DALIBO

L'Expertise PostgreSQL

Quoi de neuf avec PostgreSQL 9.0 ?

BY NC SA

1 À propos des auteurs...

» Auteurs : Marc Cousin & Damien Clochard
» Société : DALIBO
» Date : Septembre 2010
» URL : https://support.dalibo.com/kb/conferences/postgresql_9.0/

2 Licence

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

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

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

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

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

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

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

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

3 Au menu

- **Hot Standby + Streaming Replication**
- Nouveautés
- Améliorations
- Régressions potentielles

Ce document tente de présenter les principaux changements apportés par PostgreSQL 9.0, par rapport à la version majeure précédente, la version 8.4. Dans la mesure du possible, chaque fonctionnalité sera expliquée et accompagnée d'une démonstration. Toutes les nouveautés ne sont bien sûr pas présentées (il y en a plus de 200).

La version 9.0, comme son nom l'indique, est une version capitale dans la progression de PostgreSQL. Même si les solutions de réplication pour PostgreSQL sont nombreuses et répondent à des problématiques variées, la version 9.0 apporte une réplication simple, robuste et intégrée au moteur, qui sera vraisemblablement utilisée par défaut dans la plupart des configurations de Haute Disponibilité reposant sur PostgreSQL.

Les changements ont été subdivisés en 3 parties: les nouveautés, les améliorations, les régressions potentielles….

4 L'évènement

Hot Standby
+ Streaming Replication
= Oracle Active Data Guard

La réplication est LA grande nouveauté de PostgreSQL !

Attendue depuis des années, ces deux nouveautés sont celles qui ont justifié à elles seules le renommage de la version 8.5 en 9.0.

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

5 Hot Standby

- Esclaves en lecture seule
- wal_level à 'hot_standby' sur le maître
- hot_standby à 'on' sur l'esclave

La configuration sur le maître se fait à un endroit : le postgresql.conf. Voici les trois paramètres minimum à modifier :

- wal_level = 'hot_standby'
- archive_mode = on
- archive_command = 'scp %p esclave:/repertoire/archivage/%f'

Il faut ensuite redémarrer le serveur maître.

Ensuite, il faut sauvegarder les fichiers avec les trois étapes habituelles :

- psql -c “SELECT pg_start_backup('label', true)” postgres
- tar cfj maitre.tar.bz2 /repertoire/des/donnees
- psql -c “SELECT pg_stop_backup();” postgres

La mise en place de l'esclave est identique à celle d'un esclave en Warm Standby. Pour la configuration du postgresql.conf, il faut en plus modifier la valeur du paramètre `hot_standby` pour que ce dernier soit à on. Il faut ensuite démarrer le serveur esclave.

À partir de ce moment-là, il sera possible de se connecter à l'esclave en lecture seule. Les requêtes de modification tomberont en erreur, et celles de lecture réussiront.

6 Streaming Replication

- Réplication en flux
- Sur le maître, 1 paramètrage au minimum dans le postgresql.conf
- Sur le maître, configuration du pg_hba.conf
- Sur l'esclave, 4 paramètrages au minimum dans le postgresql.conf

Il faut configurer sur le maître le nombre d'esclaves pouvant se connecter à ce maître. Il existe, pour cela, le paramètre max_wal_senders : `max_wal_senders = 1`.

Le processus de réception des transactions va se connecter du serveur esclave sur le serveur maître. Il faut donc autoriser les connexions de l'esclave sur le maître via une modification du fichier pg_hba.conf :

host replication all ip_esclave/32 trust

Ensuite, il est possible de redémarrer le maître.

Pour l'esclave, tout le travail se fera sur le fichier recovery.conf.

- restore_command = 'cp /repertoire/archivage/%f %p'
- standby_mode = 'on'
- primary_conninfo = 'host=ip_maitre port=5432'
- trigger_file = '/tmp/stopstandby'

Le serveur esclave peut être redémarré.

Une fois le serveur esclave redémarré, il va tenter de se connecter au maître. Si cela fonctionne, on trouvera dans les traces du maître la ligne suivante :

```
2010-08-21 11:02:49 CEST LOG:  replication connection authorized:
user=guillaume host=127.0.0.1 port=46031
```

CC BY NC SA

Quant aux traces sur l'esclave, elles indiquent la ligne suivante :

```
2010-08-21 11:02:49 CEST LOG:  streaming replication successfully connected to primary
```

Maintenant, toute modification sur le maître sera beaucoup plus rapidement disponible sur l'esclave.

7 Autres nouveautés

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

8 64 bits sous Windows

- Version 64 bits native pour Windows
- Utile pour les applications 64 bit (libpq.dll)
- Possibilité d'augmenter work_mem et maintenance_work_mem
- shared_buffers > 500 Mo ?

