DALIBO

L'expertise PostgreSQL

Nouveautés de PostgreSQL 9.4

Table des matières

Nouveautés de PostgreSQL 9.4....3
1 L'auteur....3
1.1 Licence Creative Commons CC-BY-NC-SA....4
2 Introduction....5
3 Au menu....5
4 Réplication....6
4.1 Slots de réplication....6
4.2 Délai d'application....8
4.3 Journalisation des hints bits....8
4.4 Modifications logiques....9
5 SQL....10
5.1 Vues....10
5.2 JSON....11
5.3 Clause WITH ORDINALITY....13
5.4 Clause FILTER....14
5.5 Clause WITHIN GROUP....15
5.6 Clause ROWS FROM()....16
5.7 Triggers sur tables distantes....16
6 Administration....18
6.1 CREATE TABLESPACE....18
6.2 Déplacement d'objets....19
6.3 Verrous sur la commande ALTER TABLE....19
6.4 Configuration à distance....20
6.5 Nouveau paramètre pour l'autovacuum....21
6.6 Formatage plus aisé du préfixe des traces....21
6.7 Nouvelles valeurs par défaut....21
6.8 Nouvelle cible de restauration....22
6.9 Processus en tâche de fond....25
6.10 Divers....26
7 Supervision....27
7.1 Nouvelle vue pg_stat_archiver....27
7.2 Compteur de lignes modifiées....28
7.3 Identifiants de transaction....28
8 Performances....29
8.1 Nouvel outil pg_prewarm....29
8.2 Meilleure compression des WAL....30
8.3 Nouveautés du EXPLAIN....30
8.4 Améliorations des index GIN....31
8.5 Divers....33
9 Régressions / changements....33
10 Conclusion....34

Nouveautés de PostgreSQL 9.4

Photographie récupérée sur https://www.flickr.com/photos/tripletsisters/6932000465/
Prise par thethreesisters Licence CC BY-NC-ND 2.0

1 L'auteur

- Auteur : Guillaume Lelarge
- Société : Dalibo
- Date : Juin 2014
- URL : https://support.dalibo.com/kb/conferences/nouveautes_de_postgresql_9.4/

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

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **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

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

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

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

À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse:
http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 Introduction

- Développement commencé en mai 2013
- Déjà 14 mois de développement
- Beta 2 sortie
  - **À tester !!**
- Une sortie finale prévue pour octobre 2014

Comme toutes les versions majeures de PostgreSQL, le développement se fait en un peu plus d'un an et est suivi de quelques mois de tests approfondis. Le développement de la version 9.4 a donc commencé en mai 2013. Après 14 mois de développement, une version beta 1 est sortie. La beta 2 est sortie en juillet. Une beta 3 est prévue rapidement. La version finale est prévue pour la rentrée 2014.

3 Au menu

- Réplication
- SQL
- Administration
- Supervision
- Performances
- Régressions / changements
- Et une petite ouverture vers le futur

Pour cette nouvelle version, l'équipe de développement de PostgreSQL a suivi différents axes. Encore une fois, il n'a pas été simple de catégoriser chaque nouvelle fonctionnalité.

Il n'empêche que la réplication est de nouveau en tête de liste, que le langage SQL n'a pas été oublié, ainsi que la facilité d'administration et de supervision. Les performances sont meilleures grâce à un ensemble d'optimisations intéressantes.

4 Réplication

- Slots de réplication
- Rejeu retardé
- Journalisation des hints bits
- Début de l'implémentation de la réplication logique

Deux grosses fonctionnalités ont été sujet à beaucoup de discussions sur la liste des développeurs, pendant toute l'année de développement de la version 9.4. La première concerne les slots de réplication qui permettront de ne plus avoir d'esclaves désynchronisés à cause d'un retard trop important. La deuxième n'en est qu'à l'implémentation : la réplication logique.

Cela n'a pas empêché d'autres fonctionnalités de voir le jour :

- un rejeu retardé sur certains esclaves ;
- la journalisation des hints bits ;
- etc.

4.1 Slots de réplication

- Sur le maître
  - paramètre max_replication_slots
  - fonction pg_create_physical_replication_slot
- Sur l'esclave
  - paramètre primary_slot_name
- Sur le maître
  - vue pg_replication_slots

Depuis quelques versions, il est possible d'avoir un esclave sans archivage des journaux de transactions. C'est intéressant pour diminuer le trafic réseau entre le maître et les esclaves. Cependant, c'est aussi dangereux. En cas de grosse activité sur le maître et d'un retard de l'esclave, ce dernier peut se retrouver désynchronisé et incapable d'être mis à jour car le maître aura réutilisé les journaux encore utiles à l'esclave.

De plus, il est possible que `VACUUM` nettoie des enregistrements qui sont toujours nécessaires sur le secondaire, causant des conflits de réplication. Le paramètre `hot_standby_feedback` permet d'éviter cela.

La version 9.4 ajoute le concept de slots de réplication. Un slot permet à un esclave d'indiquer où il en est de la restauration. Cette information reste au niveau du maître, que l'esclave soit connecté ou non. Cela permet au maître de ne pas supprimer ou recycler des journaux de transactions toujours utiles à au moins un esclave. De plus, si `hot_standby_feedback` est activé, le maître empêchera `VACUUM` de nettoyer les tuples supprimés qui seraient encore utilisés sur l'esclave.

Il faut noter qu'il est nécessaire d'indiquer dans la configuration le nombre de slots souhaités. Cela correspond au paramètre `max_replication_slots`. Une fois modifié, il faut redémarrer le maître. La fonction `pg_create_physical_replication_slot()` permet de créer autant de slots que nécessaire (à priori un slot par esclave). Il faut donner à cette fonction le nom du slot. Ce nom sera indiqué dans la configuration de l'esclave au niveau du paramètre `primary_slot_name` du fichier `recovery.conf`. Là-aussi, il faut redémarrer l'esclave (si ce dernier était déjà démarré). Au démarrage de l'esclave, ce dernier va utiliser le slot indiqué pour stocker sa position dans les journaux de transactions.

Pour superviser les slots, il existe une vue appelée `pg_replication_slots` comprenant une ligne par slot.

Voyons cela en exemple. Une fois la réplication mise en œuvre, il suffit d'ajouter le paramètre `max_replication_slots`.

