DALIBO

L'Expertise PostgreSQL

Nouveautés de PostgreSQL 9.2

Table des matières

Nouveautés de PostgreSQL 9.2

1 À propos de l'auteur...

» Auteurs : Julien Rouhaud
» Société : DALIBO
» Date : Juillet 2012
» URL : https://kb.dalibo.com/conferences/postgresql_9.2/

1.1 Licence Creative Commons CC-BY-NC-SA

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

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

2 Introduction

- Développement commencé en mai 2011
- Déjà 16 mois de développement
- La phase RC1 a commencé fin août 2012
- Une sortie prévue pour début septembre

Le développement de la version 9.2 a commencé dès mai 2011, autrement dit trois mois avant la sortie officielle de PostgreSQL 9.1.

Le développement s'est déroulé de la façon standard : quatre commit fests, une période de stabilisation, quatre versions beta et enfin la version RC. La version finale est prévue à priori pour le 10 septembre.

3 Au menu

- Performances
- Réplication
- Administration
- Fonctionnalités utilisateurs
- Régressions potentielles

Ce document tente de présenter les principaux changements apportés par PostgreSQL 9.2, par rapport à la version majeure précédente, la version 9.1. Dans la mesure du possible, chaque

fonctionnalité sera expliquée et accompagnée d'une démonstration. Toutes les nouveautés ne sont bien sûr pas présentées car elles sont trop nombreuses.

La version 9.2 est une évolution importante. Elle est principalement axée sur les gains de performance avec de nombreuses améliorations, comme avec les parcours d'index seul et les temps d'acquisition des verrous.

Cette nouvelle version apporte tout d'abord la réplication en cascade, ce qui permet de mettre en place des architectures de haute disponibilité plus facilement. De plus, cela permet d'éviter des temps de reconstruction en cas de bascule du maître vers l'esclave, en se basant sur la robustesse de la réplication apportée lors des deux versions majeures précédentes.

Les changements sont donc divisés en cinq parties pour les besoins de cette présentation : les améliorations pour les performances, les nouveautés de la réplication, les nouveautés pour l'administration, les fonctionnalités pour les utilisateurs ainsi que les régressions potentielles.

4 Performances

- Parcours d'index seul
- Nouvelles options d'EXPLAIN
- Contraintes
- Divers

Avec chaque sortie d'une nouvelle version majeure de PostgreSQL, un gros travail est fait pour améliorer les performances. Cette nouvelle version ne déroge pas à la règle, et apporte de nombreuses améliorations.

4.1 Parcours d'index seul

- Avant la 9.2, pour une requête de ce type
  - SELECT c1 FROM t1 WHERE c1<10
- PostgreSQL devait lire l'index et la table
- Car les informations de visibilité ne se trouvent que dans la table
- En 9.2, le planificateur peut utiliser la « Visibility Map »
  - Nouveau nœud « Index Only Scan »

Voici un exemple en 9.1 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
```

```
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM (select generate_series(1,10000000)) AS t(a);
INSERT 0 10000000
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                                (actual time=209.569..3314.717 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                              (actual time=197.177..197.177 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 3323.497 ms
(5 rows)

b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                                 (actual time=48.620..269.907 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                               (actual time=35.780..35.780 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
 Total runtime: 273.761 ms
(5 rows)
```

Donc 3 secondes pour la première exécution (avec un cache pas forcément vide), et 273 millisecondes pour la deuxième exécution (et les suivantes, non affichées ici).

Voici ce que cet exemple donne en 9.2 :

```
b1=# CREATE TABLE demo_i_o_scan (a int, b text);
CREATE TABLE
b1=# INSERT INTO demo_i_o_scan
b1-# SELECT random()*10000000, a
b1-# FROM (select generate_series(1,10000000)) AS t(a);
INSERT 0 10000000
b1=# CREATE INDEX demo_idx ON demo_i_o_scan (a,b);
CREATE INDEX
b1=# VACUUM ANALYZE demo_i_o_scan ;
VACUUM
b1=# EXPLAIN ANALYZE SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan  (cost=0.00..3084.77 rows=86656 width=11)
                                          (actual time=0.080..97.942 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
 Total runtime: 108.134 ms
(4 rows)

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan  (cost=0.00..3084.77 rows=86656 width=11)
                                          (actual time=0.024..26.954 rows=89432 loops=1)
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=347
 Total runtime: 34.352 ms
(5 rows)
```

Donc, même à froid, il est déjà pratiquement trois fois plus rapide que la version 9.1, à chaud. La version 9.2 est dix fois plus rapide à chaud.

Essayons maintenant en désactivant les parcours d'index seul :

```
b1=# SET enable_indexonlyscan TO off;
SET
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                             (actual time=29.256..2992.289 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=346 read=43834 written=2022
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                               (actual time=27.004..27.004 rows=89432 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=346
 Total runtime: 3000.502 ms
(8 rows)

b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2239.88..59818.53 rows=86656 width=11)
                             (actual time=23.533..1141.754 rows=89432 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Rows Removed by Index Recheck: 6053582
   Buffers: shared hit=2 read=44178
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2218.21 rows=86656 width=0)
                               (actual time=21.592..21.592 rows=89432 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=2 read=344
 Total runtime: 1146.538 ms
(8 rows)
```

On retombe sur les performances de la version 9.1.

Maintenant, essayons avec un cache vide (niveau PostgreSQL et système) :

- en 9.1

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Bitmap Heap Scan on demo_i_o_scan  (cost=2299.83..59688.65 rows=89565 width=11)
                             (actual time=126.624..9750.245 rows=89877 loops=1)
   Recheck Cond: ((a >= 10000) AND (a <= 100000))
   Buffers: shared hit=2 read=44250
   ->  Bitmap Index Scan on demo_idx  (cost=0.00..2277.44 rows=89565 width=0)
                              (actual time=112.542..112.542 rows=89877 loops=1)
         Index Cond: ((a >= 10000) AND (a <= 100000))
         Buffers: shared hit=2 read=346
 Total runtime: 9765.670 ms
(7 rows)
```

- en 9.2

```
b1=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM demo_i_o_scan WHERE a BETWEEN 10000 AND 100000;
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Index Only Scan using demo_idx on demo_i_o_scan  (cost=0.00..3084.77 rows=86656 width=11)
                                       (actual time=11.592..63.379 rows=89432 loops=1)
```

```
   Index Cond: ((a >= 10000) AND (a <= 100000))
   Heap Fetches: 0
   Buffers: shared hit=2 read=345
 Total runtime: 70.188 ms
(5 rows)
```

La version 9.1 met 10 secondes à exécuter la requête, alors que la version 9.2 ne met que 70 millisecondes (elle est donc 142 fois plus rapide).