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

CC BY NC SA

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

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

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

9 Contraintes d'exclusion

- Au delà des contraintes d'unicité
- La contrainte associe un opérateur par colonne dans la contrainte
- Beaucoup d'opérateurs acceptés
- Index GiST associé

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

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

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

```
 professeur TEXT,
  periode    PERIOD
);

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

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

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

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

10 Triggers

- Trigger avec condition
  - sur une colonne : UPDATE OF colonne
  - sur une expression : WHEN

Voici d'abord un trigger par colonne.

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

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

CC BY NC SA

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

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

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

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

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

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

11 Contraintes UNIQUE différée

- Contraintes déferrables
  - PRIMARY KEY
  - UNIQUE
- Désactive la contrainte temporairement
- puis l'applique à la fin de la transaction
- INITIALLY DEFERRED… ou pas

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

BY NC SA

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

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

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

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

Avec PostgreSQL 9.0, nous pouvons maintenant faire ceci :

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

Ah zut, ça ne marche pas 😕

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

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

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

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

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

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

CC BY NC SA

12 Fonctions anonymes

- Exécuter un script à la volée
- Pas besoin de CREATE FUNCTION / DROP FUNCTION
- Utile pour réaliser des scripts ponctuels

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

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

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

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

13 Paramètres nommés

- SELECT test(b:='toto', a:=1);

CC BY NC SA

- Syntaxe plus lisible et plus pratique
- Moins de risques d'erreur

La syntaxe retenue pour nommer les paramètres est « := ». En voici un exemple :

```
CREATE FUNCTION test (a int, b text) RETURNS text AS $$
DECLARE
  valeur text;
BEGIN
  valeur := 'a vaut ' || a::text || ' et b vaut ' || b;
  RETURN valeur;
END;
$$ LANGUAGE plpgsql;
```

Jusque là, on écrivait :

```
SELECT test(1, 'toto');
         test
-------------------------
a vaut 1 et b vaut toto
(1 row)
```

Maintenant, on peut utiliser cette syntaxe explicite:

```
SELECT test(b:='toto', a:=1);
         test
-------------------------
a vaut 1 et b vaut toto
(1 row)
```

De nombreux langages permettent ce type de syntaxe d'appel de fonction, qui améliore fortement la lisibilité du code.

Par ailleurs, ceci devrait permettre d'éliminer beaucoup de fonctions d'encapsulage (« wrapper functions »). Toutefois cela provoque également une incompatibilité : vous ne pouvez plus renommer une fonction avec la commande REPLACE, il faut désormais supprimer puis recréer la fonction.

14 Gestion des droits

- Plus simple / Plus pratique !
- GRANT SELECT ON ALL TABLES IN SCHEMA
- ALTER DEFAULT PRIVILEGES

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

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

Et la marche arrière :

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

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

```
ALTER DEFAULT PRIVILEGES FOR ROLE marc GRANT SELECT ON TABLES TO PUBLIC ;
CREATE TABLE test_priv (a int);
\z test_priv
                            Access privileges
 Schema |    Name    | Type  | Access privileges | Column access privileges
--------+------------+-------+-------------------+--------------------------
 public | test_priv  | table | =r/marc          +|
        |            |       | marc=arwdDxt/marc |
```

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

15 Améliorations

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

16 Planificateur

- Join Removal
- Suppression des jointures inutiles
- Efficace face aux ORM (hibernate, doctrine, etc.)

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

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

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

```
marc=# EXPLAIN SELECT t1.a,t2.b FROM t1 JOIN t2 ON (t1.a=t2.b) LEFT JOIN t3 ON (t1.a=t3.c);
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Merge RIGHT JOIN  (cost=506.24..6146.24 rows=345600 width=8)
   Merge Cond: (t3.c = t1.a)
```

BY NC SA

```
  ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
        Sort KEY: t3.c
        ->  Seq Scan ON t3  (cost=0.00..34.00 rows=2400 width=4)
   ->  Materialize  (cost=337.49..853.49 rows=28800 width=8)
        ->  Merge JOIN  (cost=337.49..781.49 rows=28800 width=8)
              Merge Cond: (t1.a = t2.b)
              ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
                    Sort KEY: t1.a
                    ->  Seq Scan ON t1  (cost=0.00..34.00 rows=2400 width=4)
              ->  Sort  (cost=168.75..174.75 rows=2400 width=4)
                    Sort KEY: t2.b
                    ->  Seq Scan ON t2  (cost=0.00..34.00 rows=2400 width=4)
```

On a le même comportement qu'en 8.4, c'est normal. Mais imaginons que sur la table t3, on ait une contrainte UNIQUE sur la colonne c. Dans ce cas, théoriquement, la jointure sur la table t3 ne sert à rien : le nombre d'enregistrements du résultat ne sera pas modifié, pas plus, bien sûr, que leur contenu. C'est lié au fait que la colonne est UNIQUE, que la jointure est un LEFT JOIN et qu'aucune colonne de t3 n'est récupérée. Si la colonne n'était pas UNIQUE, la jointure pourrait augmenter le nombre d'enregistrements du résultat. Si ce n'était pas un LEFT JOIN, la jointure pourrait diminuer le nombre d'enregistrements du résultat.

Avec PostgreSQL 9.0 :

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

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

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

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

17 Index

- IS NOT NULL utilise les index
- Utilisation d'index pour générer des statistiques à la volée

Pour la démonstration du IS NOT NULL, nous allons comparer la version 8.4 et la 9.0. La table créée contient majoritairement des valeurs NULL.

Pour la version 8.4 :