Ensuite, sur le maître, on peut créer le slot:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('instance2_slot');
   slot_name    | xlog_position
----------------+---------------
 instance2_slot |
(1 ROW)
```

La vue pg_replication_slots donne ensuite les informations nécessaires à la supervision:

```
postgres=# SELECT * FROM pg_replication_slots ;
   slot_name    | plugin | slot_type | datoid | DATABASE | active | xmin | catalog_xmin |
restart_lsn
----------------+--------+-----------+--------+----------+--------+------+--------------
+-------------
 instance2_slot |        | physical  |        |          | f      |      |              |
(1 ROW)
```

Les colonnes disponibles sont les suivantes :

- `slot_name` est le nom du slot ;
- `slot_type` peut valoir *physical* ou *logical* selon le type de réplication ;
- `active` permet de savoir si le slot est actuellement utilisé ;
- `xmin` correspond à la plus ancienne transaction que ce slot doit conserver (ie, qui n'a pas été appliquée), si `hot_standby_feedback` est positionné à on ;
- `catalog_xmin` est la plus ancienne transaction touchant les catalogues système ;
- `restart_lsn` correspond à la plus ancienne position à conserver dans les journaux

de transactions.

Les colonnes `plugin`, `datoid` et `database` ne sont pas utilisées pour les slots physiques, nous en parlerons à propos de la réplication logique.

4.2 Délai d'application

- Paramètre recovery_min_apply_delay
- Retarde le rejeu des transactions
  - pour éviter de rejouer des erreurs
- Plus de délai une fois le nœud promu
- Horloges des nœuds à synchroniser !

Le nouveau paramètre recovery_min_apply_delay permet d'ajouter un délai dans le rejeu des transactions. Si ce paramètre vaut cinq minutes, toute transaction validée sur le maître le sera sur l'esclave avec cinq minutes de retard. Le délai est appliqué sur l'heure du `COMMIT` sur le maître. Donc si les horloges ne sont pas synchronisées ou si l'esclave a un retard très important, le rejeu pourrait se faire dès la réception.

Il est aussi à noter que le délai n'est plus pris en compte quand le nœud esclave est promu d'une manière ou d'une autre. Si l'on souhaite mettre en œuvre ce retard pour pouvoir basculer sur le secondaire après une erreur sur le maître (par exemple, `DELETE` sans clause `WHERE`), la promotion de l'esclave appliquera les modifications en attente.

On peut contourner cette limitation en traitant les WAL en attente d'application comme des WAL archivés, et faire une restauration PITR à partir de ceux-ci.

4.3 Journalisation des hints bits

- Paramètre wal_log_hints
- Utile pour les outils examinant les pages modifiées
  - pg_rewind
- Permet aussi de tester la quantité supplémentaire de journalisation avec les sommes de contrôle

Les hint bits permettent, pour un tuple, de stocker localement le statut de sa transaction sans aller vérifier le fichier pg_clog.

La journalisation de la mise à jour des hint bits est très intéressante pour les outils qui veulent examiner les pages modifiées. L'exemple typique est l'outil `pg_rewind`.

Cette journalisation n'est activée par défaut que lors de l'utilisation des sommes de contrôle. Pour la mettre en place, il faut configurer le paramètre `wal_log_hints` à on.

4.4 Modifications logiques

- Modifications logiques envoyées dans le flux de réplication
- Nouvelle valeur pour wal_level : logical
- Nouvel outil pg_recvlogical
- Nouvelles fonctions SQL
- Nouveau module test_decoding
- Nouveau paramètre table sur REPLICA IDENTITY

2ndQuadrant travaille énormément sur le concept de réplication logique. L'idée est d'envoyer les informations logiques (ie, quelle ligne a été impactée et comment), plutôt que sur des informations physiques (l'information est au niveau du bloc de données dans les fichiers). Le but à terme est de pouvoir dépasser certaines limitations de la réplication interne de PostgreSQL. Un début d'implémentation est mis en place.

Il existe donc une nouvelle valeur pour le paramètre `wal_level`, à savoir `logical`. Si cette valeur est configurée pour le paramètre `wal_level`, PostgreSQL ajoutera dans les journaux de transactions des informations sur les modifications logiques.

Il existe actuellement deux moyens de lire ces informations : `pg_recvlogical` et de nouvelles fonctions SQL (`pg_logical_slot_peek_changes`, `pg_logical_slot_get_changes`, `pg_logical_slot_peek_binary_changes` et `pg_logical_slot_get_binary_changes`).

Associés à cet outil et à ces fonctions, PostgreSQL utilise des modules qui permettent de formater la sortie selon le format souhaité. Un plugin d'exemple `test_decoding` est fourni avec les sources de PostgreSQL. Deux autres plugins existent déjà : wal2json et decoder_raw. Le premier permet d'obtenir une sortie au format JSON, le second permet de restituer des ordres SQL équivalents à ceux ayant entrainé les modifications. Le plugin `test_decoding` est celui qui est utilisé pr pg_recvlogical par défaut.

Parallèlement à ce travail, 2nd Quadrant propose une implémentation proof-of-concept étendant les fonctionnalités développées dans la 9.4 pour mettre en place une réplication bi-directionnelle. Si leurs travaux sont concluants, ceci sera peut être intégré dans une future version de PostgreSQL. Pour plus d'informations, voir ici: http://2ndquadrant.com/en/resources/bdr/

Mais pour l'instant, nous n'en sommes réellement qu'au début de l'implémentation. Il est préférable d'éviter l'activation de cette réplication en production (car cela diminue les performances sans ajouter en fonctionnalités).

5 SQL

- Amélioration sur les vues
- JSON
- Nouvelles clauses
  - WITH ORDINALITY
  - FILTER
  - WITHIN GROUP
- Triggers sur tables distantes

5.1 Vues

- Rafraichissement sans blocage des vues matérialisées
- Amélioration des vues automatiquement modifiables, sont prises en comptes
  - les vues contenant des colonnes non modifiables
  - les vues security_barrier
- Nouvelle option WITH CHECK OPTION
  - option local, vue seule
  - option cascade, table aussi

Lors de l'ajout des vues matérialisées avec la version 9.3, deux gros manques ont gêné leur utilisation. Parmi ces deux manques figurait le verrouillage exclusif de la vue pendant son rafraichissement. Notamment, cela empêchait sa lecture pendant son rafraichissement. La version 9.4 corrige cela avec la clause `CONCURRENTLY` de l'instruction `REFRESH MATERIALIZED VIEW`. Cependant, il n'est possible d'utiliser cette clause que s'il existe un index unique portant seulement sur des noms de colonne et sur l'intégralité des lignes de la vue. En voici un exemple complet :

```
postgres=# CREATE TABLE t1 (c1 integer, c2 text);
CREATE TABLE
postgres=# INSERT INTO t1 SELECT i, 'ligne '||i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
postgres=# CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM t1 WHERE c1<10000;
SELECT 9999
postgres=# REFRESH MATERIALIZED VIEW CONCURRENTLY mv1;
ERROR:  cannot refresh materialized view "public.mv1" concurrently
HINT:  Create a UNIQUE index with no WHERE clause on one or more columns of the materialized view.
postgres=# CREATE UNIQUE INDEX ON mv1(c1);
CREATE INDEX
postgres=# REFRESH MATERIALIZED VIEW CONCURRENTLY mv1;
```

```
REFRESH MATERIALIZED VIEW
```

Le nombre de vues automatiquement modifiables a augmenté étant donné que les vues contenant des colonnes non modifiables (parce que calculées) et les vues avec l'option security_barrier sont enfin prises en compte.

De plus, l'option `WITH CHECK OPTION` permet d'avoir un comportement cohérent des insertions et mises à jour. Par exemple :

