−Table des matières

PostgreSQL 9.0, les autres nouveautés

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 134 (Janvier 2011). Il est disponible maintenant sous licence Creative Commons.

PostgreSQL 9, ce n'est heureusement pas que la réplication. Cette nouvelle version comprend un grand nombre d'autres nouvelles fonctionnalités et aussi beaucoup d'améliorations sur les fonctionnalités déjà existantes. En tout, on décompte plus de 200 nouveautés dans cette version. Nous ne pourrons pas tout voir dans cet article mais néanmoins, nous allons aborder les plus importantes.

SQL

Recherche plein texte : dictionnaire filtrant

Jusqu'à présent, un dictionnaire de recherche plein texte ne pouvait renvoyer que trois types de résultats : un tableau de lexèmes si le mot recherché fait partie du dictionnaire, un tableau vide si le mot recherché est un mot courant (appelé couramment « stop word ») ou enfin NULL si le mot recherché n'est pas trouvé. Dans ce dernier cas, le mot recherché est envoyé au dictionnaire suivant dans la configuration utilisée.

La version 9.0 ajoute un nouveau type de dictionnaire : le dictionnaire filtrant. Le but de ce dictionnaire est de transformer le mot recherché en un autre, et c'est ce nouveau mot qui sera envoyé au dictionnaire suivant. Cela permet d'implémenter des fonctionnalités assez intéressantes. La première, qui est disponible sous la forme d'un module contrib, est le dictionnaire unaccent. Ce dictionnaire a pour but de transformer un mot qui peut avoir des accents en le même mot, mais sans accents. Cela facilite grandement la recherche, étant donné qu'un utilisateur n'a plus à se préoccuper de saisir ou non un accent pendant sa recherche.

Pour tester ça, commençons par l'installation du module contrib. L'installation est très simple, il suffit d'exécuter un script SQL qui s'appelle unaccent.sql. Ce script est installé sur votre distribution à partir du package contrib de la version 9.0. Si vous avez compilé vous-même PostgreSQL 9.0, vous devez compiler et installer le module contrib avec les ordres « make && make install » une fois dans le répertoire contrib/unaccent du répertoire des sources. Ceci fait, il faut exécuter le script SQL d'installation du module :

```
b1=# \i /opt/postgresql-9.0/share/contrib/unaccent.sql
SET
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE TEXT SEARCH TEMPLATE
CREATE TEXT SEARCH DICTIONARY
```

Nous allons maintenant copier la configuration française par défaut.

```
b1=# CREATE TEXT SEARCH CONFIGURATION fr (COPY = french);
CREATE TEXT SEARCH CONFIGURATION
```

Testons déjà cette configuration :

```
b1=# SELECT to_tsvector('fr', 'une fois dans le répertoire contrib/unaccent du répertoire des sources') @@ to_tsquery('fr', 'répertoire');
 ?column?
----------
 t
(1 row)

b1=# SELECT to_tsvector('fr', 'une fois dans le répertoire contrib/unaccent du répertoire des sources') @@ to_tsquery('fr', 'repertoire');
 ?column?
----------
 f
(1 row)
```

La recherche avec accent trouve un résultat alors que la recherche sans accent n'en trouve pas. Modifions maintenant la configuration fr pour qu'elle utilise le dictionnaire filtrant unaccent :

```
b1=# ALTER TEXT SEARCH CONFIGURATION fr ALTER MAPPING FOR hword, hword_part, word WITH unaccent, french_stem;
ALTER TEXT SEARCH CONFIGURATION
b1=# SELECT to_tsvector('fr', 'une fois dans le répertoire contrib/unaccent du répertoire des sources') @@ to_tsquery('fr', 'répertoire');
 ?column?
----------
```

```
 t
(1 row)

b1=# SELECT to_tsvector('fr', 'une fois dans le répertoire contrib/unaccent du répertoire des sources') @@ to_tsquery('fr', 'repertoire');
 ?column?
----------
 t
(1 row)
```

Maintenant, il fait bien une correspondance, que le mot recherché ait un accent ou pas.

Dans les nouveautés de la recherche plein texte, il ne faut pas oublier le support des préfixes dans les dictionnaires des synonymes. Chaque mot d'un dictionnaire de synonymes peut se voir affecter un préfixe de correspondance. L'exemple de la documentation est très révélateur. Le mot « indices » est souvent utilisé comme pluriel de « index ». Il est donc possible de définir le dictionnaire des synonymes en indiquant que « indices » a comme synonyme « index* », ce qui permettra une correspondance avec tout mot commençant par index (index, indexes, etc).

Contraintes d'exclusion

PostgreSQL dispose de plusieurs types de contraintes depuis très longtemps : des contraintes de vérification (CHECK), des contraintes d'unicité, des contraintes NOT NULL, des clés primaires et des clés étrangères. Prenons la contrainte d'unicité. Elle permet d'empêcher qu'une même donnée se trouve deux fois dans la même colonne ou dans le même groupe de colonnes. Par exemple, supposons que nous cherchions à avoir une table d'utilisateurs pour laquelle nous allons stocker leur login, mot de passe, nom et prénom. La requête de création de la table est simple :

```
CREATE TABLE utilisateurs (login text, motdepasse text, prenom text, nom text);
```

Les utilisateurs vont pouvoir être ajoutés avec de simples requêtes INSERT :

```
b1=# INSERT INTO utilisateurs VALUES ('guillaume', 'secret1', 'Guillaume', 'Lelarge');
INSERT 0 1
b1=# INSERT INTO utilisateurs VALUES ('jpa', 'secret2', 'Jean-Paul', 'Argudo');
INSERT 0 1
```

Il y aura bien un jour où un utilisateur va avoir le même login :

```
b1=# INSERT INTO utilisateurs VALUES ('guillaume', 'secret3', 'Guillaume', 'Smet');
INSERT 0 1
```

Syntaxiquement parlant, la requête est tout à fait valide et son exécution ne doit pas produire d'erreur. Logiquement parlant, avoir des utilisateurs qui partagent le même login est à interdire. Cela peut évidemment se faire du côté applicatif mais il faudra dans ce cas que toutes les applications soient codées pour empêcher cela. Et si un administrateur passe par psql et modifie la valeur en utilisant une valeur déjà utilisée ? Vous êtes eu. C'est principalement pour cela qu'ont été créées les contraintes d'unicité. Ajoutons-en une sur cette table après avoir supprimé notre troisième utilisateur :

```
b1=# DELETE FROM utilisateurs WHERE nom='Smet';
DELETE 1
b1=# ALTER TABLE utilisateurs ADD UNIQUE (login);
NOTICE:  ALTER TABLE / ADD UNIQUE will create implicit index "utilisateurs_login_key" for table "utilisateurs"
ALTER TABLE
```

Remarquons qu'on passe par un index pour une contrainte unique. Essayons de rajouter Guillaume :

```
b1=# INSERT INTO utilisateurs VALUES ('guillaume', 'secret3', 'Guillaume', 'Smet');
ERROR:  duplicate key value violates unique constraint "utilisateurs_login_key"
DETAIL:  Key (login)=(guillaume) already exists.
```

L'insertion a été refusée grâce à la contrainte d'unicité. En cas de sélection d'un autre login, l'insertion réussit :

```
b1=# INSERT INTO utilisateurs VALUES ('gsmet', 'secret3', 'Guillaume', 'Smet');
INSERT 0 1
```

Une contrainte d'unicité peut aussi se définir sur plusieurs colonnes mais l'opérateur chargé de la vérification sera toujours celui de l'égalité, ce qui fait que l'index utilisé pour forcer la contrainte est forcément un Btree. Il est intéressant dans certains cas de pouvoir faire varier les opérateurs utilisés suivant les colonnes. Disons que nos utilisateurs sont des conférenciers et que ces derniers peuvent demander l'attribution d'une salle pour leur conférence. Évidemment, deux conférenciers doivent pouvoir réserver la même salle, mais pas sur des horaires identiques ou qui simplement se chevauchent. Il faudrait donc pouvoir définir une contrainte qui vérifie cette règle métier : l'égalité pour la salle de conférence et le chevauchement des horaires. C'est le but des contraintes d'exclusion.