```
marc=# EXPLAIN ANALYZE SELECT max(a) FROM test;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Result  (cost=0.03..0.04 rows=1 width=0) (actual time=281.320..281.321 rows=1 loops=1)
InitPlan 1 (returns DOKUWIKI-ODT-INSERT)
  ->  LIMIT  (cost=0.00..0.03 rows=1 width=4) (actual time=281.311..281.313 rows=1 loops=1)
  ->  INDEX Scan Backward USING idxa ON test  (cost=0.00..29447.36 rows=1001000 width=4) (actual
time=281.307..281.307 rows=1 loops=1)
Filter: (a IS NOT NULL)
Total runtime: 281.360 ms
(6 rows)
```

Avec PostgreSQL 9.0 :

```
marc=# EXPLAIN ANALYZE SELECT max(a) FROM test;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Result  (cost=0.08..0.09 rows=1 width=0) (actual time=0.100..0.102 rows=1 loops=1)
InitPlan 1 (returns DOKUWIKI-ODT-INSERT)
  ->  LIMIT  (cost=0.00..0.08 rows=1 width=4) (actual time=0.092..0.093 rows=1 loops=1)
  ->  INDEX Scan Backward USING idxa ON test  (cost=0.00..84148.06 rows=1001164 width=4) (actual
time=0.089..0.089 rows=1 loops=1)
      INDEX Cond: (a IS NOT NULL)
Total runtime: 0.139 ms
(6 rows)
```

On constate que la version 9.0 parcourt uniquement les clés non nulles de l'index (Index cond, au lieu d'un filtre à posteriori). Dans ce cas précis, le gain est très net.

Observons maintenant l'utilisation des index pour produire des statistiques à la volée.

Avant de commencer à expliquer la nouveauté, un petit rappel sur les histogrammes: PostgreSQL, comme d'autres moteurs de bases de données, utilise un optimiseur statistique. Cela signifie qu'au moment de la planification d'une requête il a (ou devrait) avoir une idée correcte de ce que chaque étape de la requête va lui ramener, en termes de nombre d'enregistrements. Pour cela, il utilise des statistiques, comme le nombre approximatif d'enregistrements de la table, sa taille, la corrélation physique entre valeurs voisines dans la table, les valeurs les plus fréquentes et les histogrammes, qui permettent d'évaluer assez précisément le nombre d'enregistrements ramenés par une clause WHERE sur une colonne, suivant la valeur ou la plage demandée sur la clause WHERE.

CC BY NC SA

Il arrive que les statistiques soient rapidement périmées, et donc posent problème pour certains ordres SQL. Par exemple, une table de trace dans laquelle on insèrerait des enregistrements horodatés, et sur laquelle on voudrait presque toujours sélectionner les enregistrements des 5 dernières minutes.

Dans ce cas, il était impossible avant la 9.0 d'avoir des statistiques à jour. Maintenant, quand PostgreSQL détecte qu'une requête demande un « range scan » sur une valeur supérieure à la dernière valeur de l'histogramme (ou inférieure à la première valeur), c'est-à-dire la plus grande valeur connue au dernier calcul de statistiques, et que la colonne est indexée, il récupère le max (ou le min si c'est la première valeur) de cette colonne en interrogeant l'index AVANT d'exécuter la requête, afin d'obtenir des statistiques plus réalistes. Comme il utilise un index pour cela, il faut qu'un index existe, bien sûr.

Voici un exemple. La colonne “a” de la table test a déjà été remplie avec de nombreuses dates, antérieures. Elle a donc des statistiques à jour, avec des histogrammes lui donnant la répartition des valeurs de a.

Il est 13:37, et rien n'a encore été inséré dans la table de date supérieure à 13:37.