```
postgres=# CREATE VIEW v1 AS SELECT c1, upper(c2) FROM t1 WHERE c1<=10;
CREATE VIEW
postgres=# INSERT INTO v1 (c1) VALUES (-5);
INSERT 0 1
postgres=# INSERT INTO v1 (c1) VALUES (15);
INSERT 0 1
```

Ici, il est étonnant de pouvoir insérer dans la vue une valeur 15 alors qu'elle n'affiche que les valeurs inférieures ou égales à 10. Ceci ne serait pas possible avec une vue utilisant l'option `WITH CHECK OPTION` :

```
postgres=# CREATE VIEW v2 AS SELECT c1, upper(c2) FROM t1 WHERE c1<=10 WITH LOCAL CHECK OPTION;
CREATE VIEW
postgres=# INSERT INTO v2 (c1) VALUES (-5);
INSERT 0 1
postgres=# INSERT INTO v2 (c1) VALUES (15);
ERROR:  new row violates WITH CHECK OPTION for view "v2"
DETAIL:  Failing row contains (15, null)
```

5.2 JSON

- Nouveau type de données JSONB
  - stockage binaire
  - typage des éléments
  - puissant et performant
- Nombreuses nouvelles fonctions pour le type JSON

La grosse nouveauté est le nouveau type de données JSON appelé JSONB (B pour binaire). Le stockage des données est en binaire, ce qui a permis de respecter le typage des données. Ce nouveau type JSON est bien plus puissant et performant que l'ancien.

De nombreuses fonctions ont été ajoutées pour faciliter l'utilisation et la gestion de données de type JSON. En particulier, il est possible d'utiliser des index gin ou gist qui permettent d'accélérer les recherches quelle que soit la clé du document utilisée. Cette fonctionnalité n'a à notre connaissance aucun équivalent ni dans le monde relationnel ni dans le monde “NoSQL”.

Auparavant, le type json permettait d'indexer une clé particulière, à l'aide d'un index fonctionnel. Les requêtes pouvaient devenir pénibles à écrire. Le type JSONb

implémente le support pour les index gin. Deux familles d'opérateurs sont disponibles, supportant des opérateurs différents:

* jsonb_ops (par défaut). Supporte les opérateurs @> (contient), ? (contient une clé), et leurs variantes à base de tableaux. * jsonb_path_ops, qui ne supporte que l'opérateur @> mais est bien plus rapide, occupant moins de place sur disque.

Comparons rapidement ces types de données.

```
postgres=# CREATE TABLE j1 (document json);
CREATE TABLE
postgres=# CREATE TABLE j2 (document jsonb);
CREATE TABLE
postgres=# INSERT INTO j1 (document) (SELECT format('{"id": %s, "label": "Text %s"}', i, i)::json
FROM generate_series(1, 100000) i);
INSERT 0 100000
postgres=# INSERT INTO j2 (document) (SELECT format('{"id": %s, "label": "Text %s"}', i, i)::jsonb
FROM generate_series(1, 100000) i);
INSERT 0 100000
```

Les tables font alors 6,7 Mo (json) et 8,2 Mo (jsonb).

Pour faire des recherches sur j1, on peut passer par un index fonctionnel. En revanche, celui-ci ne fonctionnera que sur l'expression exacte:

```
postgres=# CREATE INDEX ON j1  ((document->>'id'));
CREATE INDEX
postgres=# EXPLAIN SELECT * FROM j1 WHERE document->>'id' = '3';
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan ON j1  (cost=12.17..774.92 ROWS=500 width=32)
   Recheck Cond: ((document ->> 'id'::text) = '3'::text)
   ->  Bitmap INDEX Scan ON j1_expr_idx  (cost=0.00..12.04 ROWS=500 width=0)
         INDEX Cond: ((document ->> 'id'::text) = '3'::text)
 Planning TIME: 0.197 ms
(5 ROWS)

postgres=# EXPLAIN SELECT * FROM j1 WHERE document->>'label' = '3';
                      QUERY PLAN
------------------------------------------------------
 Seq Scan ON j1  (cost=0.00..2334.00 ROWS=500 width=32)
   FILTER: ((document ->> 'label'::text) = '3'::text)
 Planning TIME: 0.042 ms
(3 ROWS)
```

Pour le type jsonb en revanche, on peut créer un index de type gin:

```
CREATE INDEX ON j2 USING gin(document);
```

Cet index est parfaitement capable de répondre à des requêtes complexes:

```
postgres=# EXPLAIN SELECT * FROM j2 WHERE document @> '{"id": 3}';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan ON j2  (cost=28.77..325.76 ROWS=100 width=49)
   Recheck Cond: (document @> '{"id": 3}'::jsonb)
   ->  Bitmap INDEX Scan ON j2_document_idx  (cost=0.00..28.75 ROWS=100 width=0)
         INDEX Cond: (document @> '{"id": 3}'::jsonb)
 Planning TIME: 0.057 ms
(5 ROWS)
```

```
postgres=# EXPLAIN SELECT * FROM j2 WHERE document @> '{"id": 3, "label": "Text"}';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan ON j2  (cost=52.77..349.76 ROWS=100 width=49)
   Recheck Cond: (document @> '{"id": 3, "label": "Text"}'::jsonb)
   ->  Bitmap INDEX Scan ON j2_document_idx  (cost=0.00..52.75 ROWS=100 width=0)
         INDEX Cond: (document @> '{"id": 3, "label": "Text"}'::jsonb)
 Planning TIME: 0.065 ms
(5 ROWS)
```

5.3 Clause WITH ORDINALITY

- À utiliser avec les fonctions SRF

```
postgres=# SELECT * FROM unnest('{un,deux,trois}'::text[])
  WITH ORDINALITY;
 unnest | ordinality
--------+------------
 un     |          1
 deux   |          2
 trois  |          3
(3 rows)
```

Cette clause permet d'ajouter une colonne numérotant les résultats.

Cela fonctionne avec toutes les fonctions SRF, comme par exemple generate_series :

```
postgres=# SELECT * FROM generate_series('2014-01-01'::timestamp, '2014-12-01'::timestamp, '1
month'::interval) with ordinality;
   generate_series    | ordinality
----------------------+------------
 2014-01-01 00:00:00 |          1
 2014-02-01 00:00:00 |          2
 2014-03-01 00:00:00 |          3
 2014-04-01 00:00:00 |          4
 2014-05-01 00:00:00 |          5
 2014-06-01 00:00:00 |          6
 2014-07-01 00:00:00 |          7
 2014-08-01 00:00:00 |          8
 2014-09-01 00:00:00 |          9
 2014-10-01 00:00:00 |         10
 2014-11-01 00:00:00 |         11
 2014-12-01 00:00:00 |         12
(12 rows)
```

Ceci est extrêmement pratique si l'on souhaite travailler de manière ensembliste sur des tableaux, sans perdre leur tri original.

5.4 Clause FILTER

- Nouvelle clause pour les agrégats
- Permet de filtrer les lignes prises en compte par la fonction d'agrégat
- Exemple