Pour définir des horaires (donc une date/heure de début et une date/heure de fin), il nous faut utiliser une extension de PostgreSQL appelée temporal. Elle définit un nouveau type de données (period) ainsi que des opérateurs dont celui de surcharge (&&). Nous allons donc créer une nouvelle table, appelée reservation ayant trois colonnes (salle, conferencier et periode) :

```
b1=# CREATE TABLE reservation (salle TEXT, conferencier TEXT, periode PERIOD);
CREATE TABLE
```

Et nous ajoutons la contrainte exactement comme nous l'avons défini au paragraphe précédent :

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

Testons le fonctionnement par quelques insertions :

```
b1=# INSERT INTO reservation (salle, conférencier, periode)
    VALUES ('Berlin 1+2', 'Bruce Momjian', period('2010-12-07 13:10:00', '2010-12-07 14:00:00'));
INSERT 0 1
b1=# INSERT INTO reservation (salle, conférencier, periode)
    VALUES ('Berlin 1+2', 'Heikki Linnakangas', period('2010-12-07 14:10:00', '2010-12-07 15:00:00'));
INSERT 0 1
b1=# INSERT INTO reservation (salle, conférencier, periode)
    VALUES ('Berlin 1+2', 'Dimitri Fontaine', period('2010-12-07 14:30:00', '2010-12-07 15:30:00'));
ERROR:  conflicting KEY value violates exclusion constraint "surcharge_période"
DETAIL:  KEY (salle, periode)=(Berlin 1+2, [20101207 14:30:00+02, 20101207 15:30:00+02))
conflicts WITH existing KEY (salle, periode)=(Berlin 1+2, [20101207 14:10:00+02, 20101207
15:00:00+02)).
```

Et voilà. La version 9.0 gère un type de contrainte qu'à ma connaissance, aucun autre moteur de base de données ne sait gérer.

Contraintes uniques déferrables

Dans les versions précédentes, il n'était pas rare de voir des utilisateurs assez déconcertés par le message d'erreur qui suivait l'exécution d'une commande comme « UPDATE t2 SET c2=c2+1 », en sachant que c2 est contrôlé par une contrainte d'unicité. En effet, même si l'unicité sera respectée une fois la commande entièrement exécutée, il se trouve que, pendant l'exécution, certaines lignes pourraient avoir la même valeur. Prenons un exemple. Voici l'ancien comportement :

```
b1=# CREATE TABLE t1 (c1 integer UNIQUE);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "t1_c1_key" for table "t1"
CREATE TABLE
b1=# INSERT INTO t1 VALUES (1), (2), (3);
INSERT 0 3
b1=# INSERT INTO t1 VALUES (2);
ERROR:  duplicate key value violates unique constraint "t1_c1_key"
DETAIL:  Key (c1)=(2) already exists.
b1=# UPDATE t1 SET c1=c1+1;
ERROR:  duplicate key value violates unique constraint "t1_c1_key"
DETAIL:  Key (c1)=(2) already exists.
```

L'insertion ne peut pas fonctionner car la valeur qu'on essaie d'insérer se trouve déjà dans la même colonne d'une autre ligne. Par contre, la mise à jour devrait pouvoir fonctionner. Ajouter la même valeur à toutes les lignes nous assure qu'à la fin de l'opération, toutes les valeurs de cette colonne seront différentes pour toutes les lignes de la table. Cependant, comme PostgreSQL vérifiait la valeur à chaque ligne modifiée, il est « logique » que PostgreSQL trouve des violations de contrainte pendant l'opération. Remarquez aussi que le message d'erreur vous indique maintenant la valeur pour laquelle la violation de contrainte est détectée.

Quant au nouveau comportement de la version 9.0, il demande à créer la contrainte avec la clause DEFERRABLE, ce qui nous donne :

```
b1=# CREATE TABLE t2 (c1 integer UNIQUE DEFERRABLE);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "t2_c1_key" for table "t2"
CREATE TABLE
b1=# INSERT INTO t2 VALUES (1), (2), (3);
INSERT 0 3
b1=# INSERT INTO t2 VALUES (2);
ERROR:  duplicate key value violates unique constraint "t2_c1_key"
DETAIL:  Key (c1)=(2) already exists.
b1=# UPDATE t2 SET c1=c1+1;
UPDATE 3
```

L'insertion/modification d'une valeur dupliquée n'est toujours pas possible si la contrainte est violée à la fin de la transaction. Par contre, elle est possible si la contrainte est assurée en fin de transaction. Cela devrait contenter plus d'un utilisateur gêné par l'ancien comportement.

Triggers : par colonne, et conditionnel

PostgreSQL dispose des triggers depuis très longtemps. Cependant, ils sont exécutés à chaque fois qu'une ligne est insérée, supprimée ou modifiée. Dans le cas de la modification, cela peut être un problème. En effet, on peut vouloir déclencher un trigger que si une colonne particulière est modifiée. Dans les versions précédentes de PostgreSQL, on devait avoir un trigger qui se déclenche à chaque modification d'une ligne (quelque soit le ou les colonnes réellement modifiées) et tester dans la procédure exécutée si la colonne modifiée était bien celle qui nous intéressait. Le déclenchement à chaque ligne a un coût qui peut se révéler important. Avec cette nouvelle version, les développeurs de PostgreSQL ont ajouté la possibilité de spécifier une ou plusieurs colonnes. Le trigger n'est déclenché que si la ou les colonnes sont mentionnées dans la clause SET de l'instruction UPDATE (donc pas forcément si la colonne est modifiée). Voici un exemple de mise en place :

```
b1=# CREATE TABLE t3 (c1 integer, c2 text);
CREATE TABLE
b1=# INSERT INTO t3 SELECT i, 'Ligne '||i::text FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000
```

La procédure stockée va simplement renvoyer une trace de niveau NOTICE. Le paramètre de configuration client_min_messages valant par défaut NOTICE, cette information sera affichée à l'écran. Notez qu'elle sera aussi enregistrée dans les journaux applicatifs. La trace indiquera l'ancienne et la nouvelle valeur de la colonne c2.

```
b1=# CREATE OR REPLACE FUNCTION log() RETURNS trigger LANGUAGE plpgsql AS $$
b1$# BEGIN
b1$# RAISE NOTICE 'modification de c2 (% devient %)', old.c2, new.c2;
b1$# RETURN new;
b1$# END;
b1$# $$;
CREATE FUNCTION
```