```
marc=# EXPLAIN ANALYZE SELECT * FROM test WHERE a > '2010-06-03 13:37:00';
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
-----------
INDEX Scan USING idxtsta ON test  (cost=0.00..8.30 rows=1 width=8) (actual time=0.007..0.007 rows=0
loops=1)
INDEX Cond: (a > '2010-06-03 13:37:00'::timestamp without time zone)
Total runtime: 0.027 ms
(3 rows)
```

Tout est donc normal. La borne supérieure de mon histogramme est 2010-06-03 13:36:16.830007 (l'information se trouve dans pg_stats). Il n'a aucun moyen d'évaluer le nombre d'enregistrements supérieurs à 13:37, et en 8.4, il aurait continué à estimer '1' tant qu'un ANALYZE n'aura pas été passé.

```
marc=# DO LANGUAGE plpgsql
$$
DECLARE
i int;
BEGIN
FOR i IN 1..10000 LOOP
   INSERT INTO test VALUES (clock_timestamp());
END LOOP;
END
$$
;
```

Nous venons d'insérer 10000 dates supérieures à 13:37.

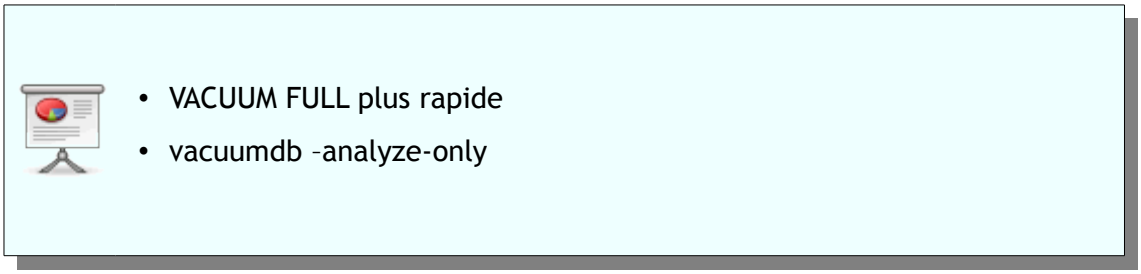
```
marc=# EXPLAIN ANALYZE SELECT * FROM test WHERE a > '2010-06-03 13:37:00';
QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
--------------------
INDEX Scan USING idxtsta ON test  (cost=0.00..43.98 rows=1125 width=8) (actual time=0.012..13.590
rows=10000 loops=1)
INDEX Cond: (a > '2010-06-03 13:37:00'::timestamp without time zone)
Total runtime: 23.567 ms
(3 rows)
```

Le nombre d'enregistrements estimé n'est pas à 0 ou 1. Et pourtant les statistiques ne sont pas à jour :

```
marc=# SELECT last_autoanalyze FROM pg_stat_user_tables WHERE relname = 'test';
last_autoanalyze
-------------------------------
2010-06-03 13:36:21.553477+02
(1 row)
```

Dans cet exemple, nous avons tout de même une erreur d'un facteur 10. Ce n'est pas si mal: sans cette optimisation, l'erreur aurait été d'un facteur 10 000. En tout cas, une erreur d'un facteur 10 nous donne de plus fortes chances de choisir un plan intelligent.

18 Des VACUUM plus efficaces



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

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

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

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

crée une nouvelle version.

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

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

En 8.4 :

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

Soit environ 8 secondes.

En 9.0 :

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

Soit pratiquement quatre fois plus rapide.

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

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

vacuumdb -analyze-only

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

CC BY NC SA

19 Configuration

- Configuration par tablespace: seq_page_cost/random_page_cost
- Configuration par base+rôle
- Meilleur suivi des modifications de configuration

Cette nouvelle version apporte également plus de granularité au niveau des paramètres de configuration.

Voici trois exemples.

19.1 Configuration par tablespace: seq_page_cost/random_page_cost

Il est désormais possible de modifier les paramètres random_page_cost et seq_page_cost pour tous les objets d'un tablespace donné.

Par exemple, pour le tablespace par défaut (`pg_default`) :

```
# ALTER TABLESPACE pg_default SET (random_page_cost = 10, seq_page_cost=5);
ALTER TABLESPACE
```

Quel peut être le cas d'utilisation ?

Imaginez des tablespaces placés sur des disques de performances différentes. Par exemple, vous avez quelques données essentielles sur un disque SSD ou bien des données d'historique sur une vieille baie moins performante que la baie flambant neuf que vous avez décidé d'utiliser pour les données actives. Cela vous permet de signaler à PostgreSQL que tous vos tablespaces ne sont pas forcément équivalents en terme de performance. Cela ne s'applique, bien sûr, que sur de très grosses bases.

19.2 Configuration par base de données+rôle

Autre avancée, on peut maintenant définir la valeur d'un paramètre pour un rôle à l'intérieur d'une base spécifique :

```
marc=# ALTER ROLE marc IN database d1 SET log_statement to 'all';
ALTER ROLE
```

CC BY NC SA

Pour connaître les valeurs des paramètres suivant le rôle et la base de données, une nouvelle méta-commande a été ajoutée à psql :