```
postgres=# SELECT count(*) count_tous,
  count(*) FILTER(WHERE bid=1) count_1,
  count(*) FILTER(WHERE bid=2) count_2
  FROM pgbench_history;
 count_tous | count_1 | count_2
------------+---------+---------
       7914 |     758 |     784
(1 row)
```

Cela a notamment pour intérêt d'éviter l'écriture de requêtes complexes comprenant des sous-requêtes, ou des instructions CASE. En effet, lorsque l'on souhaitait faire des totaux partiels, avant l'introduction de cette clause nous avions deux possibilités:

* la sous-requête. Cette méthode est extrêmement inefficace car elle nécessite généralement de parcourir plusieurs fois la table. Par exemple:

```
SELECT COUNT(*) count_tous,
(SELECT COUNT(*) FROM pgbench_history WHERE bid = 1) AS count_1,
(SELECT COUNT(*) FROM pgbench_history WHERE bid = 2) AS count_2
FROM pgbench_history;
```

* l'expression case. Celle-ci peut être assez efficace, mais est extrêmement pénible à lire et à écrire, et peut facilement générer des bugs :

```
SELECT COUNT(*) AS count_tous,
SUM(CASE WHEN bid = 1 THEN ELSE 0 END) AS count_1,
SUM(CASE WHEN bid = 2 THEN ELSE 0 END) AS count_2
FROM pgbench_history
```

L'expression FILTER permet d'écrire la requête de manière plus naturelle, et ouvre la porte à des optimisations pour le moteur.

5.5 Clause WITHIN GROUP

- Nouvelle clause pour les agrégats
- Meilleur support de la norme SQL 2008
- Nouvelles fonctions
  - percentile_cont(), percentile_disc()
  - rank(), dense_rank(), percent_rank()
  - cume_dist()
  - mode()
- SELECT mode() WITHIN GROUP (ORDER BY c1) AS plus_frequent FROM t1;

La clause `WITHIN GROUP` est une nouvelle clause pour les agrégats utilisant des fonctions dont les données doivent être triées. Quelques fonctions ont été ajoutées pour profiter au mieux de cette nouvelle clause.

Prenons l'exemple de la fonction `percentile_disc`. Celle-ci permet de renvoyer l'élément situé au plus près d'un certain percentile. Par exemple, pour afficher la médiane:

```
postgres=# SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY i) FROM (VALUES (1), (2),(2), (3),
(3) ) AS t(i);
 percentile_disc
-----------------
               2
(1 ROW)
```

En ajoutant le support de cette clause, PostgreSQL améliore son support de la norme SQL 2008 et permet le développement d'analyses statistiques plus poussées.

5.6 Clause ROWS FROM()

- ROWS FROM() permet de concaténer horizontalement les résultats d'appels de fonctions

```
postgres=# SELECT * FROM ROWS
           FROM (generate_series(1, 5),
                 generate_series(2, 6));
 generate_series | generate_series
-----------------+-----------------
               1 |               2
               2 |               3
               3 |               4
               4 |               5
               5 |               6
(5 rows)
```

- Fonction UNNEST()

Cette clause permettra de réaliser plus facilement des opérations sur les fonctions retournant des tables.

De plus, UNNEST accepte maintenant un nombre arbitraire d'arguments pour associer les éléments de même index de deux tableaux différents. C'est l'équivalent d'une convolution des tableaux donnés en entrée.

Exemple:

```
postgres=# SELECT * FROM unnest(ARRAY[1, 2], ARRAY['Couleur 1', 'Couleur 2']);
 unnest |  unnest
--------+-----------
      1 | Couleur 1
      2 | Couleur 2
(2 rows)
```

5.7 Triggers sur tables distantes

- SQL/MED
- Ajout du support des triggers sur les tables distantes
  - audit des DML sur tables distantes

Il est maintenant possible d'ajouter des triggers sur les tables distantes. Cela peut être intéressant, notamment pour ajouter un audit de ces tables, ou pour lancer une notification à l'insertion dans ces tables (mécanismes LISTEN/NOTIFY) et déclencher un traitement externe.

Aussi bien les triggers de niveau STATEMENT que ceux de niveau ROW sont supportés:

```
postgres=# create extension postgres_fdw ;
CREATE EXTENSION
postgres=# create server local_server foreign data wrapper postgres_fdw ;
CREATE SERVER
postgres=# create user mapping for dalibo server local_server ;
CREATE USER MAPPING
postgres=# create table l1 as (select generate_series(1, 10) as c1);
SELECT 10
postgres=# create foreign table ft1 (c1 int) SERVER local_server OPTIONS (table_name 'l1');
CREATE FOREIGN TABLE
postgres=# create or replace function trigger_func() RETURNS TRIGGER AS $$
BEGIN
RAISE NOTICE 'Je suis un trigger % de niveau %', TG_WHEN, TG_LEVEL;
RETURN NEW; END
$$ language plpgsql;
CREATE FUNCTION
postgres=# CREATE TRIGGER before_statement BEFORE INSERT OR UPDATE OR DELETE ON ft1 FOR EACH
STATEMENT execute procedure trigger_func();
CREATE TRIGGER
postgres=# CREATE TRIGGER after_statement AFTER INSERT OR UPDATE OR DELETE ON ft1 FOR EACH
STATEMENT execute procedure trigger_func();
CREATE TRIGGER
postgres=# CREATE TRIGGER before_row BEFORE INSERT OR UPDATE OR DELETE ON ft1 FOR EACH ROW execute
procedure trigger_func();
CREATE TRIGGER
postgres=# CREATE TRIGGER after_row AFTER INSERT OR UPDATE OR DELETE ON ft1 FOR EACH ROW execute
procedure trigger_func();
CREATE TRIGGER
postgres=# update ft1 set c1 = 2;
NOTICE:  Je suis un trigger BEFORE de niveau STATEMENT
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger BEFORE de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau ROW
NOTICE:  Je suis un trigger AFTER de niveau STATEMENT
UPDATE 10
```

Il existe une petite subtilité sur ces triggers: en effet, les triggers AFTER de niveau ROW ont besoin d'accéder à l'ancien et au nouveau tuple. Pour cela, les tuples en questions sont mis en cache localement, en utilisant le mécanisme habituel de `work_mem`: les tuples seront stockés en mémoire jusqu'à ce que cela dépasse `work_mem`. Une fois cette taille dépassée, ceux-ci déborderont sur le disque. Une modification sur une table étrangère possédant des triggers AFTER peut donc vite devenir très lente, des accès disques étant alors générés.

6 Administration

- Gestion des tablespaces
- Configuration à distance
- Configuration : nouveaux paramètres, nouvelles valeurs par défaut
- Processus en tâche de fond

La facilité d'administration n'a pas été oubliée. Il y a beaucoup de nouveautés sur les tablespaces ainsi que sur la configuration.

6.1 CREATE TABLESPACE

- Options disponibles sur ALTER TABLESPACE enfin disponible à la création
- Évite l'exécution de deux commandes

```
CREATE TABLESPACE ts1 LOCATION '/mnt/ts1'
  WITH (random_page_cost=1);
```

Auparavant, pour créer et configurer un tablespace, il fallait exécuter deux commandes :

- `CREATE TABLESPACE` pour le créer ;
- `ALTER TABLESPACE` pour le configurer.

La configuration réalisée par `ALTER TABLESPACE` est maintenant possible directement à partir du `CREATE TABLESPACE`.

6.2 Déplacement d'objets

- Déplacer les objets d'un certain type d'un tablespace vers un autre
- Types possibles
  - tables
  - index
  - vues matérialisées
- Filtre possible par propriétaire

```
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

Avant la version 9.4, il fallait déplacer les objets un par un. La version 9.4 apporte la clause `ALL` aux instructions `ALTER TABLE`, `ALTER INDEX` et `ALTER MATERIALIZED VIEW`. Elle permet de déplacer toutes les tables, tous les index ou toutes les vues matérialisées d'un tablespace vers un autre. Il est aussi possible de filtrer par le propriétaire des objets.

