

Une fois la table créée, toute lecture de cette table fera une lecture sur le fichier /etc/passwd:

```
b1=# SELECT * FROM passwd LIMIT 5;
 pseudo | password | uid | gid | nomcomplet |   repertoire   |     shell
--------+----------+-----+-----+------------+----------------+---------------
 root   | x        |   0 |   0 | root       | /root          | /bin/bash
 bin    | x        |   1 |   1 | bin        | /bin           | /sbin/nologin
 daemon | x        |   2 |   2 | daemon     | /sbin          | /sbin/nologin
 adm    | x        |   3 |   4 | adm        | /var/adm       | /sbin/nologin
 lp     | x        |   4 |   7 | lp         | /var/spool/lpd | /sbin/nologin
(5 rows)
```

Toutes les opérations habituelles du SELECT (filtre, tri, utilisation d'expression, de sous-requête) sont permises. Toute mise à jour du fichier sera vue lors de la prochaine lecture de la table. Il faut bien comprendre que PostgreSQL ne stocke rien, il fait à chaque fois une lecture de ce fichier.

Cette fonctionnalité est indéniablement très intéressante mais souffre de quelques limitations. Par exemple, toute clause WHERE, LIMIT ou ORDER BY est exécutée sur le serveur PostgreSQL. Autrement dit, si vous demandez de lire une table de 5 Go en ajoutant un LIMIT 1, PostgreSQL récupère les 5 Go et fait le LIMIT après coup, même si le Foreign Data Wrapper concerne un moteur de bases de données qui serait capable de faire le LIMIT lui-même. Ce problème pourrait être résolu en version 9.2 mais ne l'est pas sur la 9.1. Il faut donc faire attention en utilisant cette fonctionnalité. Bien qu'elle soit très intéressante, elle peut avoir un comportement gênant dans certains cas. De plus, les écritures sont impossibles.

Notez qu'il est possible de coder son propre Foreign Data Wrapper, ce qu'a expliqué Dave Page dans la conférence qu'il a donné lors des conférences

européennes sur PostgreSQL de cette année.

## SSI

Le standard SQL définit quatre niveaux d'isolation des transactions:

- read uncommitted : niveau d'isolation le plus bas, une transaction peut lire les données modifiées par une autre transaction, qu'elle ait été ou non validée (COMMIT) ;
- read committed : une transaction ne peut lire les données modifiées par une autre transaction que si cette dernière a été validée ;
- repeatable read : une transaction ne verra aucune des modifications des autres transactions, validées ou non, à partir du moment où elle aura déjà lu les données (autrement dit, deux lectures de la même table donneront le même résultat pour une même transaction même si une autre transaction a modifié des données entre temps) ;
- serializable : niveau d'isolation le plus important, les transactions sont toutes exécutées comme si elles étaient exécutées les unes à la suite des autres.

Au niveau de PostgreSQL, seuls les niveaux read commited et repeatable read étaient implémentés. Un gros travail a été réalisé pendant le développement de la version 9.1 pour ajouter le niveau serializable. D'où l'ajout de SSI, acronyme pour Serializable Snapshot Isolation.

Prenons un exemple qui va expliquer le problème que nous cherchons à contourner.

Nous allons tout d'abord utiliser le mode read commited. Commençons avec la première session. Disons que nous voulons marquer comme payé toutes les factures du client d'identifiant 15. Commençons par vérifier les factures tout d'abord :

```
b1 (session 1) =# BEGIN;
BEGIN
b1 (session 1) =# SELECT * FROM factures WHERE id_client=15 and not payer;
 id_facture | id_client |  objet   | montant | payer
------------+-----------+----------+---------+-------
          3 |        15 | GLMF 106 |    6.50 | f
          4 |        15 | GLMF 107 |    6.50 | f
          5 |        15 | GLMF 108 |    6.50 | f
          7 |        15 | GLMF 110 |    6.50 | f
(4 rows)
```

Parfait, ça correspond à ce que nous attendions, nous allons donc mettre à jour ces quatre factures. Supposons maintenant qu'un deuxième utilisateur travaille en même et ajoute une nouvelle facture pour ce client:

```
b1 (session 2) =# begin;
BEGIN
b1 (session 2) =# insert into factures values (8, 15, 'HS44', 10, false);
INSERT 0 1
b1 (session 2) =# commit;
COMMIT
```

Tout s'est bien passé pour lui. Sur la session 1, rien ne nous indique qu'une nouvelle ligne a été ajoutée, nous sommes toujours dans notre transaction.

```
b1=# UPDATE factures SET payer=true WHERE id_client=15 and not payer;
UPDATE 5
```

Oups, nous avons modifié cinq factures, pas quatre. Un utilisateur qui le fait manuellement peut voir le problème, mais rien ne dit qu'un programme verra la différence.