Notez le mot clé OF après UPDATE qui permet de préciser le ou les colonnes dont la modification doit déclencher le trigger.

```
b1=# CREATE TRIGGER tr1 BEFORE UPDATE OF c2 ON t3 FOR EACH ROW EXECUTE PROCEDURE log();
CREATE TRIGGER
b1=# UPDATE t3 SET c1=c1+1 WHERE c1<5;
UPDATE 4
```

La modification de la colonne c1 n'a causé l'affichage d'aucune trace. Il faut absolument tenter la modification de la colonne c2 pour cela :

```
b1=# UPDATE t3 SET c2='LIGNE '||c1::text WHERE c1 BETWEEN 15 AND 17;
NOTICE:  modification de c2 (Ligne 15 devient LIGNE 15)
NOTICE:  modification de c2 (Ligne 16 devient LIGNE 16)
NOTICE:  modification de c2 (Ligne 17 devient LIGNE 17)
UPDATE 3
```

Là, nous avons directement modifié la colonne c2 et nous constatons l'apparition des traces.

```
b1=# UPDATE t3 SET c1=0, c2=c2 WHERE c1 BETWEEN 250 AND 252;
NOTICE:  modification de c2 (Ligne 250 devient Ligne 250)
NOTICE:  modification de c2 (Ligne 251 devient Ligne 251)
```

```
NOTICE:  modification de c2 (Ligne 252 devient Ligne 252)
UPDATE 3
```

Dans ce dernier exemple, nous modifions réellement c1 et « faussement » c2 (faussement car sa valeur ne va pas changer, ce que les traces indiquent bien). Néanmoins, malgré que la valeur de la colonne c2 ne change pas, le trigger est déclenché. Si le trigger doit exécuter un code que si l'ancienne valeur est différente de la nouvelle valeur, il faudra toujours ajouter un test dans la procédure stockée.

Le but de cette fonctionnalité est une meilleure lisibilité et de meilleures performances. Les quelques tests que j'ai pu effectué ne montrent pas vraiment d'amélioration des performances. Par contre, le code est bien plus lisible et compréhensible.

Autre nouvelle fonctionnalité des triggers, le trigger conditionnel. Le but est de n'exécuter un trigger que dans certaines conditions explicitées par la nouvelle clause WHEN. Cette condition peut faire référence aux valeurs anciennes et nouvelles des colonnes de la table dans le cas d'un trigger niveau ligne (clause FOR EACH ROW). Il est aussi possible d'utiliser un trigger conditionnel dans le cas d'un trigger niveau instruction mais comme il n'est pas possible de tester ancienne et nouvelle valeur de chaque colonne, l'intérêt de cette fonctionnalité est beaucoup plus limité.

Testons la clause WHEN pour ajouter la détection de différence sur la colonne. Commençons par supprimer l'ancien trigger :

```
b1=# DROP TRIGGER tr1 on t3;
DROP TRIGGER
```

Puis ajoutons le nouveau :

```
b1=# CREATE TRIGGER tr1 BEFORE UPDATE OF c2 ON t3 FOR EACH ROW WHEN (old.c2 IS DISTINCT FROM new.c2) EXECUTE PROCEDURE log();
CREATE TRIGGER
```

Et maintenant, testons cela :

```
b1=# UPDATE t3 SET c1=0, c2=c2 WHERE c1 BETWEEN 253 AND 255;
UPDATE 3
b1=# UPDATE t3 SET c1=0, c2=c2||' ' WHERE c1 BETWEEN 256 AND 258;
NOTICE:  modification de c2 (Ligne 256 devient Ligne 256 )
NOTICE:  modification de c2 (Ligne 257 devient Ligne 257 )
NOTICE:  modification de c2 (Ligne 258 devient Ligne 258 )
UPDATE 3
```

Le trigger est bien exécuté quand le contenu de c2 est modifié et pas dans le cas contraire.

Amélioration dans les agrégats

Prenons comme exemple une table contenant un numéro d'ordre et un libellé.

```
b1=# CREATE TABLE t4 (c1 integer, c2 text);
CREATE TABLE
b1=# INSERT INTO t4 (c1, c2) VALUES (4, 'quatre'), (2, 'deux'), (1, 'un'), (5, 'cinq'), (3, 'trois');
INSERT 0 5
```

Maintenant, nous voulons agréger tous les éléments de la colonne c2 sous la forme d'un tableau :

```
b1=# SELECT array_agg(c2) FROM t4;
         array_agg
-----------------------------
 {quatre,deux,un,cinq,trois}
(1 row)
```

Si nous voulons en plus ordonner les éléments du tableau suivant la colonne d'ordre (c'est-à-dire c1), nous nous heurtons à un problème assez compliqué :

```
b1=# SELECT array_agg(c2) FROM t4 ORDER BY c1;
ERROR:  column "t4.c1" must appear in the GROUP BY clause or be used in an aggregate function
LINE 1: SELECT array_agg(c2) FROM t4 ORDER BY c1;
                                             ^
```

En effet, nous ne pouvons pas ordonner sans prendre en compte les éléments de l'agrégat. La solution passe par une sous-requête mais cela devient assez rapidement illisible. La version 9.0 apporte une solution assez élégante. La clause ORDER BY va se placer dans la fonction d'agrégat, comme ceci :

```
b1=# SELECT array_agg(c2 ORDER BY c1) FROM t4;
         array_agg
-----------------------------
 {un,deux,trois,quatre,cinq}
(1 row)
```

À noter que si la fonction d'agrégat utilise plusieurs arguments, il est nécessaire de placer la clause ORDER BY en fin des arguments. Par exemple :

```
b1=# SELECT string_agg(c2, ',' ORDER BY c1) FROM t4;
       string_agg
--------------------------
 un,deux,trois,quatre,cinq
(1 row)
```

Notez aussi cette nouvelle fonction d'agrégat, string_agg qui permet de concaténer les chaînes de caractères, avec ou sans délimiteur.

Diverses nouveautés sur le SQL

Les développeurs ont continué à ajouter des clauses « IF EXISTS » et « OR REPLACE » aux instructions DDL. Ainsi, « ALTER TABLE DROP COLUMN » et « ALTER TABLE DROP CONSTRAINT » disposent du « IF EXISTS ». Quant à « CREATE LANGUAGE », il se voit ajouter la clause « OR REPLACE ». Ça ne peut que faciliter l'écriture de scripts SQL. Les requêtes de fenêtrage bénéficient de trois nouvelles clauses : CURRENT ROW, ROWS n PRECEDING et FOLLOWING. Notez enfin qu'il est possible de créer un index sans lui donner de nom, le système s'occupant automatiquement du nommage. Cette fonctionnalité étant déjà disponible pour les contraintes (clé primaire, clé unique), elle a été étendue à tous les index.

Programmation interne

PL/pgsql activé par défaut

Ce n'est pas à proprement parler une nouvelle fonctionnalité mais cela va simplifier la vie à de nombreux administrateurs de bases de données. En 9.0, le langage PL/pgsql est activé par défaut, quelque soit la base de données. Cependant, il est toujours possible de désactiver le langage si nécessaire.