6.3 Verrous sur la commande ALTER TABLE

- Réduction du niveau de verrou pour certaines commandes ALTER TABLE
  - moins de contention, plus de performances
- Clauses
  - VALIDATE CONSTRAINT
  - CLUSTER ON
  - SET WITHOUT CLUSTER
  - ALTER COLUMN SET STATISTICS
  - ALTER COLUMN SET (attribute_option)
  - ALTER COLUMN RESET

Diminuer le niveau des verrous de certaines commandes permet de diminuer la contention et, de ce fait, d'améliorer les performances.

6.4 Configuration à distance

- Avant la 9.4, impossible nativement de faire de la configuration à distance
  - seule possibilité, adminpack avec pgAdmin
- Avec la 9.4
  - ALTER SYSTEM
- Modifie le fichier postgresql.auto.conf, pas le fichier postgresql.conf
- Ne pas modifier postgresql.auto.conf manuellement
  - tout commentaire ajouté sera perdu
- Nécessite un reload ou un restart selon le paramètre modifié

Voici un exemple de son utilisation. Au départ, le fichier ne contient que deux lignes de commentaires :

```
$ cat postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
```

L'utilisation de la nouvelle instruction va modifier le fichier :

```
$ psql postgres
psql (9.4beta1)
Type "help" for help.

postgres=# ALTER SYSTEM SET shared_buffers TO '1GB';
ALTER SYSTEM
postgres=# show shared_buffers;
 shared_buffers
----------------
 128MB
(1 row)

postgres=# \q
$ cat postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
shared_buffers = '1GB'
```

Le fichier est réécrit à chaque fois. En dehors des deux premières lignes de commentaires, tous les autres commentaires sont ignorés. Chaque paramètre est repris avec son ancienne configuration, sauf celle qui est la cible du ALTER SYSTEM.

6.5 Nouveau paramètre pour l'autovacuum

- Paramètre autovacuum_work_mem
- Surcharge la valeur de maintenance_work_mem pour l'autovacuum

Ce nouveau paramètre `autovacuum_work_mem` va être utilisé que par le processus `autovacuum` pour surcharger la valeur du paramètre `maintenance_work_mem`.

6.6 Formatage plus aisé du préfixe des traces

- log_line_prefix gagne des options de formatage à-la-printf
- Exemple:
  - log_line_prefix='%t %10d %10u %-5p '

Les informations peuvent être alignées à gauche ou à droite en donnant une valeur numérique (respectivement positive ou négative) après le % et avant l'option. Cela a pour but de rendre les traces plus lisibles.

6.7 Nouvelles valeurs par défaut

- Trois paramètres ont leur valeur par défaut augmentée
- work_mem : 1MB –> 4 MB
- maintenance_work_mem : 16MB –> 64 MB
- effective_cache_size : 128MB –> 4GB

Il faut néanmoins garder à l'esprit que, malgré cela, une revue des valeurs par défaut des paramètres est toujours nécessaire.

6.8 Nouvelle cible de restauration

- Option PITR (fichier recovery.conf)
- Jusqu'ici, plusieurs stratégies
  - par défaut : rejouer tous les WALs disponibles
  - time : rejeu jusqu'à une date
  - xid : rejeu jusqu'à une transaction
  - name: rejeu jusqu'à un point de restauration
- Nouvelle option
  - immediate
  - rejeu jusqu'au point de cohérence le plus proche

Voici un exemple complet de restauration utilisant la nouvelle option `immediate` :

```
[guillaume@localhost ~]$ pginit 94 r94
Switching to version 94
postgres (PostgreSQL) 9.4beta1
Creating cluster in /home/guillaume/var/lib/postgres/r94
The files belonging to this database system will be owned by user "guillaume".
This user must also own the server process.

The database cluster will be initialized with locale "C".
The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory /home/guillaume/var/lib/postgres/r94 ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
creating template1 database in /home/guillaume/var/lib/postgres/r94/base/1 ... ok
initializing pg_authid ... ok
initializing dependencies ... ok
creating system views ... ok
loading system objects' descriptions ... ok
creating collations ... ok
creating conversions ... ok
creating dictionaries ... ok
setting privileges on built-in objects ... ok
creating information schema ... ok
loading PL/pgSQL server-side language ... ok
vacuuming database template1 ... ok
copying template1 to template0 ... ok
copying template1 to postgres ... ok
syncing data to disk ... ok

WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    postgres -D /home/guillaume/var/lib/postgres/r94
```

```
or
    pg_ctl -D /home/guillaume/var/lib/postgres/r94 -l logfile start

Set parameter : "port=5438"
[guillaume@localhost ~]$ pgstart r94
Switching to version 94
postgres (PostgreSQL) 9.4beta1
Attempting to start r94: [1] 9063
Waiting.
Cluster r94 started.
[guillaume@localhost ~]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# ALTER SYSTEM SET wal_level TO 'archive';
ALTER SYSTEM
postgres=# ALTER SYSTEM SET archive_mode TO on;
ALTER SYSTEM
postgres=# ALTER SYSTEM SET archive_command TO 'cp %p /opt/postgresql/archives/r94/%f';
ALTER SYSTEM
postgres=# \q
[guillaume@localhost ~]$ sudo mkdir -p /opt/postgresql/archives/r94
[sudo] password for guillaume:
[guillaume@localhost ~]$ sudo chown guillaume:guillaume /opt/postgresql/archives/r94
[guillaume@localhost ~]$ pgrestart r94
Switching to version 94
postgres (PostgreSQL) 9.4beta1
waiting for server to shut down.... done
server stopped
Switching to version 94
postgres (PostgreSQL) 9.4beta1
Attempting to start r94: [1] 9140
Waiting.
Cluster r94 started.
[guillaume@localhost ~]$ createdb benchs
[guillaume@localhost ~]$ pgbench -i -s10 benchs
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 1000000 tuples (10%) done (elapsed 0.08 s, remaining 0.70 s).
200000 of 1000000 tuples (20%) done (elapsed 0.56 s, remaining 2.24 s).
300000 of 1000000 tuples (30%) done (elapsed 1.13 s, remaining 2.64 s).
400000 of 1000000 tuples (40%) done (elapsed 1.42 s, remaining 2.12 s).
500000 of 1000000 tuples (50%) done (elapsed 1.82 s, remaining 1.82 s).
600000 of 1000000 tuples (60%) done (elapsed 2.46 s, remaining 1.64 s).
700000 of 1000000 tuples (70%) done (elapsed 2.83 s, remaining 1.21 s).
800000 of 1000000 tuples (80%) done (elapsed 3.82 s, remaining 0.95 s).
900000 of 1000000 tuples (90%) done (elapsed 5.17 s, remaining 0.57 s).
1000000 of 1000000 tuples (100%) done (elapsed 5.27 s, remaining 0.00 s).
vacuum...
set primary keys...
done.
[guillaume@localhost ~]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# \x
Expanded display (expanded) is on.
postgres=# select * from pg_stat_archiver ;
-[ RECORD 1 ]------+------------------------------
 archived_count     | 8
 last_archived_wal  | 000000010000000000000008
 last_archived_time | 2014-06-19 18:20:12.167006+02
 failed_count       | 0
 last_failed_wal    |
 last_failed_time   |
 stats_reset        | 2014-06-19 18:17:44.585545+02

postgres=# \q
```

```
[guillaume@localhost ~]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# SELECT pg_start_backup('test 9.4', true);
 pg_start_backup
-----------------
 0/A000028
(1 row)

postgres=# \q
[guillaume@localhost ~]$ cd var/lib/postgres/
[guillaume@localhost postgres]$ cp -r r94 r94_2
[guillaume@localhost postgres]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# CREATE TABLE t1(id integer);
CREATE TABLE
postgres=# \q
[guillaume@localhost postgres]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# SELECT pg_stop_backup();
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
 pg_stop_backup
----------------
 0/A00FAF8
(1 row)

postgres=# CREATE TABLE t2(id integer);
CREATE TABLE
postgres=# \q
[guillaume@localhost postgres]$ pgbench -c 10 -T 60 benchs
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
duration: 60 s
number of transactions actually processed: 10755
latency average: 55.788 ms
tps = 179.058065 (including connections establishing)
tps = 179.089032 (excluding connections establishing)
[guillaume@localhost postgres]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# CREATE TABLE t3(id integer);
CREATE TABLE
postgres=# \q
```