Voir aussi http://pgsnaga.blogspot.com/2011/10/index-only-scans-and-heap-block-reads.html

4.2 Commande EXPLAIN

- Nouvelle clause `TIMING` pour `EXPLAIN`
- Clause BUFFERS : affiche maintenant les blocs modifiés
- Affichage du nombre de lignes rejetées par un filtre

`EXPLAIN ANALYZE` est le meilleur outil pour optimiser des requêtes. Malheureusement, sur certains systèmes, le surcoût induit par le chronométrage des nœuds augmente de façon considérable le temps d'exécution d'`EXPLAIN ANALYZE`. En désactivant le chronométrage, on peut donc réduire le temps d'exécution, tout en gardant les nombres réels de lignes par nœud.

Voilà un exemple de l'affichage des lignes rejetées par un filtre :

```
demo_9_2=# CREATE TABLE t1 (c1 integer, c2 varchar);
CREATE TABLE
demo_9_2=# INSERT INTO t1 SELECT i,'ligne ' || i FROM generate_series(1,10000) i;
INSERT 0 10000
demo_9_2=# VACUUM ANALYZE t1;
VACUUM
demo_9_2=# EXPLAIN ANALYZE SELECT * FROM t1 WHERE c1 < 500;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..180.00 rows=500 width=14) (actual time=0.005..0.671 rows=499 loops=1)
   Filter: (c1 < 500)
   Rows Removed by Filter: 9501
 Total runtime: 0.698 ms
(4 lignes)
```

La même requête en désactivant le chronométrage des nœuds :

```
demo_9_2=# EXPLAIN (ANALYZE on, TIMING off) SELECT * FROM t1 WHERE c1 < 500;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..180.00 rows=500 width=14) (actual rows=499 loops=1)
   Filter: (c1 < 500)
   Rows Removed by Filter: 9501
 Total runtime: 0.685 ms
(4 lignes)
```

Pour les nouveaux affichages générés par l'option BUFFERS, il faut noter le nombre de blocs disques modifiés (`dirtied`) :

```
demo_9_2=# EXPLAIN (ANALYZE , BUFFERS) UPDATE t1 SET c2 = 'nouveau ' || c1 WHERE c1 < 500;
                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------
------
 Update on t1  (cost=0.00..183.75 rows=500 width=10) (actual time=1.343..1.343 rows=0 loops=1)
   Buffers: shared hit=1065 read=2 dirtied=2
   ->  Seq Scan on t1  (cost=0.00..183.75 rows=500 width=10)
                       (actual time=0.008..0.796 rows=499 loops=1)
         Filter: (c1 < 500)
         Rows Removed by Filter: 9501
         Buffers: shared hit=55
 Total runtime: 1.359 ms
(7 lignes)
```

Enfin, dernière nouveauté, EXPLAIN affiche aussi le nombre de lignes ignorées par le filtre (`Rows Removed by Filter`).

4.3 Contraintes

- Possibilité de ne pas revérifier une clé étrangère dans certains cas
- Possibilité de créer des contraintes CHECK invalides

Lors du changement de type d'une colonne, si les données n'ont pas besoin d'être réécrites (varchar(10) → varchar(20), passage d'un domaine en son type de base …), la vérification des contraintes de clé étrangère sur ces colonnes n'est plus effectuée.

L'intérêt de créer des contraintes CHECK invalides est de pouvoir les créer immédiatement sans gêner les utilisateurs. Elles ne sont pas respectées jusqu'à leur validation. Cette validation peut se faire plus tard, dans la nuit, à un moment de moindre activité. En voilà un exemple :

```
demo_9_2=# CREATE TABLE t1 (c integer, num integer);
CREATE TABLE
demo_9_2=# INSERT INTO t1 SELECT i,(random()*10)::int FROM generate_series(1,10000) i;
INSERT 0 10000
demo_9_2=# ALTER TABLE t1 ADD CONSTRAINT t2_chk_num CHECK (num <> 0) NOT VALID;
ALTER TABLE
demo_9_2=# SELECT count(*) FROM t1 WHERE num=0;
 count
-------
   459
(1 row)

demo_9_2=# INSERT INTO t1 VALUES (10001, 0);
ERROR:  new row for relation "t1" violates check constraint "t2_chk_num"
DETAIL:  Failing row contains (10001, 0).
demo_9_2=# ALTER TABLE t1 VALIDATE CONSTRAINT t2_chk_num;
ERROR:  check constraint "t2_chk_num" is violated by some row
demo_9_2=# DELETE FROM t1 WHERE num = 0;
DELETE 467
demo_9_2=# ALTER TABLE t1 VALIDATE CONSTRAINT t2_chk_num;
ALTER TABLE
```

4.4 Tri

- Plus grande rapidité des tris en mémoire
- Nouveau paramètre `temp_file_limit`
  - limite sur la taille des fichiers temporaires

Non seulement les tris sont plus rapides en mémoire mais il est aussi possible de borner la taille des fichiers créés sur disque dans le cas où le tri ne tient pas en mémoire :

```
postgres=# EXPLAIN ANALYZE SELECT * FROM t1 ORDER BY c1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=150112.84..152612.84 rows=1000000 width=16)
       (actual time=560.419..673.443 rows=1000000 loops=1)
   Sort Key: c1
   Sort Method: external sort  Disk: 26296kB
   ->  Seq Scan on t1  (cost=0.00..16274.00 rows=1000000 width=16)
                       (actual time=0.033..97.147 rows=1000000 loops=1)
 Total runtime: 714.746 ms
(5 rows)

postgres=# SET temp_file_limit TO '20MB';
SET
postgres=# EXPLAIN ANALYZE SELECT * FROM t1 ORDER BY c1;
ERROR:  temporary file size exceeds temp_file_limit (20480kB)
```

4.5 Processus

- Nouveau processus `checkpointer`
- Amélioration du `walwriter` pour de meilleures performances du commit asynchrone
- Remplacement de boucles d'attente par des « latches »

Jusqu'en version 9.1, le processus d'écriture en tâche de fond (appelé `bgwriter`) s'occupait de gérer toutes les écritures en tâche de fond sur les fichiers de données. Il les gérait de deux façons :

- un nettoyage peu intensif mais très fréquent (par défaut 100 blocs disques toutes les 200 millisecondes) ;
- un nettoyage de fond, appelé CHECKPOINT, beaucoup moins fréquent (par défaut toutes les cinq minutes), mais exhaustif.

Cette gestion double commence à poser des problèmes, en terme de performance et de configuration. Pour essayer de les contourner, il a été décidé qu'il y aurait maintenant deux processus d'écriture en tâche de fond :

- le premier (qui garde le nom de `writer process`) s'occupe du nettoyage partiel très

fréquent ;

- le deuxième qui s'appelle `checkpointer` ne s'occupe que des `CHECKPOINT`.