```
marc=# \drds
         List of settings
role | DATABASE |     settings
-----+----------+-------------------
marc | d1       | log_statement=ALL
(1 row)
```

Il y a donc eu une modification du catalogue pour gérer cette nouvelle fonctionnalité :

```
TABLE "pg_catalog.pg_db_role_setting"

  COLUMN     |  Type   | Modifier
-------------+---------+----------
setdatabase | oid     | NOT NULL
setrole     | oid     | NOT NULL
setconfig   | text    |
```

19.3 Tracer les paramètres modifiés

Après le rechargement du fichier `postgresql.conf`, Postgres vous signalera dans le fichier de trace les paramètres qui ont été modifiés.

Voici un exemple, lors de la modification du paramètre log_line_prefix :

```
LOG:  received SIGHUP, reloading configuration files
<%> LOG:  parameter "log_line_prefix" changed to "<%u%%%d> "
```

20 EXPLAIN

- Plus souple grâce à sa nouvelle syntaxe
- Plus de formats de sortie : XML, JSON, YAML
- Plus d'options :
  - ANALYZE pour avoir des informations réelles
  - VERBOSE pour avoir des informations plus verbeuses
  - COSTS pour afficher les coûts

- BUFFERS pour dénombrer les lectures dans le cache de PostgreSQL et en dehors

Voici un EXPLAIN ANALYZE dans les versions précédentes :

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

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

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

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

Par exemple :

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

```
            Buffers: shared hit=54 READ=12
 Total runtime: 1.526 ms
(18 rows)
```

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

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

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

21 Statistiques d'activité

- Nom de l'application
  - Configurable lors de la connexion ou par son paramètre
  - Affiché dans les traces et dans pg_stat_activity
- Forcer le nombre valeurs distinctes par colonne

21.1 application_name pour pg_stat_activity

Dans la session de supervision :

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

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

Dans la session de PID 5991 :

CC BY NC SA

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

Dans la session de supervision :

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

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

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

21.2 Forcer le nombre de valeurs distinctes d'une colonne

Ceci permet de forcer le nombre de valeurs différentes d'une colonne. Ce n'est pas à utiliser à la légère, mais uniquement quand l'ANALYZE sur la table n'arrive pas à obtenir une valeur raisonnable.

Voici comment procéder :

```
marc=# ALTER TABLE test ALTER COLUMN a SET (n_distinct = 2);
ALTER TABLE
```

Il faut repasser un ANALYZE pour que la modification soit prise en compte :

```
marc=# ANALYZE test;
ANALYZE
```

Essayons maintenant :

```
marc=# EXPLAIN SELECT distinct * from test;
QUERY PLAN
------------------------------------------------------------------
HashAggregate  (cost=6263.00..6263.02 rows=2 width=8)
->  Seq Scan ON test  (cost=0.00..5338.00 rows=370000 width=8)
(2 rows)
```

C'est l'exemple même de ce qu'il ne faut pas faire : il y a bien 370 000 valeurs distinctes dans ma table. Maintenant, avec ce mauvais paramétrage, les plans d'exécution seront très mauvais.

Si la valeur n_distinct est positive, il s'agit du nombre de valeurs distinctes. Si la valeure est négative (entre 0 et -1), il s'agit du coefficient multiplicateur par rapport au nombre d'enregistrements estimés de la table : par exemple -0.2 signifie qu'il y a 1 enregistrement distinct pour 5 enregistrements dans la table. 0 ramène le comportement à celui par défaut (c'est ANALYZE qui effectue la mesure).

CC BY NC SA

Ne touchez à ceci que si vous êtes absolument sûr d'avoir correctement diagnostiqué votre problème. Sinon, vous pouvez être sûr d'empirer les performances.

22 Langages procéduraux

- PL/pgSQL activé par défaut
- Améliorations sur les divers langages PL
- ALIAS avec PL/pgsql

Quelques nouveautés intéressantes au niveau des langages de procédures.

PL/pgsql est activé par défaut. Vous n'aurez plus à ajouter le langage PL/pgsql dans chaque base où vous en avez besoin car il est installé systématiquement.

Par ailleurs, beaucoup de langages ont vu leur support grandement amélioré, PL/perl et PL/python notamment. Consultez les `release notes` si vous voulez davantage de détails, les modifications étant trop nombreuses pour être citées ici.

Enfin, nous pouvons maintenant utiliser le mot clé ALIAS. Comme son nom l'indique, il permet de créer des alias de variables. La syntaxe est « nouveau_nom ALIAS FOR ancien_nom ». Cela se place dans la section DECLARE d'un code PL/pgsql.

C'est utilisable dans deux cas principalement :

- pour donner des noms aux variables d'une fonction PL (`monparam ALIAS FOR $ 0`)
- pour renommer des variables qui pourraient être en conflit, dans un trigger par exemple (`nouvelle_valeur ALIAS FOR new`)

23 Modules

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

23.1 Ajout du module contrib pg_upgrade

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

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

23.2 Ajout du module contrib passwordcheck

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

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

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

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

23.3 Amélioration du module contrib hstore

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

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

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

23.4 Compteurs sur buffers dans pg_stat_statements

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