DO

Avec PostgreSQL, il est impossible d'utiliser des instructions de test comme IF ou des boucles comme FOR ailleurs que dans des procédures stockées. IF, FOR, WHILE ne sont pas connus par le standard SQL, ils ne sont donc pas reconnus directement par PostgreSQL. Pour les utiliser, il faut passer par la création d'une procédure stockée, ce qui est assez lourd quand on veut pouvoir exécuter un petit script SQL une seule fois.

Comme la majorité des développeurs s'opposaient à changer ce comportement, l'idée a été de créer une nouvelle instruction, spécifique à PostgreSQL, qui demanderait l'exécution immédiate d'une fonction dont le code est donné dans l'instruction. Cela permet d'éviter d'avoir à créer une fonction, qu'il faudra supprimer par la suite, tout en évitant l'intégration d'un grand nombre de mots-clés de tests et de boucles dans le langage SQL. Cette nouvelle instruction s'appelle DO. Son utilisation est très simple. Après l'instruction, il faut indiquer le langage, puis le code. Le langage est par défaut PL/pgsql. Il est possible de l'omettre dans ce cas. Mais vous pouvez utiliser tout langage disponible dans votre base.

Supposons, par exemple, que nous voulons supprimer toutes les tables qui commencent par la lettre t. Voici comment faire avec PostgreSQL 9.0 :

```
b1=# DO $$
b1$# DECLARE
b1$#   t text;
b1$# BEGIN
b1$# FOR t IN SELECT relname FROM pg_class WHERE relkind='r' AND relname LIKE 't%'
b1$# LOOP
b1$#   RAISE NOTICE 'Suppression de la table %', t;
b1$#   EXECUTE 'DROP TABLE ' || t;
b1$# END LOOP;
b1$# END;
b1$# $$;
NOTICE:  Suppression de la table t1
NOTICE:  Suppression de la table t2
NOTICE:  Suppression de la table t3
NOTICE:  Suppression de la table t4
DO
```

Notez bien que le code doit se conformer strictement au langage. Dans le cas du PL/pgsql par exemple, cela sous-entend qu'il faut absolument avoir l'instruction BEGIN en début (précédée si nécessaire du bloc DECLARE) et l'instruction END en fin.

Appels des procédures avec des paramètres nommés

L'une des grandes nouveautés pour les langages de procédures stockées est la possibilité d'appeler une procédure en fournissant les arguments dans n'importe quel ordre à condition de préciser le nom du paramètre. Le plus simple est de montrer un exemple :

```
b1=# CREATE FUNCTION sum(p1 integer, p2 integer) RETURNS integer LANGUAGE plpgsql AS $$
b1$# BEGIN
b1$# RAISE NOTICE 'p1 vaut %', p1;
b1$# RAISE NOTICE 'p2 vaut %', p2;
b1$# RETURN p1+p2;
b1$# END
b1$# $$;
CREATE FUNCTION
```

La fonction sum() est une bête fonction additionnant les deux entiers en argument et renvoyant le résultat. Beaucoup plus intéressant, la fonction envoie deux informations dans les traces : la valeur du premier argument (p1), et la valeur du second argument (p2). Exécutons cette fonction de la manière traditionnelle :

```
b1=# SELECT sum(10, 20);
NOTICE:  p1 vaut 10
NOTICE:  p2 vaut 20
 sum
-----
  30
(1 ligne)
```

Le premier argument est bien 10, le second 20. Tout va bien. Avec PostgreSQL 9.0, il est maintenant possible de nommer les arguments en préfixant la valeur avec le nom de l'argument et l'opérateur « := ». Voici ce que cela donne :

```
b1=# SELECT sum(p1:=10, p2:=20);
NOTICE:  p1 vaut 10
NOTICE:  p2 vaut 20
 sum
-----
  30
(1 ligne)
```

Le gros intérêt est qu'on peut fournir les arguments dans n'importe quel ordre :

```
b1=# SELECT sum(p2:=10, p1:=20);
NOTICE:  p1 vaut 20
NOTICE:  p2 vaut 10
 sum
-----
  30
(1 ligne)
```

Tant qu'on discute des nouveautés sur les paramètres, autre point intéressant. Auparavant, le langage PL/pgsql considérait les paramètres en entrée comme des constantes. Il était impossible de modifier leur valeur dans la procédure stockée. Cette restriction avait peu d'intérêt. Elle a donc été supprimée, faisant en sorte que PL/pgsql les considère maintenant comme des variables locales. Leur valeur finale n'a toujours aussi peu d'intérêt en dehors de la

procédure mais il est possible de modifier leur valeur pendant l'exécution de la procédure stockée.

Clause ALIAS

Le mot clé ALIAS permet de créer des alias de variables. Par exemple, dans un trigger, si vous ne souhaitez pas utiliser la variable new par son nom et que vous préférez la renommer pour avoir un nom plus significatif, vous utiliserez ALIAS de cette façon

```
nouvelle_ligne_t4 ALIAS FOR new
```

Gestion des conflits de variables

Sur les versions précédentes, PL/pgsql permettait l'utilisation de variables et de paramètres de même nom que des noms de colonnes ou tables, voire même que des noms de mots-clés.

La version 9.0 arrange cela grâce à un paramètre (variable_conflict) qui permet de gérer les conflits de variable. Par défaut, en cas de conflit, l'exécution est annulée. Vous pouvez changer ce paramètre pour indiquer qu'il faut utiliser par défaut la variable ou la colonne.

PL/perl et PL/python

Ces deux langages ont eu droit à un très gros travail. Je n'entrerais pas dans les détails, n'étant un expert ni de l'un ni de l'autre.

Configuration

Une configuration plus fine

PostgreSQL permet une configuration globale de l'instance via son fichier postgresql.conf. Cette configuration est surchargeable par certains objets : les bases de données, les utilisateurs et les procédures stockées. Un quatrième objet vient se joindre à ce groupe. Il s'agit des tablespaces. Il existe deux paramètres PostgreSQL permettant d'indiquer le coût d'accès à une page séquentielle et à une page aléatoire (donc en prenant en compte le temps relatif au déplacement de la tête de lecture). Ces deux paramètres sont respectivement seq_page_cost et random_page_cost. Ils dépendent principalement du nombre de tours par minutes (l'acronyme anglais est RPM) du disque, ainsi que de la vitesse de déplacement de la tête de lecture. Or, l'utilisation habituelle des tablespaces est de pouvoir disposer de plusieurs disques pour une même instance. Tous ces disques (ou plutôt système disque) n'ont pas forcément les mêmes performances, il est donc intéressant de pouvoir les différencier. La version 9.0 permet cela en autorisant la configuration de ces deux paramètres par tablespace. Par exemple :

```
postgres=# CREATE TABLESPACE ts1 LOCATION '/opt/postgresql-9.0/ts1';
CREATE TABLESPACE
postgres=# ALTER TABLESPACE ts1 SET (random_page_cost=1.5);
ALTER TABLESPACE
```

Remarquez que la syntaxe n'est pas identique à un « ALTER DATABASE » ou à un « ALTER ROLE ». Vous devez utiliser les parenthèses après la clause SET.