Voici l'arborescence des processus d'un serveur PostgreSQL en 9.1:

```
~$ ps fu -u postgres
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
postgres 21109  1.3  0.1 132152 10376 ?        S    10:59   0:00
/usr/lib/postgresql/9.1/bin/postgres -D /var/lib/postgresql/9.1/main -c
config_file=/etc/postgresql/9.1/main/postgresql.conf
postgres 21110  0.0  0.0 100508  1488 ?        Ss   10:59   0:00  \_ postgres: logger process
postgres 21113  0.0  0.0 132152  1876 ?        Ss   10:59   0:00  \_ postgres: writer process
postgres 21114  0.0  0.0 132152  1636 ?        Ss   10:59   0:00  \_ postgres: wal writer process
postgres 21115  0.0  0.0 133516  3336 ?        Ss   10:59   0:00  \_ postgres: autovacuum launcher
process
postgres 21116  0.0  0.0 101240  2124 ?        Ss   10:59   0:00  \_ postgres: stats collector
process
```

Et voila celles des processus de la 9.2 :

```
~$ ps fu -u postgres
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
postgres 21151  0.0  0.1 132080  9956 ?        S    10:59   0:00
/usr/lib/postgresql/9.2/bin/postgres -D /var/lib/postgresql/9.2/main -c
config_file=/etc/postgresql/9.2/main/postgresql.conf
postgres 21152  0.0  0.0 100452  1344 ?        Ss   10:59   0:00  \_ postgres: logger process
postgres 21154  0.0  0.0 132080  1500 ?        Ss   10:59   0:00  \_ postgres: writer process
postgres 21155  0.0  0.0 132080  1496 ?        Ss   10:59   0:00  \_ postgres: checkpointer process
postgres 21156  0.0  0.0 132080  1496 ?        Ss   10:59   0:00  \_ postgres: wal writer process
postgres 21157  0.0  0.0 132928  2824 ?        Ss   10:59   0:00  \_ postgres: autovacuum launcher
process
postgres 21158  0.0  0.0 100584  1640 ?        Ss   10:59   0:00  \_ postgres: stats collector
process
```

Toutes les boucles d'attente ont été remplacées par des `latches` (loquets en français). Ces nouveaux évènements permettent des périodes d'inactivité plus efficaces en réduisant le nombre de réveils des processeurs, ce qui réduit énormément la consommation énergétique sur les serveurs inactifs.

4.6 Divers

- Meilleure scalabilité (jusqu'à 64 cœurs)
- Optimisation pour le calcul des agrégats `bool_and` et `bool_or` avec le type booléen
- Réduction du volume d'enregistrement dans les journaux de transactions avec COPY

L'optimisation des agrégats booléens utilise la même technique que pour `min()` et `max()` avec des données de type integer par exemple. Voici ce que ça donne en 9.2 :

```
postgres=# CREATE TABLE t2 (c1 integer, c2 boolean);
CREATE TABLE
postgres=# INSERT INTO t2 SELECT i, i%3=1 FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000
postgres=# CREATE INDEX ON t2(c2);
CREATE INDEX
postgres=# EXPLAIN ANALYZE SELECT bool_and(c2) FROM t2 ;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Result  (cost=0.04..0.05 rows=1 width=0) (actual time=0.127..0.127 rows=1 loops=1)
   InitPlan 1 (returns )
     ->  Limit  (cost=0.00..0.04 rows=1 width=1) (actual time=0.120..0.120 rows=1 loops=1)
           ->  Index Only Scan using t2_c2_idx on t2  (cost=0.00..42069.57 rows=1000000 width=1)
                                                      (actual time=0.116..0.116 rows=1 loops=1)
                 Index Cond: (c2 IS NOT NULL)
                 Heap Fetches: 1
 Total runtime: 0.213 ms
(7 rows)

postgres=# EXPLAIN ANALYZE SELECT bool_or(c2) FROM t2 ;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Result  (cost=0.04..0.05 rows=1 width=0) (actual time=0.078..0.078 rows=1 loops=1)
   InitPlan 1 (returns )
     ->  Limit  (cost=0.00..0.04 rows=1 width=1) (actual time=0.074..0.074 rows=1 loops=1)
           ->  Index Only Scan Backward using t2_c2_idx on t2
               (cost=0.00..42069.57 rows=1000000 width=1)
               (actual time=0.073..0.073 rows=1 loops=1)
                 Index Cond: (c2 IS NOT NULL)
                 Heap Fetches: 1
 Total runtime: 0.133 ms
(7 rows)
```

Et voici ce qu'on avait en 9.1 :

```
postgres=# CREATE TABLE t2 (c1 integer, c2 boolean);
CREATE TABLE
postgres=# INSERT INTO t2 SELECT i, i%3=1 FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000
postgres=# CREATE INDEX ON t2(c2);
CREATE INDEX
postgres=# EXPLAIN ANALYZE SELECT bool_and(c2) FROM t2 ;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
-
 Aggregate  (cost=16925.00..16925.01 rows=1 width=1) (actual time=183.136..183.136 rows=1 loops=1)
   ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=1)
                       (actual time=0.028..92.067 rows=1000000 loops=1)
 Total runtime: 183.198 ms
(3 rows)

postgres=# EXPLAIN ANALYZE SELECT bool_or(c2) FROM t2 ;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
-
 Aggregate  (cost=16925.00..16925.01 rows=1 width=1) (actual time=185.971..185.971 rows=1 loops=1)
   ->  Seq Scan on t2  (cost=0.00..14425.00 rows=1000000 width=1)
                       (actual time=0.049..93.582 rows=1000000 loops=1)
 Total runtime: 186.011 ms
(3 rows)
```

L'amélioration de COPY est surtout visible lors de son utilisation par plusieurs processus. Avec cinq processus, on est six fois plus rapides. Les performances descendent après 15 processus en parallèle, mais sont toujours six fois plus rapides qu'avant (y compris à 35 processus en parallèle). Cela permet une très bonne montée en charge des insertions par COPY.

En ce qui concerne les SELECT, l'amélioration permet d'être 3,5 fois plus rapide grâce à un travail sur le gestionnaire des verrous.

Des tests ont montré une montée en charge régulière jusqu'à 64 cœurs grâce à ces améliorations.

Pour plus de détails, lire http://rhaas.blogspot.fr/2012/04/did-i-say-32-cores-how-about-64.html

5 Réplication

- Réplication en cascade
- Création d'un esclave à partir d'un autre esclave
- Réplication en mémoire

Les deux dernières versions majeures de PostgreSQL ont apporté une réplication native, d'abord asynchrone puis synchrone. Cette nouvelle version majeure permet de créer plus facilement des architectures de haute-disponibilité, et de minimiser les temps de reconstruction et l'impact sur les performances en cas d'incident.

5.1 Réplication en cascade

- Pouvoir diminuer la charge du travail de réplication en la déportant sur un esclave
- Était possible avec d'autres solutions de réplication
  - Enfin possible nativement avec PostgreSQL
- `pg_basebackup` sur l'esclave

Le maître se trouve dans le répertoire `/opt/postgresql-head/data`.

Configuration du `postgresql.conf` du maître:

```
port                            =                               5432
wal_level                       =                       hot_standby
archive_mode                            =                       on
archive_command   =   'cp   %p   /opt/postgresql-head/archives/%f'
max_wal_senders                         =                       2
hot_standby = on
```

Configuration du `pg_hba.conf` du maître :

```
local  replication  all  trust
```

Après redémarrage du maître, il est possible de créer le premier esclave :

```
$ pg_basebackup -D /opt/postgresql-head/data2 -p 5432 -c fast
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
```

Il est maintenant possible de configurer l'esclave. Voici les changements à partir de la configuration du maître.