Au niveau du répertoire de copie, le fichier `postgresql.conf` a été modifié pour changer le numéro de port et le fichier `recovery.conf` a été modifié ainsi :

```
restore_command = 'cp /opt/postgresql/archives/r94/%f %p'
recovery_target = 'immediate'
```

Puis le deuxième serveur a été démarré :

```
[guillaume@localhost postgres]$ cat postgresql-r94_2.log
LOG:  database system was interrupted; last known up at 2014-06-19 18:21:27 CEST
LOG:  starting point-in-time recovery to earliest consistent point
LOG:  restored log file "000000010000000000000A" from archive
LOG:  redo starts at 0/A000090
LOG:  consistent recovery state reached at 0/A00FAF8
```

```
LOG:  recovery stopping after reaching consistency
LOG:  redo done at 0/A00FAD0
LOG:  last completed transaction was at log time 2014-06-19 18:22:19.085523+02
cp: cannot stat '/opt/postgresql/archives/r94/00000002.history': No such file or directory
LOG:  selected new timeline ID: 2
cp: cannot stat '/opt/postgresql/archives/r94/00000001.history': No such file or directory
LOG:  archive recovery complete
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections

[guillaume@localhost postgres]$ psql
psql (9.4beta1)
Type "help" for help.

postgres=# \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | t1   | table | postgres
(1 row)
```

Seules les données modifiées avant et pendant la sauvegarde ont été récupérées. Toutes les modifications après la sauvegarde ayant généré des journaux de transactions n'ont pas été prises en compte.

6.9 Processus en tâche de fond

- Paramètre max_worker_processes
- Background worker apparu en 9.3
- Enregistrement dynamique
  - démarrage et arrêt à tout moment
- Allocation dynamique de mémoire partagée

La version 9.3 introduisait le concept de `background worker`, autrement dit des processus utilisateurs en tâche de fond gérés par le démon postgres. Ces processus étaient automatiquement démarrés au lancement de PostgreSQL et arrêtés à l'arrêt de PostgreSQL. En cas de crash de ce processus, PostgreSQL le redémarrait aussitôt.

La version 9.4 améliore les possibilités de ces processus en tâche de fond avec la possibilité d'être démarré ou arrêté à tout moment, ainsi que la possibilité de récupérer dynamiquement de la mémoire partagée.

Enfin, le paramètre `max_worker_processes` a été introduit pour éviter de démarrer trop de background workers.

Cette infrastructure a été mis en place dans le cadre du travail de EnterpriseDB pour le parallélisme dans PostgreSQL.

6.10 Divers

- Possibilité de changer le fait qu'une contrainte soit DEFERRABLE
- Comportement de DROP IF EXISTS plus cohérent
- Nouvelle option –if-exists pour pg_restore
- Options multiple de pg_restore: I, -P, -T and -n
- Nouvelle option –tablespace-mapping pour pg_base_backup

Auparavant, il était impossible de changer le fait que l'évaluation d'une contrainte puisse être reportée à la fin de la transaction (attribut DEFERRABLE). Il fallait donc supprimer et recréer la contrainte. Il est maintenant possible de le faire directement:

```
postgres=# ALTER TABLE t2 ALTER CONSTRAINT t2_id_fkey DEFERRABLE;
ALTER TABLE
```

`DROP IF EXISTS` pouvait avoir un comportement inattendu. Par exemple, essayer de supprimer une fonction dans un schéma qui n'existe pas:

En 9.3:

```
postgres=# DROP FUNCTION IF EXISTS schema.fonction() ;
ERREUR:  le schéma « schema » n'existe pas
```

En 9.4:

```
postgres=# DROP FUNCTION IF EXISTS schema.fonction() ;
NOTICE:  schema "schema" does not exist, skipping
DROP FUNCTION
```

Cela permet d'implémenter l'option `–if-exists` de pg_restore. Celle-ci ne lèvera pas d'erreurs pour les objets n'existant pas lorsque l'option –clean est utilisée.

`pg_restore` permet maintenant de combiner les options sélectionnant spécifiquement des objets à restaurer. Cela rend son comportement plus cohérent avec celui de pg_dump.

L'option `–tablespace-mapping` de pg_basebackup permet de ne pas restaurer les tablespaces au même endroit que sur le serveur copié. On peut donc faire un basebackup sur le même serveur, même si l'on utilise des tablespaces.

7 Supervision

- Nouvelle vue pg_stat_archiver
- Nouvelles colonnes dans...
  - pg_stat_activity
  - pg_stat_replication
  - pg_stat_all_tables

La supervision est un pan essentiel des SGBD. PostgreSQL n'y fait pas exception et propose de nombreuses vues statistiques permettant de connaître l'activité du moteur par objets : bases, tables, index, etc. Cette nouvelle version propose des nouvelles informations.