Une limitation des anciennes versions est la façon dont les paramètres étaient appliqués. À la connexion d'un utilisateur u1 sur une base de données b1, la session avait pour paramétrage la configuration du postgresql.conf, surchargé par la configuration spécifique de la base de données b1, elle-même surchargée par la configuration spécifique de l'utilisateur u1. Si la base de données dispose d'une configuration particulière pour le paramètre client_encoding (disons LATIN9) et que l'utilisateur dispose lui-aussi d'une configuration pour ce même paramètre (disons WIN1252), alors la configuration de la session sera par défaut celle de l'utilisateur (donc WIN1252). Si ce même utilisateur se connecte à une autre base de données qui a encore un autre paramétrage pour client_encoding (par exemple UTF-8), c'est toujours celui du client qui gagne. Or, on pourrait vouloir forcer la configuration d'une base de données par rapport à celle de l'utilisateur. Cela devient possible avec la version 9.0. La configuration peut se faire par base de données, par utilisateur ou par paire base de données/utilisateur. Ce dernier utilise uniquement l'ordre SQL « ALTER ROLE ». Voici un exemple complet :

```
postgres=# ALTER DATABASE b1 SET client_encoding TO latin9;
ALTER DATABASE
postgres=# ALTER ROLE a SET client_encoding TO win1252;
ALTER ROLE
postgres=# ALTER DATABASE b2 SET client_encoding TO utf8;
ALTER DATABASE
postgres=# ALTER ROLE a IN DATABASE b2 SET client_encoding TO utf8;
ALTER ROLE
postgres=# \q
```

Là, toute la configuration est faite. Les connexions sur b1 se font avec un client_encoding à LATIN9 (première requête), celles de l'utilisateur a se fait avec un client_encoding à WIN1252 (requête 2) sauf quand ce dernier se connecte à la base b2 (requête 4). Les connexions sur b2 utilisent un client_encoding à UTF-8 (requête 3). Pas de configuration spécifique pour l'utilisateur b. Vérifions tout ça. L'outil psql est utilisé pour se connecter en tant qu'un certain utilisateur (option -U) et pour exécuter une requête SQL qui récupère le nom d'un paramètre, la valeur de ce paramètre et sa source de configuration. L'option -t permet d'éviter l'affichage de l'en-tête (nom des colonnes) et du bas de page (nombre de lignes récupérées par la requête).

```
guillaume@laptop:~$ export REQUETE="SELECT name, setting, source FROM pg_settings WHERE name='client_encoding'"
guillaume@laptop:~$ psql -t -U a -c "$REQUETE" postgres
 client_encoding | win1252 | user
guillaume@laptop:~$ psql -t -U b -c "$REQUETE" postgres
 client_encoding | UTF8    | default
guillaume@laptop:~$ psql -U a -c "$REQUETE" b1
 client_encoding | win1252 | user
guillaume@laptop:~$ psql -U b -c "$REQUETE" b1
 client_encoding | latin9  | database
guillaume@laptop:~$ psql -U a -c "$REQUETE" b2
 client_encoding | utf8    | database user
guillaume@laptop:~$ psql -U b -c "$REQUETE" b2
 client_encoding | utf8    | database
```

Une meilleure traçabilité des modifications de paramètres

Un comportement très gênant lors de la modification de paramètres est qu'il est impossible de savoir après coup les paramètres qui ont été modifiés. Il faut tracer cela soit-même, soit dans un document relatif au serveur, soit en créant différentes versions du fichier de configuration. L'expérience indique que,

même avec la plus grande volonté au monde, c'est assez difficile à respecter dans le long terme.

La version 9.0 solutionne ce problème assez élégamment. Les paramètres modifiés sont tracés dans les journaux applicatifs de PostgreSQL. Il n'y a d'ailleurs rien à activer pour disposer de ce nouveau comportement.

```
guillaume@laptop:~$ echo "work_mem = 3MB
> random_page_cost = 2 " >> /opt/postgresql-9.0/data/postgresql.conf
guillaume@laptop:~$ pg_ctl reload
LOG:  a reçu SIGHUP, rechargement des fichiers de configuration
LOG:  le paramètre « logging_collector » ne peut pas être modifié sans redémarrer le serveur
LOG:  paramètre « work_mem » modifié par « 3MB »
LOG:  paramètre « random_page_cost » modifié par « 2 »
envoi d'un signal au serveur
```

Nous pouvons donc nous apercevoir qu'un petit malin a modifié la configuration de PostgreSQL sans avoir redémarré le serveur (le changement du paramètre logging_collector le réclame, comme pour 31 autres des 202 paramètres) et que les deux paramètres que nous venons de modifier ont bien été pris en compte.

Des paramètres supplémentaires

Quelques nouveaux paramètres apparaissent, comme bonjour pour activer la gestion du protocole Bonjour et enable_material pour activer ou plutôt désactiver la prise en compte d'un algorithme utilisé par le planificateur de requêtes.

Monitoring

Connaître le nom des applications connectées

PostgreSQL dispose d'un grand nombre d'informations sur les connexions en cours grâce à la vue système pg_stat_activity. Néanmoins, si plusieurs connexions émanent de la même adresse IP, il est difficile de savoir quel outil est à l'origine d'une de ces connexions. D'où la possibilité en 9.0 de préciser le nom de l'application qui se connecte. Il existe plusieurs moyens pour indiquer le nom de l'application. Tout d'abord à la connexion en tant que paramètre de connexion. C'est ce que fait pgAdmin lorsqu'il essaie de se connecter à une base de données. Ensuite, il est possible d'utiliser le paramètre de configuration application_name. Ça permet à un outil qui utilise psql de changer le nom de l'application par son nom propre. Par exemple, il est possible d'écrire un script SQL de maintenance ou de mise à jour applicative dont l'une des premières instructions serait « SET application_name TO 'maintenance du soir' ». Le nom de l'application est disponible dans la vue pg_stat_activity :

```
b1=# SELECT procpid, datname, usename, application_name FROM pg_stat_activity;
 procpid | datname  |  usename  |      application_name
---------+----------+-----------+---------------------------
    8073 | b1       | guillaume | psql
    8910 | postgres | guillaume | pgAdmin III - Browser
    8921 | postgres | guillaume | pgAdmin III - Server Status
(3 lignes)

b1=# SET application_name TO 'mon test';
SET
b1=# SELECT procpid, datname, usename, application_name FROM pg_stat_activity;
 procpid | datname  |  usename  |      application_name
---------+----------+-----------+---------------------------
    8073 | b1       | guillaume | mon test
    8910 | postgres | guillaume | pgAdmin III - Browser
    8921 | postgres | guillaume | pgAdmin III - Server Status
(3 lignes)
```

Notez que pgAdmin a été modifié pour afficher cette information dans sa fenêtre d'état du serveur.

Divers

pg_relation_size() et pg_total_relation_size() sont souvent assez mal comprises. Deux nouvelles fonctions ont été ajoutées pour obtenir l'information plus simplement : pg_table_size() et pg_indexes_size(). Le premier indique la taille de la relation indiquée en argument, le second la taille des index de l'argument.