Notez qu'ici l'astuce habituelle consistant à utiliser un SELECT … FOR UPDATE ne fonctionnera pas.

Essayons maintenant avec le niveau d'isolation serializable. Pour cela, nous allons modifier le paramètre default_transaction_isolation dans le fichier de configuration postgresql.conf:

```
default_transaction_isolation = 'serializable'
```

Cela étant fait, nous demandons à PostgreSQL de recharger sa configuration:

```
/etc/init.d/postgresql reload
```

Et maintenant, nous recommençons notre scénario

```
b1 (session 1) =# BEGIN;
BEGIN
b1 (session 1) =# SELECT * FROM factures WHERE id_client=15 and not payer;
 id_facture | id_client |  objet   | montant | payer
------------+-----------+----------+---------+-------
          3 |        15 | GLMF 106 |    6.50 | f
          4 |        15 | GLMF 107 |    6.50 | f
          5 |        15 | GLMF 108 |    6.50 | f
          7 |        15 | GLMF 110 |    6.50 | f
(4 rows)
```

OK, maintenant, l'action du deuxième utilisateur:

```
b1 (session 2) =# begin;
BEGIN
b1 (session 2) =# insert into factures values (8, 15, 'HS44', 10, false);
INSERT 0 1
b1 (session 2) =# commit;
COMMIT
```

Très bien et enfin la mise à jour des factures:

```
b1=# UPDATE factures SET payer=true WHERE id_client=15 and not payer;
UPDATE 4
b1=# commit;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

La session 2 ayant changé le contexte de la session 1 en ayant validé sa transaction avant celle de la session 1, la session 1 est forcée d'annuler sa transaction. L'application, constatant cette erreur, peut avertir l'utilisateur que le contexte du client 15 a changé et qu'il faut recommencer l'opération. L'utilisateur peut alors recommencer et finir son opération en modifiant exactement les factures qu'il souhaite.

PostgreSQL n'est pas le seul moteur qui implémente l'isolation sérialisable. Cependant, il le fait d'une façon très intelligente qui permet d'éviter le verrouillage généralement constaté sur les autres moteurs de bases de données. Par contre, il est à noter que ce mode d'isolation n'est pas utilisable sur les serveurs esclaves en mode Hot Standby.

## Support des triggers sur les vues

Avant la version 9.1, il n'était pas possible d'ajouter des triggers sur les vues. Si un utilisateur voulait avoir une action automatique suite à une action sur les vues, il devait passer par le système (complexe) des règles.

L'ajout d'un trigger sur une vue est maintenant possible avec la nouvelle clause INSTEAD OF en lieu et place des clauses BEFORE et AFTER. Ainsi, il est simple de créer des vues dont les données sont modifiables. Le plus gros avantage est cependant le fait qu'il ne soit plus nécessaire d'utiliser le système des règles.

## Dépendances fonctionnelles des clés primaires

Tout le monde a déjà eu à subir l'erreur suivante:

```
b1=# SELECT c1, c2, count(*) FROM t1 GROUP BY c1;
ERROR:  column "t1.c2" must appear in the GROUP BY clause or be used in an aggregate function
LINE 1: SELECT c1, c2, count(*) FROM t1 GROUP BY c1;
               ^
```

L'erreur est logique si la clé primaire n'est pas sur c1. Cependant, s'il se trouve que la table a été déclarée avec c1 comme clé primaire, alors le fait que c2 ne soit pas intégré dans la clause GROUP BY n'a aucune incidence sur le résultat. La version 9.1 accepte donc cette requête.

En dehors d'un comportement plus logique, cela apporte aussi une meilleure compatibilité entre les requêtes pour MySQL et celles pour PostgreSQL, étant donné que MySQL supportait déjà cette construction de requêtes.

# Changement important: standard_conforming_strings

Le plus gros problème pour passer à cette nouvelle version concerne l'activation par défaut du paramètre standard_conforming_strings. Ce changement fait que la chaîne 'l\'hôpital' n'est plus compris comme une chaîne complète, le standard SQL n'acceptant pas l'antislash comme caractère d'échappement. PostgreSQL l'acceptait quand même pour faciliter la vie des développeurs mais il a été décidé de revenir sur cette décision pour mieux respecter le standard. Ce paramètre a été ajouté en version 8.2, mais était resté désactivé par défaut. Le but était de permettre à ceux qui le voulaient de forcer le respect de la norme tout en laissant les autres utiliser leur anciennes applications. Seul un message d'avertissement indiquait la détection d'un problème. Au bout de quatre ans, il a été décidé d'activer par défaut ce paramètre. Pour les personnes peu prévoyantes, cela signifie que leur applications ne sont pas compatibles par défaut avec PostgreSQL. Néanmoins, il leur reste la possibilité de désactiver ce paramètre en le positionnant à off. Mais quelqu'un de non averti peut facilement tomber dans le piège, il est donc important d'être prévenu.

Néanmoins, espérons que cela aidera les développeurs à corriger leurs applications pour être plus proche du standard SQL. La chaîne 'l\'hôpital' n'est plus acceptée, mais la chaîne 'l''hôpital' l'est. Et pour être complet, la chaîne E'l\'hôpital' l'est aussi mais, attention, c'est spécifique à PostgreSQL. Tant qu'à corriger son application, autant le faire complètement et utiliser la syntaxe SQL.

## Quant à la prochaine version

Le développement de la version 9.2 a commencé plus tôt qu'habituellement, avec une première Commit Fest dès le 15 juin 2011. Et les nouvelles fonctionnalités ont commencé à arriver.

Le travail continue sur la réplication avec enfin la possibilité de faire de la réplication en cascade. De plus, un nouvel outil fait son apparition, pg_receivexlog. Ce dernier écoute le flux de réplication pour recréer les journaux de transactions sur un autre serveur. C'est une modification du protocole de réplication qui a permis le codage de ce nouvel outil.

Les utilisateurs devraient beaucoup apprécier l'ajout du type de données range. Il s'agit plus exactement d'un ensemble de types. Par exemple, int4range pour un intervalle d'entiers, mais il existe aussi le type numrange, tsrange (pour « timestamp range »), tstzrange, daterange, etc. Il est de toute façon toujours possible de définir son propre type range. Cette fonctionnalité ressemble énormément à l'extension temporal créée par Jeff Davis mais étendue à d'autres types de données comme les entiers et les numeric.

Une nouvelle méthode d'accès à un index a été ajouté. Nommée SP-GiST (pour Space Partitioned GiST), elle est comparable à GiST en flexibilité mais supporte aussi les structures de recherche partitionnées non balancées. Cette méthode peut battre la méthode GiST traditionnelle à la fois en vitesse de création des index et en rapidité de recherche.

Par ailleurs, il y a un énorme travail en cours sur les performances : support des parcours d'index seul, diminution du niveau des verrous pour les commandes ALTER TABLE, utilisation de latches, amélioration de la commande COPY permettant de gagner en performances grâce à une réduction du volume d'enregistrement dans les journaux de transactions, etc. L'ajout d'un nouveau processus appelé checkpointer devrait aussi participer à l'amélioration des performances. En fait, l'activité du processus d'écriture en tâche de fond (souvent appelé bgwriter) est divisé en deux parties : l'exécution des CHECKPOINT et le nettoyage du cache. Les développeurs ont fait en sorte que chaque tâche soit attribuée à un seul processus: à checkpointer l'exécution des CHECKPOINT et à bgwriter le nettoyage périodique du cache disque de PostgreSQL.

Les développeurs ont aussi ajouté l'export de snapshot (autrement dit l'image de la base qu'obtient une transaction). Ceci pourrait permettre une parallélisation de l'exécution de certaines requêtes ou de la sauvegarde. D'ailleurs, pour en savoir plus sur ce sujet, Heikki Linnakangas fera une conférence sur ce thème au FOSDEM 2012.

## Conclusion

Après 13 mois de travail, 4 commit-fests, 3 versions beta et une version RC, la communauté PostgreSQL a de nouveau fourni une version comportant de nombreuses améliorations, au niveau fonctionnalités comme au niveau performances. Est-ce que cette version vous intéressera ? Je pense que oui, mais le mieux est de tester vous-même et de vérifier qu'elle correspond bien à vos besoins.