7.1 Nouvelle vue pg_stat_archiver

- Statistiques sur l'archivage
- Une seule ligne, 6 informations statistiques
  - nombre de tentatives réussies
  - dernier journal archivé
  - horodatage du dernier archivage réussi
  - nombre de tentatives échouées
  - dernier journal dont l'archivage a échoué
  - horodatage du dernier archivage échoué

Voici un exemple de contenu de cette vue :

```
postgres=# SELECT * FROM pg_stat_archiver;
-[ RECORD 1 ]------+------------------------------
 archived_count     | 141
 last_archived_wal  | 000000010000000000000024
 last_archived_time | 2014-06-17 11:32:01.94076+02
 failed_count       | 276
 last_failed_wal    | 000000010000000000000025
 last_failed_time   | 2014-06-17 13:06:04.876071+02
 stats_reset        | 2014-06-17 10:11:27.299133+02
```

7.2 Compteur de lignes modifiées

- Colonne n_mod_since_analyze dans pg_stat_all_tables
- Indique le nombre de lignes écrites depuis le dernier ANALYZE
- Testé par l'autovacuum pour savoir quand déclencher un ANALYZE
- Passe à zéro après exécution d'un ANALYZE

```
postgres=# SELECT relname, n_mod_since_analyze
  FROM pg_stat_user_tables ;
     relname      | n_mod_since_analyze
------------------+---------------------
 pgbench_accounts |               26912
 pgbench_history  |                 774
 pgbench_branches |                   0
 pgbench_tellers  |                   0
(4 rows)
```

Cette information nous permet de prédire quand `autovacuum` va déclencher un `ANALYZE`, ou quand un `ANALYZE` serait nécessaire si l' `autovacuum` a été désactivé.

7.3 Identifiants de transaction

- Nouvelles colonnes backend_xid et backend_xmin pour la vue pg_stat_activity
- Nouvelle colonne backend_xmin pour la vue pg_stat_replication
- backend_xid : identifiant de transaction courant pour cette session
- backend_xmin : identifiant de transaction représentant la vision de la base pour cette session

Voici les deux nouvelles colonnes dans pg_stat_activity :

```
postgres=# SELECT * FROM pg_stat_activity ;
-[ RECORD 1 ]----+---------------------------------
 datid           | 13056
 datname         | postgres
 pid             | 28118
 usesysid        | 10
 usename         | postgres
 application_name | psql
 client_addr     |
 client_hostname |
 client_port     | -1
 backend_start   | 2014-06-17 10:56:33.404005+02
 xact_start      | 2014-06-17 11:21:38.466331+02
 query_start     | 2014-06-17 11:21:38.466331+02
 state_change    | 2014-06-17 11:21:38.466335+02
 waiting         | f
```

```
state           | active
backend_xid     |
backend_xmin    | 28841
query           | SELECT * FROM pg_stat_activity ;
```

8 Performances

8.1 Nouvel outil pg_prewarm

- Nouvelle extension en contrib
- Charge une relation en mémoire
- Cinq arguments
  - nom de la relation
  - mode de préchauffage (prefetch, read, buffer)
  - type de fichier (main, fsm, vm)
  - premier et dernier blocs à préchauffer

```
SELECT pg_prewarm('pgbench_history');
```

Après le démarrage de PostgreSQL, le cache est vide (froid). Jusqu'à la version 9.4, il fallait attendre que des requêtes soient exécutées pour que les blocs des relations lues soient chargés en mémoire dans le cache. Cette nouvelle extension permet de pré-charger les objets voulus. Il existe trois méthodes de préchauffage.

- `buffer` lit les données depuis le système de fichiers, et les cache dans les `shared_buffers`
- `prefetch` et `read` ne servent qu'à chauffer le cache du système de fichiers: le cache PostgreSQL (shared_buffers) ne sera pas rempli avec cette méthode. La différence entre les deux concernent le types d'appels systèmes utilisés: `prefetch` lancera des demande de prefetch asynchrones au système, là où `read` effectue des opérations de lectures synchrone. `prefetch` est plus rapide, mais n'est pas supporté sur toutes les plateformes.

```
postgres=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
postgres=# SELECT pg_prewarm('pgbench_history');
 pg_prewarm
------------
        189
(1 row)

postgres=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM pgbench_history;
                                                    QUERY PLAN
```

```
------------------------------------------------------------------------------------------------------------------
 Seq Scan on pgbench_history  (cost=0.00..465.85 rows=27685 width=116) (actual time=0.014..4.000
rows=26912 loops=1)
   Buffers: shared hit=189
 Planning time: 0.534 ms
 Execution time: 6.172 ms
(4 rows)
```

Attention, il ne s'agit pas d'un outil de sauvegarde et restauration du cache !

8.2 Meilleure compression des WAL

- pg_switch_xlog nettoie la place non utilisée dans le journal
- Meilleure compression possible des journaux
- Moins d'écriture, plus de performances

`pg_switch_xlog()` nettoie la place non utilisée dans le journal. L'intérêt est de pouvoir avoir une meilleure compression des journaux avec des outils comme `gzip` ou `bzip2`.

8.3 Nouveautés du EXPLAIN

- Ajout de la durée de planification

```
QUERY PLAN
-----------------------------------------------------------
 Hash Join  (cost=5.22..9.60 rows=100 width=352) [...]
         ->  Seq Scan on pgbench_branches b [...]
 Planning time: 0.275 ms
 Execution time: 0.161 ms
```

- Affichage des colonnes de regroupage

```
                              QUERY PLAN
------------------------------------------------------------------------
 HashAggregate  (cost=3.50..3.60 rows=10 width=4)
   Group Key: bid
   ->  Seq Scan on pgbench_tellers  (cost=0.00..3.00 rows=100 width=4)
 Planning time: 0.109 ms
```

Deux nouveautés majeures apparaissent avec la commande EXPLAIN. La première est de loin la plus intéressante. Dans la durée d'exécution d'une requête se trouvent mêlées la durée de planification et la durée d'exécution. Parfois, sur les grosses requêtes, la durée de planification a un rôle prépondérant. La version 9.4 nous permet d'accéder à cette durée en plus de la durée d'exécution :

```
postgres=# EXPLAIN ANALYZE SELECT t.* FROM pgbench_tellers t JOIN pgbench_branches b ON
b.bid=t.bid;
                                                          QUERY PLAN

------------------------------------------------------------------------------------------------------------
-----------------------
 Hash Join  (cost=5.22..9.60 rows=100 width=352) (actual time=0.054..0.113 rows=100 loops=1)
   Hash Cond: (t.bid = b.bid)
   ->  Seq Scan on pgbench_tellers t  (cost=0.00..3.00 rows=100 width=352) (actual
time=0.014..0.035 rows=100 loops=1)
   ->  Hash  (cost=5.10..5.10 rows=10 width=4) (actual time=0.023..0.023 rows=10 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on pgbench_branches b  (cost=0.00..5.10 rows=10 width=4) (actual
time=0.003..0.015 rows=10 loops=1)
 Planning time: 0.275 ms
 Execution time: 0.161 ms
(8 rows)
```

L'autre nouveauté est une information supplémentaire nous indiquant les colonnes de regroupage dans les nœuds Agg et Group. La nouvelle ligne a comme libellé « Group Key » :

```
postgres=# EXPLAIN SELECT bid, count(*) FROM pgbench_tellers GROUP BY bid;
                              QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=3.50..3.60 rows=10 width=4)
   Group Key: bid
   ->  Seq Scan on pgbench_tellers  (cost=0.00..3.00 rows=100 width=4)
 Planning time: 0.109 ms
(4 rows)
```

8.4 Améliorations des index GIN

- Optimisation des requêtes de type “rare & fréquent”
- Réduction de la taille des index

Pour mettre en évidence ces évolutions, jouons le même scénario sur une instance 9.3 et 9.4.