Autres nouvelles fonctions, pg_stat_reset_shared() et pg_stat_reset_single_table_counters(). La première sert à réinitialiser les statisques de la table système pg_stat_bgwriter, la seconde à réinitialiser les statistiques d'une seule table. En effet, tout ce que nous avions auparavant était pg_stat_reset() qui permettait de réinitialiser toutes les statisques des objets spécifiques à une base de données. Donc pas les statistiques partagées comme pg_stat_bgwriter et pas les statistiques d'une table spécifique.

Un nouveau caractère joker est disponible pour le paramètre log_line_prefix. Il s'agit de %e. Ce joker est remplacé par l'état SQL (SQL STATE) tel que défini dans la norme SQL.

Enfin, le schéma information_schema est remis en conformité avec la norme SQL:2008.

Maintenance

Nouvelle implémentation du VACUUM FULL

L'option FULL de l'instruction VACUUM permet de défragmenter une table. Son fonctionnement est assez simple. Il fait une première lecture séquentielle de la table pour détecter tous les espaces libres de cette table. Une fois cette première passe terminée, il fait un deuxième passage en sens inverse et, pour chaque ligne toujours visible, il enregistre cette ligne dans un des espaces vides de la table (de préférence, celui qui a la taille nécessaire en début de table). Une fois qu'il a terminé cette deuxième passe, la majorité des espaces libres se trouve en fin de fichier et ce dernier peut être tronqué rapidement. Cette implémentation est particulièrement lente du fait du déplacement des lignes, sans parler qu'elle atténue considérablement l'efficacité des index en leur ajoutant des pointeurs pour chaque ligne déplacée, au point qu'il est très fortement conseillé de faire un REINDEX tout de suite après.

La version 9.0 change cela par un algorithme qui ne fait qu'une passe : lecture séquentielle des lignes dans le fichier contenant la table, et écriture de

chaque ligne visible dans un autre fichier. Une fois l'opération terminée, l'ancien fichier est supprimé. Le gros avantage est une rapidité bien plus importante et une absence de dégradation des index. Un inconvénient apparaît rapidement. En cas de défragmentation d'une table peu fragmentée, il faudra disposer d'un espace libre sur le disque équivalent à la taille de la table.

Dans une table d'un million de lignes pour lesquelles une sur deux est supprimées, la différence de vitesse d'exécution entre une 8.4 et une 9.0 est flagrante : 4 secondes en 8.4 et 1,3 en 9.0. C'est le résultat direct d'un test effectué sur mon portable pour des lignes ne comportant qu'une seule colonne de type integer. La taille de la table une fois les données insérées est de 36 Mo, imaginez le gain avec une table de plusieurs Go.

De plus, nous pouvons aussi vérifier l'état de l'index. L'installation du module pgstattuple permet, entre autres, de vérifier la densité de l'index grâce à la procédure stockée pgstatindex(). Une vérification de cette densité sur l'index en 8.4 révèle que la densité est de 90% une fois les données insérées (normal, le FILL FACTOR d'un index est par défaut de 90%), de 90% une fois le DELETE exécuté (il n'y a pas d'ajout de lignes dans la table, donc pas de modification de l'index) et de 45% après le VACUUM FULL. En 9.0, cette densité est de 90% après l'INSERT, après le DELETE ainsi qu'après le VACUUM FULL. Autrement dit, il est inutile de faire un REINDEX en 9.0 après un VACUUM FULL. C'est toujours 800 millisecondes en plus de gagner sur la 8.4 (dans le cadre de l'exemple donné plus haut).

Nouvelle option pour l'outil vacuumdb

Les versions précédentes ne permettaient qu'un VACUUM seul ou un VACUUM avec l'option ANALYZE mais pas un ANALYZE seul. Si un administrateur voulait seulement faire un ANALYZE de manière périodique, il n'avait d'autres choix que d'utiliser l'outil psql de cette façon :

```
guillaume@laptop:~$ psql -c "ANALYZE t1;" b1
```

Donc, au département « petites améliorations qui simplifient drôlement la vie », notons que l'outil vacuumdb permet maintenant d'exécuter un ANALYZE seul grâce à l'option –analyze-only.

Performances

Implémentation haute performance de la fonctionnalité LISTEN/NOTIFY

PostgreSQL dispose d'un système de messages entre sessions. Ce système existe depuis longtemps et son fonctionnement est simple mais bien rodé. Prenons deux sessions, la première va être en écoute des messages de la seconde. Cela ne sous-entend pas que la première est bloquée en attente du message car le système est heureusement asynchrone.

Commençons par nous mettre en attente sur la première session :

```
b1=# LISTEN toto;
LISTEN
```

À ce moment-là, nous pouvons toujours exécuter tout type de requête à partir cette session. Par contre, quand la session 2 lancera le message toto :

```
b1=# NOTIFY toto;
NOTIFY
```

Cette information sera conservée et sera fournie à la session 1 dès qu'elle la consultera. Avec psql, il faut exécuter une requête mais d'autres outils vérifient cela très fréquemment au sein de leur boucle de gestion des événements.

```
b1=# SELECT true;
 bool
------
 t
(1 row)
Asynchronous notification "toto" received from server process with PID 32093.
```

Il y avait un gros soucis avec cette implémentation, visible généralement sur les systèmes ayant beaucoup de processus en écoute. La table pg_listener indique la liste de ces processus. S'il y en a beaucoup et s'ils changent fréquemment, la table devient particulièrement grosse et fragmentée. Étant une table système très utilisée, elle devient difficilement maintenable. Par exemple, cela arrive assez fréquemment avec le système de réplication Slony.

La version 9 permet donc de supprimer ce problème en remplaçant cette table par une structure en mémoire. Elle en profite aussi pour ajouter la possibilité de fournir une chaîne d'informations supplémentaires. Cette dernière nouveauté est intéressante pour diminuer le nombre de queues d'attente. En effet, une des utilisations les plus fréquentes de ce système est de permettre à différentes sessions de se faire un cache de certaines tables de type dictionnaire et d'être automatiquement prévenues quand une modification est réalisée sur ces tables. Avec les versions antérieures à la 9.0, il fallait absolument ajouter une queue pour chaque table dictionnaire. Maintenant, une session peut être en écoute des changements et récupérer dans la chaîne d'informations la table qui a été modifiée (cela peut même aller plus loin en précisant ce qui a été changé…). Par exemple, si la session 2 exécute :

```
b1=# NOTIFY toto, 'table t10';
NOTIFY
```

La session 1 récupérera toute cette information :

```
Asynchronous notification "toto" with payload "table t10" received from server process with PID 32093.
```

Suppression des jointures inutiles

La plupart des développeurs d'applications clientes passent maintenant par des outils de développement rapide capables de se connecter à des bases des données, d'y ajouter des informations et de les en extraire. Ces outils sont souvent appelées des ORM (Object Relational Mapping).

Le gros avantage de ces outils, c'est la facilité et la rapidité de programmation. Cela va évidemment de pair avec quelques inconvénients, dont par exemple des requêtes créées par l'outil souvent assez mal écrites. Malgré quelques progrès dans ce domaine, il n'empêche que beaucoup ont tendance à créer des requêtes inutilement grosses. Notamment avec des jointures parfois inutiles.

Un travail important a été effectué au niveau du planificateur pour améliorer les performances avec ce type d'outils. Le travail de Robert Haas est assez caractéristique dans ce domaine, avec son patch permettant de supprimer les jointures inutiles.