Configuration du `postgresql.conf` sur l'esclave :

```
port = 5433
```

Aucun changement n'est nécessaire dans le fichier `pg_hba.conf` de l'esclave.

Configuration du `recovery.conf` sur l'esclave :

```
restore_command = 'cp /opt/postgresql-head/archives/%f %p'
standby_mode = on
primary_conninfo = 'port=5432'
```

Il ne reste plus qu'à démarrer l'esclave :

```
$ pg_ctl -D data2 start
server starting
$ psql -p 5433 -c "SELECT pg_is_in_recovery()" postgres
 pg_is_in_recovery
-------------------
 t
(1 row)
```

L'esclave est bien disponible en lecture seule. Tout ceci était déjà possible en 9.1.

La 9.2 nous permet d'attacher un esclave à un autre esclave, évitant ainsi une charge supplémentaire sur le maître. Comme la réplication est possible depuis l'esclave, il est également possible d'utiliser `pg_basebackup` depuis l'esclave. Nous spécifions donc le port 5433 :

```
$ pg_basebackup -D /opt/postgresql-head/data3 -p 5433 -c fast
```

Il faut maintenant configurer l'esclave pour que ce dernier utilise un autre numéro de port.

Configuration du `postgresql.conf` sur le deuxième esclave :

```
port = 5434
```

Il n'y a toujours pas de changement dans le fichier `pg_hba.conf` de l'esclave.

Configuration du `recovery.conf` sur l'esclave :

```
restore_command = 'cp /opt/postgresql-head/archives/%f %p'
standby_mode = on
primary_conninfo = 'port=5433'
```

Notez que le numéro de port est différent. Le deuxième esclave doit se connecter au premier esclave, disponible sur le port 5433.

Il ne reste plus qu'à démarrer l'esclave :

```
$ pg_ctl -D data3 start
```

```
server starting
$ psql -p 5434 -c "SELECT pg_is_in_recovery()" postgres
 pg_is_in_recovery
-------------------
 t
(1 row)
$ psql -p 5433 -x -c "SELECT * FROM pg_stat_replication" postgres
-[ RECORD 1 ]----+------------------------------
pid              | 6586
usesysid         | 10
usename          | guillaume
application_name | walreceiver
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2012-03-15 17:05:42.824842+01
state            | streaming
sent_location    | 0/1F0000B8
write_location   | 0/1F0000B8
flush_location   | 0/1F0000B8
replay_location  | 0/1F0000B8
sync_priority    | 0
sync_state       | async

$ psql -p 5432 -x -c "SELECT * FROM pg_stat_replication" postgres
-[ RECORD 1 ]----+------------------------------
pid              | 5939
usesysid         | 10
usename          | guillaume
application_name | walreceiver
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2012-03-15 16:48:49.087458+01
state            | streaming
sent_location    | 0/1F0000B8
write_location   | 0/1F0000B8
flush_location   | 0/1F0000B8
replay_location  | 0/1F0000B8
sync_priority    | 0
sync_state       | async
```

Le deuxième esclave s'est bien connecté au premier esclave, et ce dernier est toujours connecté au maître.

5.2 Réplication en mémoire

- Réplication en mémoire seulement sur l'esclave
- Avantage
  - meilleures performances
- Inconvénient
  - durabilité moindre
- Nouvelle valeur `remote_write` pour le paramètre `synchronous_commit`

En cas de réplication synchrone, ce nouveau mode de réplication permet au maître de n'attendre que l'écriture en mémoire des données sur l'esclave et non la synchronisation sur disque. Cela

permet un commit plus rapide, mais en cas de crash simultané du maître et de l'esclave, il peut y avoir une perte de donnée (mais pas de corruption).

6 Administration

- Nouvel outil pg_receivexlog
- Facilité d'administration
- Supervision
- Sécurité

PostgreSQL 9.2 apporte de nouvelles possibilités pour la facilité d'administration (des tablespaces, de l'autovacuum), la supervision (des nouvelles colonnes dans les tables statistiques) et la sécurité (des procédures stockées, des vues, etc).

6.1 Nouvel outil pg_receivexlog

- Utilise le protocole de réplication
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
- Va plus loin que l'archivage standard
  - Pas de `archive_timeout` car toujours au plus près du maître

`pg_receivexlog` est un nouvel outil permettant de se faire passer pour un esclave dans le but d'archiver des journaux de transactions au plus près du maître grâce à l'utilisation du protocole de réplication en flux.

Comme il utilise le protocole de réplication, les journaux archivés ont un retard bien inférieur à celui induit par le paramétrage de `archive_command`, les journaux de transactions étant écrits au fil de l'eau. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le maître.

Cet outil utilise les même options de connexion que la plupart des outils PostgreSQL, avec en plus l'option -D pour spécifier le répertoire où sauvegarder les journaux de transactions. L'utilisateur spécifié doit bien évidemment avoir les attributs `LOGIN` et `REPLICATION`.

```
pg_receivexlog receives PostgreSQL streaming transaction logs.

Usage:
  pg_receivexlog [OPTION]...
```

```
Options:
  -D, --directory=DIR    receive transaction log files into this directory
  -n, --no-loop          do not loop on connection lost
  -v, --verbose          output verbose messages
  -V, --version          output version information, then exit
  -?, --help             show this help, then exit

Connection options:
  -h, --host=HOSTNAME    database server host or socket directory
  -p, --port=PORT        database server port number
  -s, --status-interval=INTERVAL
                         time between status packets sent to server (in seconds)
  -U, --username=NAME    connect as specified database user
  -w, --no-password      never prompt for password
  -W, --password         force password prompt (should happen automatically)

Report bugs to <pgsql-bugs@postgresql.org>.
```

Voilà un exemple de mise en place :

- Modifications sur le serveur maître
  - Fichier `postgresql.conf`
    - `wal_level = hot_standby`
    - `max_wal_senders = 3`
  - Fichier `pg_hba.conf`
    - `host replication repli_user 192.168.0.0/24 trust`
  - Créer l'utilisateur de réplication
    - `psql -c “CREATE ROLE repli_user LOGIN REPLICATION”`

Une fois le serveur PostgreSQL 9.2 redémarré, on peut alors lancer `pg_receivexlog` :

```
pg_receivexlog -h 192.168.0.1 -U repli_user -D /data/archives
```

Les journaux de transactions sont alors créés en temps réel dans le répertoire indiqué (ici, `/data/archives`) :

```
-rwx------  1 postgres postgres   16777216 juil. 27 11:14 000000010000000000000000E*
-rwx------  1 postgres postgres   16777216 juil. 27 11:15 000000010000000000000000F*
-rwx------  1 postgres postgres   16777216 juil. 27 11:16 000000010000000000000010.partial*
```

En cas d'incident sur le maître, il est alors possible de partir d'une sauvegarde binaire et de rejouer les journaux de transactions disponibles (sans oublier de supprimer l'extension `.partial` du dernier journal).