```
marc=# SELECT * FROM pg_stat_statements ORDER BY total_time DESC LIMIT 2;
-[ RECORD 1 ]-------+---------------------
userid              | 10
dbid                | 16485
query               | SELECT * FROM fils ;
calls               | 2
total_time          | 0.491229
rows                | 420000
shared_blks_hit     | 61
shared_blks_read    | 2251
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
-[ RECORD 2 ]-------+---------------------
userid              | 10
dbid                | 16485
query               | SELECT * FROM pere;
calls               | 2
total_time          | 0.141445
rows                | 200000
shared_blks_hit     | 443
shared_blks_read    | 443
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
```

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

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

- Qui génère des tris ?

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

23.5 Amélioration du module contrib auto_explain

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

24 Divers

- Fenêtrage : plus d'options
- Tri dans les agrégats
- Recherche plein texte : filtrage (unaccent)

24.1.1 Nouvelles options de 'frame' dans les fonctions de fenêtrage

Si vous ne vous connaissez pas les fonctions de fenêtrage, lisez la présentation de la 8.4 (http://blog.postgresql.fr/index.php?post/2009/04/28/Nouveaut%C3%A9s-PostgreSQL-8.4) et un article de GNU/Linux Magazine France (http://www.dalibo.org/hs44_postgresql_8.4).

Il y a donc des nouveautés dans le paramétrage du 'frame' des fonctions de fenêtrage. Soit la table suivante :

```
marc=# SELECT * FROM salaire ;
entite | personne  | salaire | date_embauche
-------+-----------+---------+---------------
R&D    | marc      |  700.00 | 2010-02-15
Compta | etienne   |  800.00 | 2010-05-01
R&D    | maria     |  700.00 | 2009-01-01
R&D    | kevin     |  500.00 | 2009-05-01
R&D    | jean      | 1000.00 | 2008-07-01
R&D    | tom       | 1100.00 | 2005-01-01
```

Voici un exemple de fonctions de fenêtrage, sans préciser le frame.

```
marc=# SELECT entite, personne, salaire, date_embauche, avg(salaire) OVER (PARTITION BY entite
ORDER BY date_embauche) FROM salaire;
entite | personne  | salaire | date_embauche |          avg
-------+-----------+---------+---------------+-----------------------
Compta | stephanie |  850.00 | 2006-01-01    |  850.0000000000000000
Compta | etienne   |  800.00 | 2010-05-01    |  825.0000000000000000
R&D    | tom       | 1100.00 | 2005-01-01    | 1100.0000000000000000
R&D    | jean      | 1000.00 | 2008-07-01    | 1050.0000000000000000
R&D    | maria     |  700.00 | 2009-01-01    |  933.3333333333333333
R&D    | kevin     |  500.00 | 2009-05-01    |  825.0000000000000000
R&D    | marc      |  700.00 | 2010-02-15    |  800.0000000000000000
```

Le frame est le groupe d'enregistrements sur lequel la fonction de fenêtrage est appliquée. Évidemment, si on ne précise pas le frame, il met une valeur par défaut. Voici la même requête, écrite avec un frame explicite.

```
marc=# SELECT entite, personne, salaire, date_embauche, avg(salaire) OVER (PARTITION BY entite
ORDER BY date_embauche RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM salaire;
entite | personne  | salaire | date_embauche |          avg
-------+-----------+---------+---------------+-----------------------
Compta | stephanie |  850.00 | 2006-01-01    |  850.0000000000000000
Compta | etienne   |  800.00 | 2010-05-01    |  825.0000000000000000
R&D    | tom       | 1100.00 | 2005-01-01    | 1100.0000000000000000
R&D    | jean      | 1000.00 | 2008-07-01    | 1050.0000000000000000
R&D    | maria     |  700.00 | 2009-01-01    |  933.3333333333333333
R&D    | kevin     |  500.00 | 2009-05-01    |  825.0000000000000000
R&D    | marc      |  700.00 | 2010-02-15    |  800.0000000000000000
```

Le frame est donc par 'range', entre le début du range et l'enregistrement courant (pas vraiment l'enregistrement courant en fait, mais laissons de côté un instant les subtilités). On constate que la fonction de moyenne (avg) est appliquée entre le premier des enregistrements du frame (les enregistrements de la même entité) et l'enregistrement courant.

Première nouveauté : en 9.0, le frame peut se calculer entre l'enregistrement courant et la fin du groupe (au lieu d'entre le début du groupe et l'enregistrement courant) :