On crée une table avec une colonne de type tableau d'entier. Ce tableau aura une particularité, puisque toutes les lignes contiendront l'entier 1, et un autre entier qui ne sera présent nulle part, présent plusieurs fois. Cette table se voit dotée d'un index gin:

```
postgres=# CREATE TABLE for_gin AS (SELECT ARRAY[1, x, x, x, x, x, x, x, x, x, x, x, x] AS tableau
FROM generate_series(1, 100000) AS x);
SELECT 100000
postgres=# CREATE INDEX ON for_gin USING gin(tableau);
CREATE INDEX
postgres=# ANALYZE for_gin;
ANALYZE
```

On peut dores et déjà comparer la taille des index. En 9.3, l'index fait 6080 Ko pour 5600 Ko en 9.4. Les tables, elles, font 10 Mo dans les deux cas.

Maintenant, cherchons toutes lignes qui contiennent 1 et qui contiennent 20000:

En 9.3:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM for_gin WHERE tableau @> ARRAY[1, 20000];
                                                              QUERY PLAN

---------------------------------------------------------------------------------------------------------------
-----------------------------
 Bitmap Heap Scan on for_gin  (cost=31.88..1014.07 rows=500 width=73) (actual time=6.496..6.497
rows=1 loops=1)
   Recheck Cond: (tableau @> '{1,20000}'::integer[])
   ->  Bitmap Index Scan on for_gin_tableau_idx  (cost=0.00..31.75 rows=500 width=0) (actual
time=6.494..6.494 rows=1 loops=1)
         Index Cond: (tableau @> '{1,20000}'::integer[])
 Total runtime: 6.515 ms
(5 lignes)
```

En 9.4:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM for_gin WHERE tableau @> ARRAY[1, 20000];
                                                              QUERY PLAN

---------------------------------------------------------------------------------------------------------------
-----------------------------
 Bitmap Heap Scan ON for_gin  (cost=31.88..1014.07 ROWS=500 width=73) (actual TIME=0.112..0.113
ROWS=1 loops=1)
   Recheck Cond: (tableau @> '{1,20000}'::INTEGER[])
   Heap Blocks: exact=1
   ->  Bitmap INDEX Scan ON for_gin_tableau_idx  (cost=0.00..31.75 ROWS=500 width=0) (actual
TIME=0.106..0.106 ROWS=1 loops=1)
         INDEX Cond: (tableau @> '{1,20000}'::INTEGER[])
 Planning TIME: 0.081 ms
 Execution TIME: 0.134 ms
(7 ROWS)
```

La requête est 60 fois plus rapide en 9.4 ! Cette optimisation bénéficiera grandement aux index Full Text ou trigramme (extension pg_trgm). Le nouveau type jsonb profite aussi de cette optimisation.

8.5 Divers

- UPDATE et WAL
  - réduction de la taille des enregistrements dans les WAL pour les UPDATE
- Écritures parallèles et WAL
  - meilleure gestion d'écritures multiples dans les tampons WAL
- VACUUM FREEZE
  - xmin et xmax ne sont plus modifiés
- FREEZE et réécriture de tables
  - tentative de FREEZE des lignes quand les tables sont réécrites avec CLUSTER et VACUUM FULL
- Meilleures estimations sur la mémoire consommée par les opérations d'agrégation
- Support des huge_pages

9 Régressions / changements

- Suppression du support de l'authentification krb5
  - utiliser GSSAPI !
- Changement -u en -U sur pg_upgrade
- DISCARD ALL réinitialise le statut des séquences
- Gestion des tableaux, du json
- Interdiction de référencer des colonnes systèmes dans les contraintes CHECK

Cette nouvelle version propose des changements qui peuvent entraîner des régressions dans vos applications.

Tous ces changements sont documentés sur cette page:
http://www.postgresql.org/docs/9.4/static/release-9-4.html#AEN118470

Parmi les plus marquants:

- L'authentification krb5 a disparu. Il convient d'utiliser GSSAPI à la place.
- L'option -u de pg_upgrade n'était pas cohérente avec les autres outils, qui utilisent

-U pour l'utilisateur. C'est désormais corrigé, mais il faudra veiller à mettre à jour vos scripts de migrations si vous utilisez pg_upgrade.

- DISCARD ALL réinitialise aussi le statut des séquences. Les utilisateurs d'un pooler de connexion sont les plus susceptibles d'être impactés. Exemple:

```
postgres=# SELECT nextval('ma_seq'::regclass);
 nextval
---------
       1
(1 row)

postgres=# SELECT currval('ma_seq'::regclass);
 currval
---------
       1
(1 row)

postgres=# DISCARD ALL;
DISCARD ALL
postgres=# SELECT currval('ma_seq'::regclass);
ERROR:  currval of sequence "ma_seq" is not yet defined in this session
```

- Le parser de tableau est plus strict, et la gestion des tableaux vides harmonisée: ce sont désormais des tableaux à zéro dimension partout.
- Le format json subit quelques évolutions: les dates sont au format ISO 8601, et les caractères unicode ne sont plus échappés à l'aide de backslash.
- Les contraintes CHECK ne sont plus autorisées à utiliser les colonnes systèmes, excepté tableoid. Tableoid est certainement la colonne la plus utilisée sur ce type de contraintes check, pour s'assurer qu'un système d'héritage ou de partitionnement insère dans les bonnes tables.

10 Conclusion

- JSONb permet de concurrencer les bases orientées documents
- Plein de petites nouveautés
  - qui facilitent la vie
- Des performances
- Et de belles bases pour le futur:
  - parallélisation
  - gros volumes de données
  - réplication logique

Le nouveau type de donnée JSONb permet de concurrent les bases orientées documents. Il permet de profiter du meilleur des deux mondes, en stockant une partie

des informations de manière dénormalisée, sous forme de document, tout en gardant la cohérence offerte par le modèle relationnel.

En parallèle, nous avons un très grand nombre de petites fonctionnalités, souvent attendues, qui nous facilitent bien la vie.

Le développement a repris de plus belle en même temps que les tests de cette version beta… On peut s'attendre à de belles innovations reposant sur les briques mises en place dans cette version: réplication logique, bi-directionnelle, parallélisation…