6.2 Facilités d'administration

- Annulation de ses propres requêtes avec `pg_cancel_backend()` par un utilisateur de base
- Possibilité de déplacer plus facilement un tablespace, moteur éteint
- Plus de traces pour l'activité disque d'autovacuum

6.2.1.1 Nouvelle possibilité dans l'annulation de requête

En version 9.1, un utilisateur sans l'attribut SUPERUSER ne pouvait pas annuler ses propres requêtes:

```
demo_9_1=> SELECT pg_cancel_backend(procpid) FROM pg_stat_activity WHERE usename = current_user AND
procpid <> pg_backend_pid();
ERROR:  must be superuser to signal other server processes
```

C'est dorénavant possible en 9.2 :

```
demo_9_2=> SELECT pg_cancel_backend(pid) FROM pg_stat_activity WHERE usename = current_user AND pid
<> pg_backend_pid();
 pg_cancel_backend
-------------------
 t
(1 ligne)
```

6.2.1.2 Déplacer facilement un tablespace

Avant la 9.2, pour déplacer un tablespace, il fallait arrêter le moteur, déplacer le tablespace, reconstruire le lien symbolique et ne pas oublier de modifier le chemin qui était aussi enregistré dans le catalogue système `pg_tablespace`. Désormais, cette dernière modification n'est plus utile, le catalogue n'ayant plus cette information. Il est toujours possible de récupérer cette information avec une nouvelle procédure stockée appelée `pg_tablespace_location(oid)`. En voici la démonstration :

```
$ mkdir /opt/postgresql-head/ts1
$ psql postgres
psql (9.2devel)
Type "help" for help.

postgres=# CREATE TABLESPACE grosdisque LOCATION '/opt/postgresql-head/ts1';
CREATE TABLESPACE
postgres=# SELECT oid, * FROM pg_tablespace;
  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16400 | grosdisque |       10 |        |
(3 rows)
```

```
postgres=# SELECT pg_tablespace_location(16400);
  pg_tablespace_location
--------------------------
 /opt/postgresql-head/ts1
(1 row)

postgres=# \q
$ pg_ctl stop
waiting for server to shut down..... done
server stopped
$ mv /opt/postgresql-head/ts1 /opt/postgresql-head/ailleurs
$ ln -sf /opt/postgresql-head/ailleurs /opt/postgresql-head/data/pg_tblspc/16400
$ ll /opt/postgresql-head/data/pg_tblspc
total 0
lrwxrwxrwx. 1 postgres postgres 29 Mar 15 17:39 16400 -> /opt/postgresql-head/ailleurs
$ pg_ctl start
server starting
$ psql postgres
psql (9.2devel)
Type "help" for help.

postgres=# SELECT pg_tablespace_location(16400);
    pg_tablespace_location
------------------------------
 /opt/postgresql-head/ailleurs
(1 row)
```

6.2.1.3 Traces de l'autovacuum

Les traces de l'autovacuum vont bien plus loin : compteur de blocs disques (lus dans le cache, en dehors du cache, et modifiés) ainsi qu'un taux de lecture et d'écriture. Cet exemple le montre :

```
2012-07-25 12:32:58 CEST [13144]: [1-1] user=,db= LOG:  automatic
vacuum of table "test.public.t1": index scans: 2
                        pages: 0 removed, 63598 remain
                    tuples: 5000000 removed, 5000000 remain
          buffer usage: 245660 hits, 29 misses, 154542 dirtied
         avg read rate: 0.001 MiB/s, avg write rate: 3.467 MiB/s
           system usage: CPU 0.45s/13.12u sec elapsed 348.20 sec
2012-07-25 12:33:02 CEST [13144]: [2-1] user=,db= LOG:  automatic
analyze of table "test.public.t1" system usage: CPU 0.01s/0.29u
sec               elapsed               3.34               sec
```

6.3 Supervision

- pg_stat_activity, nouveau champ state
  - indique l'état des processus serveur
  - modification de current_query
- pg_stat_activity, champ procpid renommé en pid
- pg_stat_database, nouveaux champs temp_files et temp_bytes
  - indique le nombre et la taille totale des fichiers temporaires
- pg_stat_database, nouveau champ deadlocks
  - indique le nombre de deadlocks

Exemple de ces nouveaux champs :

```
postgres=# SELECT * FROM pg_stat_activity ;
-[ RECORD 1 ]----+---------------------------------
datid            | 12893
datname          | postgres
pid              | 7649
usesysid         | 10
usename          | guillaume
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2012-03-15 17:43:37.582141+01
xact_start       | 2012-03-15 17:44:13.562294+01
query_start      | 2012-03-15 17:44:13.562294+01
state_change     | 2012-03-15 17:44:13.562297+01
waiting          | f
state            | active
query            | select * from pg_stat_activity ;

postgres=# SELECT * FROM pg_stat_database WHERE datname='postgres';
-[ RECORD 1 ]-+------------------------------
datid         | 12893
datname       | postgres
numbackends   | 1
xact_commit   | 371
xact_rollback | 8
blks_read     | 225097
blks_hit      | 17074184
tup_returned  | 19030603
tup_fetched   | 1160116
tup_inserted  | 8000094
tup_updated   | 2500010
tup_deleted   | 1000007
conflicts     | 0
temp_files    | 8
temp_bytes    | 149978752
deadlocks     | 0
stats_reset   | 2012-03-15 15:38:30.596015+01
```

6.4 Sécurité

- Support des labels de sécurité sur les objets partagés
- Droit sur l'objet TYPE (USAGE)
- Vue : option `security_barrier`
- Procédure stockée : option `leakproof`

6.5 Sécurité - Vues avec barrière de sécurité

- Nouvelle option security_barrier
- Utiliser les vues comme moyen de filtrer les lignes est dangereux
- Si l'utilisateur a la possibilité de créer une fonction, il peut facilement contourner cette sécurité…
  - sauf en 9.2 avec l'option `security_barrier`

Voici un exemple complet.

```
postgres=# CREATE TABLE elements (id serial, contenu text, prive boolean);
NOTICE:  CREATE TABLE will create implicit sequence "elements_id_seq" for serial column
"elements.id"
CREATE TABLE
postgres=# INSERT INTO elements (contenu, prive)
VALUES ('a', false), ('b', false), ('c super prive', true), ('d', false), ('e prive aussi', true);
INSERT 0 5
postgres=# SELECT * FROM elements;
 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
  5 | e prive aussi | t
(5 rows)
```

La table `elements` contient cinq lignes, trois considérés comme privés. Nous allons donc créer une vue ne permettant de ne voir que les lignes publiques.

```
postgres=# CREATE OR REPLACE VIEW elements_public AS SELECT * FROM elements WHERE CASE WHEN
current_user='guillaume' THEN TRUE ELSE NOT prive END;
CREATE VIEW
postgres=# SELECT * FROM elements_public;
 id |    contenu    | prive
----+---------------+-------
  1 | a             | f
  2 | b             | f
  3 | c super prive | t
  4 | d             | f
```

```
  5 | e prive aussi | t
(5 rows)

postgres=# CREATE USER u1;
CREATE ROLE
postgres=# GRANT SELECT ON elements_public TO u1;
GRANT
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT * FROM elements;
ERROR:  permission denied for relation elements
postgres=> SELECT * FROM elements_public ;
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
(3 rows)
```