Créons trois tables et ajoutons-y quelques données :

```
b7=# CREATE TABLE t1 (a int);
CREATE TABLE
```

```
b7=# CREATE TABLE t2 (b int);
CREATE TABLE
b7=# CREATE TABLE t3 (c int);
CREATE TABLE
b7=# INSERT INTO t1 SELECT generate_series(1, 100000);
INSERT 0 100000
b7=# INSERT INTO t2 SELECT generate_series(1, 100000);
INSERT 0 100000
b7=# INSERT INTO t3 SELECT generate_series(1, 100000);
INSERT 0 100000
```

Puis nous exécutons notre requête avec jointure :

```
b1=# 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
-------------------------------------------------------------------------------
 Hash Left Join  (cost=6168.00..13957.00 rows=100000 width=8)
   Hash Cond: (t1.a = t3.c)
   ->  Hash Join  (cost=3084.00..7700.00 rows=100000 width=8)
         Hash Cond: (t1.a = t2.b)
         ->  Seq Scan on t1  (cost=0.00..1443.00 rows=100000 width=4)
         ->  Hash  (cost=1443.00..1443.00 rows=100000 width=4)
               ->  Seq Scan on t2  (cost=0.00..1443.00 rows=100000 width=4)
   ->  Hash  (cost=1443.00..1443.00 rows=100000 width=4)
         ->  Seq Scan on t3  (cost=0.00..1443.00 rows=100000 width=4)
(9 rows)
```

Nous voyons que les trois tables sont parcourues séquentiellement et que les jointures sont réalisées grâce à un Hash Join. Autrement dit, le même comportement qu'en 8.4. Cependant, si nous ajoutons une contrainte unique sur la colonne c de la table t3, la jointure avec la table t3 devient inutile. En effet, comme nous ne récupérons aucune colonne de la table t3 dans la partie SELECT et que le nombre d'enregistrements ne sera pas modifié par la jointure (c'est un LEFT JOIN), alors le résultat ne changera pas, que la jointure soit présente ou non. PostgreSQL 9.0 est capable de détecter un tel cas et de supprimer la jointure inutile :

```
b1=# alter table t3 add constraint c1 unique (c);
NOTICE:  ALTER TABLE / ADD UNIQUE will create implicit index "c1" for table "t3"
ALTER TABLE
b1=# 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
-------------------------------------------------------------------------------
 Hash Join  (cost=3084.00..7700.00 rows=100000 width=8)
   Hash Cond: (t1.a = t2.b)
   ->  Seq Scan on t1  (cost=0.00..1443.00 rows=100000 width=4)
   ->  Hash  (cost=1443.00..1443.00 rows=100000 width=4)
         ->  Seq Scan on t2  (cost=0.00..1443.00 rows=100000 width=4)
(5 rows)
```

Si nous avions utilisé l'option ANALYZE d'EXPLAIN, nous nous serions rendu compte que le gain est appréciable. Je passe de 288 millisecondes à 147 millisecondes sur mon portable. Pratiquement une division par deux. Cela ne sera pas toujours aussi important mais cette optimisation devrait se révéler rentable pour les utilisateurs d'ORM. Cela sera aussi être très utile pour les utilisateurs de vues. En effet, une association de plusieurs vues peut occasionner une requête finale utilisant une jointure inutile.

Le plus important dans tout ça, c'est de voir que la déclaration de contraintes dans la base n'est pas utile uniquement à s'assurer de la qualité des données (au cas où ça ne suffirait pas pour vous convaincre de les utiliser). C'est aussi un excellent moyen pour donner des pistes d'optimisations au planificateur de requêtes. Amélioration de la clause IS NOT NULL

Une des améliorations essentielles de la version 9.0 pour les index est de permettre leur utilisation lorsque la requête doit exécuter une clause IS NOT NULL.

Voici un script qui permet de créer une table, de lui ajouter un grand nombre de lignes tout en s'assurant que la majorité d'entre elles valent NULL :

```
CREATE TABLE t4 (c1 integer);
INSERT INTO t4 SELECT generate_series(1, 1000000);
UPDATE t4 SET c1=NULL WHERE c1%3!=2;
CREATE INDEX i4 ON t4(c1);
```

Après avoir exécuté ce script sur une version 8.4, voici le plan d'exécution d'une requête assez simple :

```
b1=# EXPLAIN ANALYZE SELECT max(c1) from t4;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Result  (cost=0.05..0.06 rows=1 width=0) (actual time=618.648..618.649 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.05 rows=1 width=4) (actual time=618.642..618.642 rows=1 loops=1)
           ->  Index Scan Backward using i4 on t4  (cost=0.00..48892.36 rows=1000000 width=4) (actual time=618.639..618.639 rows=1 loops=1)
                 Filter: (c1 IS NOT NULL)
 Total runtime: 618.693 ms
(6 rows)
```

Prenons la même requête sur une version 9.0 :

```
b7=# EXPLAIN ANALYZE SELECT max(c1) from t4;
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Result  (cost=0.03..0.04 rows=1 width=0) (actual time=0.083..0.083 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.03 rows=1 width=4) (actual time=0.078..0.078 rows=1 loops=1)
           ->  Index Scan Backward using i4 on t4  (cost=0.00..26219.30 rows=1000000 width=4) (actual time=0.076..0.076 rows=1 loops=1)
                 Index Cond: (c1 IS NOT NULL)
 Total runtime: 0.133 ms
(6 rows)
```

6000 fois plus rapide. La différence se situe au niveau de la ligne contenant le texte « c1 IS NOT NULL ». Dans le cas de la version 8.4, PostgreSQL est

obligé de lire l'index en entier et de filtrer lui-même la clause IS NOT NULL. En version 9.0, PostgreSQL parcourt partiellement l'index en utilisant la condition IS NOT NULL.

Nouveautés sur EXPLAIN

Pour permettre l'ajout de nouvelles options, la commande EXPLAIN a changé de syntaxe. Maintenant, toutes les options sont à mettre entre parenthèses après le mot-clé EXPLAIN. Cela étant dit, l'ancienne syntaxe est toujours supportée mais n'accepte que les anciennes options.

Il existe quatre options en tout :

- ANALYZE, permettant d'exécuter réellement la requête pour obtenir les coûts réels de chaque nœud du plan ;
- BUFFERS, permettant d'afficher le nombre de blocs lus dans le cache de PostgreSQL et hors de ce cache pour chaque nœud du plan ;
- COST, permettant d'afficher ou non les coûts de chaque nœud du plan ;
- FORMAT, permettant de sélectionner un autre format de sortie (entre texte, XML, JSON, YAML) ;
- VERBOSE, pour obtenir encore plus d'informations.