```
marc=# SELECT entite, personne, salaire, date_embauche, avg(salaire) OVER (PARTITION BY entite
ORDER BY date_embauche RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) FROM salaire;
entite | personne  | salaire | date_embauche |          avg
-------+-----------+---------+---------------+-----------------------
Compta | stephanie |  850.00 | 2006-01-01    | 825.0000000000000000
Compta | etienne   |  800.00 | 2010-05-01    | 800.0000000000000000
R&D    | tom       | 1100.00 | 2005-01-01    | 800.0000000000000000
R&D    | jean      | 1000.00 | 2008-07-01    | 725.0000000000000000
R&D    | maria     |  700.00 | 2009-01-01    | 633.3333333333333333
R&D    | kevin     |  500.00 | 2009-05-01    | 600.0000000000000000
R&D    | marc      |  700.00 | 2010-02-15    | 700.0000000000000000
```

Deuxième nouveauté, on peut calculer des frames sur les n enregistrements précédents et n enregistrements suivants. Aucun intérêt avec ce jeu de données, mais voici malgré tout un exemple :

```
marc=# SELECT entite, personne, salaire, date_embauche, avg(salaire) OVER (PARTITION BY entite
ORDER BY date_embauche RANGE ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) FROM salaire;
entite | personne  | salaire | date_embauche |          avg
-------+-----------+---------+---------------+-----------------------
Compta | stephanie |  850.00 | 2006-01-01    |  825.0000000000000000
Compta | etienne   |  800.00 | 2010-05-01    |  825.0000000000000000
R&D    | tom       | 1100.00 | 2005-01-01    | 1050.0000000000000000
R&D    | jean      | 1000.00 | 2008-07-01    |  933.3333333333333333
```

```
R&D     | maria     |  700.00 | 2009-01-01    |  733.3333333333333333
R&D     | kevin     |  500.00 | 2009-05-01    |  633.3333333333333333
R&D     | marc      |  700.00 | 2010-02-15    |  600.0000000000000000
```

On reste bien sûr sur le groupe (voir l'enregistrement de tom par exemple, l'enregistrement d'etienne ne rentre pas dans le calcul de sa moyenne).

Si on voulait la même requête que précédemment, mais avec des moyennes sur 3 enregistrements glissants, sans réinitialiser à chaque entité (toujours aucun intérêt pratique dans l'exemple).

```
marc=# SELECT entite, personne, salaire, date_embauche, avg(salaire) OVER (ORDER BY entite,
date_embauche ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) FROM salaire;
entite | personne  | salaire | date_embauche |         avg
--------+-----------+---------+---------------+----------------------
Compta | stephanie |  850.00 | 2006-01-01    | 825.0000000000000000
Compta | etienne   |  800.00 | 2010-05-01    | 916.6666666666666667
R&D    | tom       | 1100.00 | 2005-01-01    | 966.6666666666666667
R&D    | jean      | 1000.00 | 2008-07-01    | 933.3333333333333333
R&D    | maria     |  700.00 | 2009-01-01    | 733.3333333333333333
R&D    | kevin     |  500.00 | 2009-05-01    | 633.3333333333333333
R&D    | marc      |  700.00 | 2010-02-15    | 600.0000000000000000
```

Bref, un outil à maîtriser d'urgence, si ce n'est pas déjà le cas (même s'il est difficile de donner un exemple décent).

24.1.2 Tris dans les agrégats

Cette nouveauté est un peu subtile : le résultat de certaines fonctions d'agrégat dépend de l'ordre dans lequel on leur fournit les données.

Il ne s'agit évidemment pas de count, avg, mais plutôt de array_agg, xml_agg, string_agg…

Ce qui va permettre de vous présenter deux nouvelles fonctionnalités d'un coup, string_agg étant une nouveauté de la 9.0.

Reprenons la table salaire. Je voudrais la liste des employés, concaténés dans un seul champ, par entité. C'est pour stocker dans mon tableur 😀

```
marc=# SELECT entite, string_agg(personne, ', ') FROM salaire GROUP BY entite;
entite |          string_agg
-------+------------------------------
Compta | etienne, stephanie
R&D    | marc, maria, kevin, jean, tom
```

C'est déjà bien. Mais j'aimerais bien les avoir par ordre alphabétique, parce que je ne sais pas écrire de macro dans mon tableur pour retrier les données.