L'utilisateur u1 n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Avec une simple fonction, cela peut changer :

```
postgres=> CREATE OR REPLACE FUNCTION abracadabra(integer, text, boolean) RETURNS bool
AS $$
BEGIN
RAISE NOTICE '% - % - %', , , ;
RETURN true;
END$$
LANGUAGE plpgsql
COST 0.00000000000000000001;
CREATE FUNCTION
postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);
NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  3 - c super prive - t
NOTICE:  4 - d - f
NOTICE:  5 - e prive aussi - t
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
(3 rows)
```

Que s'est-il passé ? pour comprendre, il suffit de regarder l'EXPLAIN de cette requête :

```
postgres=> EXPLAIN SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);
                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------------------
----------------------
 Seq Scan on elements  (cost=0.00..28.15 rows=202 width=37)
   Filter: (abracadabra(id, contenu, prive) AND CASE WHEN ("current_user"() = 'u1'::name) THEN (NOT
prive) ELSE true END)
(2 rows)
```

La fonction `abracadrabra` a un coût si faible que PostgreSQL l'exécute avant le filtre de la vue. Du coup, la fonction voit toutes les lignes de la table.

Seul moyen d'échapper à cette optimisation du planificateur, utiliser l'option `security_barrier` en 9.2 :

```
postgres=> \c - guillaume
You are now connected to database "postgres" as user "guillaume".
postgres=# CREATE OR REPLACE VIEW elements_public WITH (security_barrier) AS SELECT * FROM elements
WHERE CASE WHEN current_user='guillaume' THEN true ELSE NOT prive END;
CREATE VIEW
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);
NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  4 - d - f
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
(3 rows)

postgres=> EXPLAIN SELECT * FROM elements_public WHERE abracadabra(id, contenu, prive);
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..34.20 rows=202 width=37)
   Filter: abracadabra(elements_public.id, elements_public.contenu, elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.15 rows=605 width=37)
         Filter: CASE WHEN ("current_user"() = 'u1'::name) THEN (NOT prive) ELSE true END
(4 rows)
```

Voir aussi http://rhaas.blogspot.com/2012/03/security-barrier-views.html .

6.6 Sécurité : fonction LEAKPROOF

- Une fonction LEAKPROOF est une fonction sûre
- Son créateur garantit qu'elle ne laisse pas échapper d'informations
- Important pour le planificateur

Le fait de déclarer une fonction comme LEAKPROOF autorisera le planificateur à évaluer le contenu de la fonction à n'importe quel moment de l'exécution de la requête, puisque, ne laissant échapper aucune information, le problème vu précédemment avec le paramètre security_barrier ne pourra pas se produire. Il est donc possible d'avoir de meilleurs plans d'exécution selon les cas.

7 Fonctionnalités utilisateur

- JSON
- Range Type
- pg_stats_statement
- pg_receivexlog
- Procédures stockées

7.1 json

- Fonctionne globalement comme le type XML
- Validation
- Fonctions spécifiques
  - array_to_json
  - row_to_json

Les données JSON sont de plus en plus fréquentes. Elles sont très présentes en Javascript, ainsi que dans d'autres langages ciblés web.

PostgreSQL est capable d'enregistrer et de générer des données au format json, dont la validité est assurée. PostgreSQL dispose de fonctions capables de traiter des données au format JSON :

```
postgres=# CREATE TABLE t6 (x json);
CREATE TABLE
postgres=# INSERT INTO t6 (x) VALUES ('{"valeur":123}');
INSERT 0 1
postgres=# SELECT * FROM t6;
       x
----------------
 {"valeur":123}
(1 row)

postgres=# INSERT INTO t6 (x) VALUES ('{"valeur":123');
ERROR:  invalid input syntax for type json: "{"valeur":123"
LINE 1: INSERT INTO t6 (x) VALUES ('{"valeur":123');
                                   ^
DETAIL:  The input string ended unexpectedly.
postgres=# INSERT INTO t6 (x) VALUES ('autre test');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO t6 (x) VALUES ('autre test');
                                   ^
DETAIL:  line 1: Token "autre" is invalid.
postgres=# SELECT row_to_json(db) FROM pg_database db ;
                                          row_to_json
```

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

{"datname":"template1","datdba":10,"encoding":6,"datcollate":"en_US.utf8","datctype":"en_US.utf8","
datistemplate":true,"datallo
wconn":true,"datconnlimit":-
1,"datlastsysoid":12888,"datfrozenxid":"1831","dattablespace":1663,"datacl":
["=c/guillaume","guillau
me=CTc/guillaume"]}

{"datname":"template0","datdba":10,"encoding":6,"datcollate":"en_US.utf8","datctype":"en_US.utf8","
datistemplate":true,"datallo
wconn":false,"datconnlimit":-
1,"datlastsysoid":12888,"datfrozenxid":"1831","dattablespace":1663,"datacl":["=c/guillaume","guilla
ume=CTc/guillaume"]}

{"datname":"postgres","datdba":10,"encoding":6,"datcollate":"en_US.utf8","datctype":"en_US.utf8","d
atistemplate":false,"datallo
wconn":true,"datconnlimit":-
1,"datlastsysoid":12888,"datfrozenxid":"1831","dattablespace":1663,"datacl":null}
(3 rows)
```

7.2 Type de données range

- Permet d'exprimer un intervalle mathématique de valeurs
- Valable pour différents types
  - int4range
  - numrange
  - tsrange, tstzrange, daterange
  - etc
- Et de faire des tests par rapport à cet intervalle mathématique de valeurs

Est-ce que le 15 janvier fait partie de l'intervalle mathématique du 1er janvier 2012 exclu au 1er février 2012 inclus ?

```
postgres=# select '("2012-01-01", "2012-02-01"]'::daterange @> '2012-01-15'::date;
 ?column?
----------
 t
(1 row)
```

Oui. Et est-ce que le 15 février fait partie de cet intervalle mathématique ?

```
postgres=# select '("2012-01-01", "2012-02-01"]'::daterange @> '2012-02-15'::date;
 ?column?
----------
 f
(1 row)
```

Non.

Cette fonctionnalité propose ce que faisait l'extension temporal, mais en plus global car elle peut gérer n'importe quel type sous-jacent (par défaut, seuls les entiers sur 4 ou 8 bits, les numerics, les timestamps avec ou sans time zone ainsi que les dates sont gérés, mais on peut créer ses propres types d'intervalle mathématique).

7.3 Module pg_stat_statements

- Existe depuis la version 8.4
- Ajout de la normalisation des requêtes en 9.2

L'extension pg_stat_statements permet de tracer facilement l'ensemble des requêtes sur un serveur sans trop impacter les performances, par rapport à une solution d'analyse de traces. La nouvelle version de pg_stat_statements permet maintenant de normaliser les requêtes, de la même façon que les outils d'analyse de traces tel que pgBadger, ce qui rend son utilisation beaucoup plus intéressante. Voila un exemple de mise en place :

Dans le postgresql.conf

```
shared_preload_library = 'pg_stat_statements'
[...]
pg_stat_statements.max = 10000;
```

Redémarrage du serveur obligatoire (pg_ctl -D /var/lib/pgsql/9.2/data restart)

Installation de l'extension pg_stat_statements :

```
postgresql@demo_9_2=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
```

Utilisation :

```
postgres@test=# CREATE TABLE t1(c1 integer primary key, c2 varchar);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t1_pkey" for table "t1"
CREATE TABLE
postgres@test=# INSERT INTO t1 SELECT i, 'Ligne ' || i FROM generate_series(1,10000) i;
INSERT 0 10000
postgres@test=# SELECT * FROM t1 WHERE c1 = 50;
 c1  |    c2
-----+-----------
 50 | Ligne 50
(1 row)


postgres@test=# SELECT * FROM t1 WHERE c1 = 150;
 c1  |    c2
-----+-----------
 150 | Ligne 150
(1 row)


postgres@test=# SELECT * FROM t1 WHERE c1 = 350;
 c1  |    c2
```

```
-----+-----------
 350 | Ligne 350
(1 row)


postgres@test=# SELECT * FROM t1 WHERE c1 = 450;
 c1  |    c2
-----+-----------
 450 | Ligne 450
(1 row)

postgres@test=# SELECT query,calls,total_time, rows
postgres@test-# FROM pg_stat_statements
postgres@test-# WHERE query LIKE '%t1%';

                           query                            | calls | total_time | rows
------------------------------------------------------------+-------+------------+-------
 INSERT INTO t1 SELECT i, ? || i FROM generate_series(?,?) i; |     1 |     36.171 | 10000
 SELECT * FROM t1 WHERE c1 = ?;                               |     4 |      0.152 |     4
 CREATE TABLE t1(c1 integer primary key, c2 varchar);         |     2 |     203.93 |     0
(3 rows)
```

7.4 Procédures stockées

- Langage SQL : référence des paramètres par des noms
- Langage PL/pgsql : ajout de `GET STACKED DIAGNOSTICS`
- Fonctions trigger : ajout de la fonction `pg_trigger_depth()`

7.4.1.1 Référence des paramètres par nom

C'était possible avec les langages PL, c'est maintenant possible avec une procédure stockée en SQL.

```
postgres=> CREATE FUNCTION incremente(valeur integer) RETURNS integer AS $$
SELECT valeur+1;
$$ LANGUAGE sql IMMUTABLE;
postgres=> SELECT incremente(10);
 incremente
------------
         11
(1 row)
```

7.4.1.2 GET STACKED DIAGNOSTICS

Cette nouvelle instruction permet de récupérer beaucoup d'informations sur l'exception qui a eu lieu :

```
postgres=# CREATE TABLE t5(c1 integer PRIMARY KEY);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t5_pkey" for table "t5"
CREATE TABLE
postgres=# INSERT INTO t5 VALUES (1);
INSERT 0 1
```

```
postgres=# CREATE OR REPLACE FUNCTION test(INT4) RETURNS void AS $$
DECLARE
    v_state   TEXT;
    v_msg     TEXT;
    v_detail  TEXT;
    v_hint    TEXT;
    v_context TEXT;
BEGIN
    BEGIN
        INSERT INTO t5 (id) VALUES ();
    EXCEPTION WHEN others THEN
        GET STACKED DIAGNOSTICS
            v_state   = RETURNED_SQLSTATE,
            v_msg     = MESSAGE_TEXT,
            v_detail  = PG_EXCEPTION_DETAIL,
            v_hint    = PG_EXCEPTION_HINT,
            v_context = PG_EXCEPTION_CONTEXT;
        raise notice E'Et une exception :
            state  : %
            message: %
            detail : %
            hint   : %
            context: %', v_state, v_msg, v_detail, v_hint, v_context;
    END;
    RETURN;
END;
$$ LANGUAGE plpgsql;
postgres=# SELECT test(2);
 test
------

(1 row)

postgres=# SELECT test(2);
NOTICE:  Et une exception :
            state  : 23505
            message: duplicate key value violates unique constraint "t5_pkey"
            detail : Key (c1)=(2) already exists.
            hint   :
            context: SQL statement "INSERT INTO t5 (c1) VALUES ()"
PL/pgSQL function test(integer) line 10 at SQL statement
 test
------

(1 row)
```

7.5 SQL : ordres

- Nouvelle clause `IF EXISTS` pour les commandes `ALTER`
- Facilite l'écriture de scripts de mise à jour de schémas
- Nouvelles commandes `ALTER`
- nouvelle méthode : SP-GiST (pour Space Partitioned GiST)

Exemple des nouvelles commandes :

```
ALTER FOREIGN TABLE IF EXISTS nom RENAME TO nouveau_nom
ALTER DOMAIN IF EXISTS ...
```

Les ALTER suivants existent maintenant :

- ALTER FOREIGN DATA WRAPPER / RENAME
- ALTER SERVER / RENAME
- ALTER DOMAIN / RENAME

Les index SP-GiST sont des index similaires aux index GiST de part leur flexibilité, mais permettent de construire des arbres différents (suffixés, non balancés, décomposés dans l'espace …). Cela permet des gains de performances dans certains cas, la recherche dans l'index étant dépendante uniquement de la taille de la requête.

8 Régressions

- Suppression de l'opérateur ⇒ pour les hstore
- Suppression du champ `spclocation` de la table `pg_tablespace`
- Extract epoch prend en compte le fuseau horaire local
- Modification de `to_date` et `to_timestamp` avec une date sur trois chiffres
- Modification de la vue `pg_stat_activity`
- Fonction xpath()
- Chronométrages statistiques en millisecondes
- Suppression de paramètres dans le `postgresql.conf`

8.1 Opérateur => (hstore)

En 9.1 :

```
SELECT pg_typeof('a'=>'b');
 pg_typeof
-----------
 hstore
(1 row)

=# SELECT 'a'=>'b';
 ?COLUMN?
----------
 "a"=>"b"
(1 row)

=# SELECT pg_typeof('a'=>'b');
 pg_typeof
-----------
 hstore
(1 row)
```

En 9.2:

```
SELECT 'a'=>'b';
ERROR:  operator does NOT exist: unknown => unknown at character 11
HINT:  No operator matches the given name AND argument type(s). You might need TO ADD explicit type
casts.
STATEMENT:  SELECT 'a'=>'b';
ERROR:  operator does NOT exist: unknown => unknown
LINE 1: SELECT 'a'=>'b';
                  ^
HINT:  No operator matches the given name AND argument type(s). You might need TO ADD explicit type
casts.
```

8.2 pg_tablespace

Depuis la version 9.2, l'information de location d'un tablespace ne se trouve plus dans la table mais est directement récupérée du point de montage. Cela permet de changer facilement l'emplacement d'un tablespace lorsque la base est arrêtée. Il faut donc maintenant utiliser la fonction `pg_tablespace_location()`:

```
=# SELECT *, pg_tablespace_location(oid) AS spclocation FROM pg_tablespace;
  spcname   | spcowner | spcacl | spcoptions |  spclocation
------------+----------+--------+------------+----------------
 pg_default |       10 |        |            |
 pg_global  |       10 |        |            |
 tmptblspc  |       10 |        |            | /tmp/tmptblspc
```

8.3 Extract epoch

Jusqu'en 9.1, l'extraction de epoch (nombre de secondes depuis le 1er janvier 1970 à minuit) prenait toujours comme référence le “minuit” du fuseau horaire UTC, que la date ait ou pas un fuseau horaire. À partir de la version 9.2, si la date n'a pas de fuseau horaire, epoch est calculé en fonction de minuit, heure locale.

En 9.1 :

```
=#SELECT extract(epoch FROM '2012-07-02 00:00:00'::timestamp);
 date_part
------------
 1341180000
(1 row)

=# SELECT extract(epoch FROM '2012-07-02 00:00:00'::timestamptz);
 date_part
------------
 1341180000
(1 row)
```

En 9.2 :

```
=#SELECT extract(epoch FROM '2012-07-02 00:00:00'::timestamp);
 date_part
------------
 1341187200
(1 row)

=# SELECT extract(epoch FROM '2012-07-02 00:00:00'::timestamptz);
 date_part
------------
 1341180000
```

```
(1 row)
```

8.4 to_date et to_timestamp

La façon dont est calculée une date selon qu'elle est sur 2 ou 3 chiffres n'était pas cohérente: les dates sur 2 chiffres étaient toujours rapprochées de la date la plus proche de 2020, et celles sur 3 chiffres rapprochées de 1100 à 1199 pour les valeurs de 999 à 1100, et de 2000 à 2099 pour les valeurs de 000 à 099.

Dorénavant, PostgreSQL choisit la date la plus proche de 2020 dans tous les cas.

Exemple en 9.1:

```
=# SELECT to_date('200-07-02','YYY-MM-DD');
  to_date
------------
 1200-07-02
```

En 9.2:

```
SELECT to_date('200-07-02','YYY-MM-DD');
  to_date
------------
 2200-07-02
```

8.5 Vue pg_stat_activity

La définition de la vue `pg_stat_activity` a changée. La colonne `state` a été ajoutée. Elle donne l'état de la session au moment ou la vue est appelée. Elle peut prendre les valeurs suivante :

- active: Le processus serveur exécute une requête;
- idle: Le processus serveur est en attente d'une nouvelle commande;
- idle in transaction: Le processus serveur est en transaction, mais n'exécute pas de requête;
- idle in transaction (aborted): Similaire à `idle in transaction`, sauf qu'une des requêtes de la transaction a causé une erreur;
- fastpath function call: Le processus serveur exécute une fonction fast-path;
- disabled: Affiché si le paramètre `track_activities` est désactivé pour ce processus serveur.

Le champ `procpid` a été renommé en `pid`, pour plus de cohérence avec les autres vues système. Le champ `query` affiche désormais la dernière requête a avoir été exécutée, sauf si `state` vaut `active`, auquel cas il montrera la requête en cours d'exécution.

8.6 Fonction xpath()

La fonction xpath échappe désormais les caractères spéciaux.

Exemple en 9.1 :

```
SELECT (XPATH('/*/text()', '<root>&lt;</root>'))[1];
 xpath
-------
 <
```

'<' n'est pas du XML valide.

En 9.2:

```
SELECT (XPATH('/*/text()', '<root>&lt;</root>'))[1];
 xpath
-------
 &lt;
```

8.7 Chronométrage en millisecondes

Les champs pg_stat_user_functions.total_time, pg_stat_user_functions.self_time, pg_stat_xact_user_functions.total_time, pg_stat_xact_user_functions.self_time et pg_stat_statements.total_time (contrib) sont maintenant en millisecondes, pour être cohérent avec les autres valeurs de chronométrage.

8.8 postgresql.conf

Le mode silencieux a été supprimé. Il faut maintenant utiliser `pg_ctl -l postmaster.log`.
Le paramètre `wal_sender_delay` a été supprimée, celui-ci n'étant plus nécessaire avec l'utilisation des `latches`.
Le paramaètre `custom_variable_classes` a été supprimé. Toutes les «`classes`» sont maintenant acceptées sans déclaration préalable.
Les paramètres `ssl_ca_file`, `ssl_cert_file`, `ssl_crl_file`, `ssl_key_file` ont été ajoutés, ce qui veut dire qu'il est maintenant possible de spécifier les fichier utilisés pour l'authentification SSL.

9 Bilan

- Des fonctionnalités demandées par les utilisateurs
- Des innovations importantes
- Un gros travail sur les performances
- Le tout respectant le mode de fonctionnement de la communauté

Le développement réalisé pour la version 9.2 a été une nouvelle fois très important. Les nouvelles fonctionnalités sont nombreuses, les performances ont été améliorées, ainsi que la facilité d'utilisation.

10 Et la suite ?

- Version 9.3 prévue pour mi 2013
- Déjà présent
  - Trigger sur DDL (`event trigger`)
  - Changement de gestion du cache de PostgreSQL
  - Performances des colonnes de type range
- En cours
  - Requêtes parallèles
  - Vue matérialisée

La prochaine version devrait être la 9.3. Étant donné le cycle de développement actuel, il faut s'attendre à une sortie mi-2013.

Le développement a commencé très fort. Des nouveaux triggers ont été ajoutés. Ils sont capables de se déclencher sur des opérations DDL. C'est justement ce qui manquaient à des outils de réplication comme Slony ou Londiste pour faciliter leur administration. Il ne fait aucun doute que leurs développeurs vont les utiliser rapidement.

Un point très gênant sur les performances de PostgreSQL était lié au type de mémoire partagée utilisée. Pour avoir de grosses quantités de mémoire, il fallait généralement configurer le noyau. L'implémentation de la mémoire partagée a changé, ce qui fait que cette configuration du noyau Linux ne sera plus utile.

De nombreux autres développements sont en cours, comme un pg_dump parallélisé.

11 Pour aller plus loin

- La documentation officielle en ligne
- What's new in PostgreSQL 9.2 ?
- Des articles dans GNU/Linux Magazine France (bientôt…)
- Télécharger cette conférence : http://www.dalibo.org/conferences

La meilleur source d'information reste avant tout la documentation officielle de PostgreSQL 9.2. Vous y trouverez les informations essentielles sur chaque nouveautés :

http://docs.postgresql.fr/9.2/

Plus spécifiquement, vous pouvez consultez le guide illustré de PostgreSQL 9.2, résultat du travail de Marc Cousin.

Enfin vous pouvez télécharger cette présentation à l'adresse suivante :

https://support.dalibo.com/kb/conferences/

12 Questions ?

- N'hésitez pas, c'est le moment !