```
marc=# SELECT entite, string_agg(personne, ', ' ORDER BY personne) FROM salaire GROUP BY entite;
entite |          string_agg
--------+------------------------------
Compta | etienne, stephanie
R&D    | jean, kevin, marc, maria, tom
```

Il suffit donc de rajouter une clause de tri à l'intérieur de la fonction d'agrégat, sans virgule à la fin.

24.2 Dictionnaire de filtrage (unaccent)

Il est possible maintenant de paramétrer des dictionnaires de filtrage. On parle bien sûr des dictionnaires du Full Text Search.

Le but de ces dictionnaires de filtrage est d'appliquer un premier filtrage sur les mots avant de les indexer. Le module présenté ci-dessous (unaccent) est l'illustration de ce mécanisme. Le filtrage peut consister en la suppression de mots ou en leur modification.

Unaccent ne supprime pas les mots, il supprime les accents (tous les signes diacritiques en fait), en remplaçant les caractères accentués par leur version sans accent. Unaccent est un module contrib.

Nous allons à peu près suivre la documentation d'unaccent, les auteurs ayant eu la gentillesse de donner leurs exemples en français.

Nous créons un nouveau dictionnaire fr (pour ne pas polluer le dictionnaire french 'standard') :

```
marc=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
CREATE TEXT SEARCH CONFIGURATION
```

Nous modifions le paramétrage de 'fr' pour les lexemes de type mot, en lui demandant de les faire passer par unaccent et french_stem (au lieu de seulement french_stem) :

```
marc=# ALTER TEXT SEARCH CONFIGURATION fr ALTER MAPPING FOR hword, hword_part, word WITH unaccent,
french_stem;
ALTER TEXT SEARCH CONFIGURATION
marc=# SELECT to_tsvector('fr','Hôtels de la Mer');
to_tsvector
-------------------
'hotel':1 'mer':4
(1 row)

marc=# SELECT to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
?COLUMN?
----------
t
(1 row)
```

Cela vous permet donc, sans changer une ligne de code, et en gardant les caractères accentués, de rechercher maintenant sans accent.

25 Régressions

- PL/pgSQL : nom des variables
- PL/pgSQL : mots réservés

Ces deux changements dans PL/pgSQL peuvent entraîner des régressions dans du code fonctionnant en 8.4. Si vous avez du code PL/pgSQL, vérifiez-le avant de migrer en 9.0. Le moteur génère des erreurs à l'exécution, comme illustré ci-dessous.

25.1 PL/pgSQL : meilleur contrôle du nom des variables

```
marc=# DO LANGUAGE plpgsql
$$
DECLARE
a int;
BEGIN
SELECT a FROM test;
END
$$
ERROR:  COLUMN reference "a" IS ambiguous
LINE 1: SELECT a FROM test
DETAIL:  It could refer TO either a PL/pgSQL variable OR a TABLE COLUMN.
QUERY:  SELECT a FROM test
CONTEXT:  PL/pgSQL FUNCTION "inline_code_block" line 4 at SQL statement
```

Si vous voulez modifier ce comportement, vous pouvez le faire globalement mais il est préférable de le faire par fonction, en exécutant une de ces commandes au début de votre fonction:

```
variable_conflict error        # mode par défaut
variable_conflict use_variable # choisir le nom de variable
variable_conflict use_column   # choisir le nom de colonne
```

25.2 Protection des mots réservés en PL/pgsql

```
marc=# DO LANGUAGE plpgsql
$$
DECLARE
TABLE int;
BEGIN
TABLE :=TABLE+1;
END
$$
;
ERROR:  syntax error at OR near "table"
LINE 6: TABLE :=TABLE+1;
              ^
marc=# DO LANGUAGE plpgsql
$$
DECLARE
"table" int;
```

CC BY NC SA

```
BEGIN
"table" :="table"+1;
END
$$
;
DO
```

26 Bilan

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

27 Et la suite ?

- Sortie prévue : été 2011 ?
- Version alpha 1 déjà disponible.
- Réplication synchrone, support des tables distantes via SQL/Med, support des labels de sécurité (?)
- Réduction de la taille des champs de type NUMERIC sur disque
- Compteurs du nombre de VACUUM et ANALYZE pour les tables pg_stat_*_tables
- Attention, standard_conforming_strings par défaut !

BY NC SA

http://www.postgresql.org/about/news.1233

28 Pour aller plus loin

- La documentation officielle en ligne
- Illustrated PostgreSQL
- Des articles dans GNU/Linux Magazine France
- Télécharger cette conférence

La meilleur source d'information reste avant tout la documentation officielle de PostgreSQL 9.0. Vous y trouverez les informations essentielles sur chaque nouveautés :

http://docs.postgresql.fr/9.0/

Plus spécifiquement, vous pouvez consultez la page What's new in PostgreSQL 9.0 basée sur le travail de Marc Cousin :

http://wiki.postgresql.org/wiki/What's_new_in_PostgreSQL_9.0

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

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

29 Questions ?

- N'hésitez pas c'est le moment !

CC BY NC SA