De toutes ces options, seules BUFFERS et FORMAT sont vraiment nouvelles. Voici un exemple avec ANALYZE, BUFFERS et COST (ce dernier est implicite) :

```
b7=# EXPLAIN (ANALYZE, BUFFERS) SELECT t1.a,t2.b FROM t1 JOIN t2 ON (t1.a=t2.b) LEFT JOIN t3 ON (t1.a=t3.c);
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=3084.00..7700.00 rows=100000 width=8) (actual time=38.491..133.852 rows=100000 loops=1)
   Hash Cond: (t1.a = t2.b)
   Buffers: shared hit=886, temp read=296 written=294
   ->  Seq Scan on t1  (cost=0.00..1443.00 rows=100000 width=4) (actual time=0.013..11.423 rows=100000 loops=1)
         Buffers: shared hit=443
   ->  Hash  (cost=1443.00..1443.00 rows=100000 width=4) (actual time=38.442..38.442 rows=100000 loops=1)
         Buckets: 8192  Batches: 2  Memory Usage: 1760kB
         Buffers: shared hit=443, temp written=146
         ->  Seq Scan on t2  (cost=0.00..1443.00 rows=100000 width=4) (actual time=0.009..15.146 rows=100000 loops=1)
               Buffers: shared hit=443
 Total runtime: 138.734 ms
(11 rows)
```

La ligne Buckets est nouvelle elle-aussi. Elle indique le nombre de jetons utilisés ainsi que la mémoire prise pour le hachage.

Les différents formats possibles ont pour but de faciliter la vie aux développeurs d'applications externes récupérant un EXPLAIN. Le résultat est plus facilement analysable par un programme.

Diverses améliorations

L'instruction ALTER TABLE a été améliorée pour éviter son enregistrement dans les journaux de transactions lorsqu'une réécriture de la table est nécessaire. De même, quand une table a été créé puis tronquée dans la même transaction, toute une optimisation a eu lieu pour les informations écrites dans les journaux de transactions.

Dans certains cas, le planificateur a du mal à calculer le pourcentage de valeurs distinctes. Il a donc été ajouté la possibilité de forcer cette valeur. Attention, en cas d'erreur dans le paramétrage, cela peut se révéler être vraiment contre-performant.

Sécurité

Au niveau sécurité, PostgreSQL dispose déjà d'un grand nombre de fonctionnalités. Pour les droits, il est déjà possible de donner ou supprimer des droits pour les différents rôles sur pratiquement tous les objets. Il existait deux manques importants.

Donner des droits à tous les objets d'un schéma

Dans les versions précédentes, on devait forcément donner les droits objet par objet, ce qui peut se révéler fastidieux si vous devez donner des droits à 400 tables par exemple. Arrive donc la commande GRANT IN SCHEMA. Son but est de donner des droits à tous les objets d'un schéma spécifié. Par exemple :

```
GRANT SELECT ON ALL TABLES IN SCHEMA s1;
```

permet de donner le droit SELECT à l'utilisateur u1 pour toutes les tables du schéma s1. Donc, il n'est plus nécessaire d'exécuter une requête SQL pour chaque table de la base de données, mais il faudra en exécuter une pour chaque schéma de cette base de données, ce qui facilite déjà grandement le travail. De plus, cela fonctionne pour d'autre types d'objets que les tables : vues, procédures stockées, séquences sont aussi concernées par cette nouvelle clause.

Configurer des droits par défaut

L'autre soucis majeur pour la gestion des droits est qu'il faut penser à eux à chaque création d'objets. La version 9.0 ajoute la possibilité d'indiquer des droits par défaut pour tous les objets nouvellement créés. Il est possible de les déclarer pour tous les utilisateurs ou seulement pour certains utilisateurs. Il est possible de le faire pour tous les objets d'une base ou seulement pour les objets d'un schéma. Prenons un exemple :

```
b1=# GRANT b TO a;
GRANT ROLE
b1=# \c b1 a
You are now connected to database "b1" as user "a".
b1=> CREATE TABLE t5 (c1 integer);
CREATE TABLE
b1=> \c b1 b
You are now connected to database "b1" as user "b".
b1=> SELECT * FROM t5;
ERROR:  permission denied for relation t5
b1=> \c b1 a
You are now connected to database "b1" as user "a".
b1=> ALTER DEFAULT PRIVILEGES GRANT SELECT ON TABLES TO b;
ALTER DEFAULT PRIVILEGES
```

```
b1=> CREATE TABLE t6 (c1 integer);
CREATE TABLE
b1=> \c b1 b
You are now connected to database "b1" as user "b".
b1=> SELECT * FROM t6;
 c1
----
(0 rows)
```

Un des points à bien comprendre, c'est que le rôle qui crée l'objet doit avoir pour membre le rôle qui va obtenir les droits (d'où le GRANT au tout début de la démonstration ci-dessus).

Des droits pour les Large Objects

Les Large Objects sont des objets binaires particuliers au niveau de PostgreSQL. Historiquement, il n'y a jamais eu de gestion de droits pour eux. Ce manque a été remarqué avec tout le travail effectué autour de SE-PostgreSQL et a fait l'objet d'un patch qui a été intégré. Ce patch ajoute la gestion des Large Objects par les commandes GRANT et REVOKE. Voici le détail pour l'ajout de droits :

```
GRANT { { SELECT | UPDATE } [,...] | ALL [ PRIVILEGES ] }
   ON LARGE OBJECT loid [, ...]
   TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

Vérification de la solidité des mots de passe

Les mots de passe stockés en interne ne sont pas testés. Rien ne cherche à forcer leur validité, y compris pour des règles basiques (comme ne pas avoir un mot de passe identique au nom de l'utilisateur ou ne pas avoir un mot de passe qui fait moins de X caractères). Généralement, quand on cherche à avoir cette fonctionnalité, la première réponse est d'utiliser une authentification externe comme LDAP ou GSSAPI qui permettent ce genre de contrôle.

En 9.0, un hook a été ajouté pour permettre à une bibliothèque chargée de valider un mot de passe. Cela ne fonctionne que si le mot de passe fourni est en clair car c'est le seul moyen de faire certains tests (comme la longueur). Ça a évidemment un impact négatif sur la sécurité du coup. Évidemment, avec un mot de passe chiffré, il est malgré tout possible de vérifier certains critères comme le fait de ne pas avoir un mot de passe identique au nom de l'utilisateur, mais les tests seront beaucoup moins nombreux, et du coup la vérification sera moins efficace.

Un module contrib est fourni pour montrer la programmation d'un tel outil. Il s'appelle passwordcheck et teste quelques critères : taille du mot de passe, s'il contient le nom de l'utilisateur, s'il contient des lettres et d'autres caractères. Si vous souhaitez utiliser ce module, il est très fortement suggérer de le modifier pour qu'il corresponde mieux à vos besoins (par exemple, la taille minimum d'un mot de passe, l'activation de l'utilisation de cracklib, etc). Il faut vraiment considérer ce module comme un exemple de ce qui est possible de faire et non pas comme un outil prêt à l'emploi.

Divers

Toujours dans le domaine de la sécurité, d'autres possibilités ont été ajoutées. Une méthode d'authentification a été ajoutée pour gérer le protocole Radius. L'authentification LDAP se voit ajouter le support du mode search/bind.

Deux nouveaux mots clés sont disponibles pour le fichier pg_hba.conf : samehost et samenet.

Windows

Enfin une version 64 bits native

PostgreSQL est disponible sous Windows depuis la version 8.0 (cela fait donc 5 ans). PostgreSQL était disponible uniquement en 32 bits. C'est bien sûr utilisable sur des plateformes 64 bits mais la partie serveur ne peut pas bénéficier de l'apport principal du 64 bits, à savoir l'adressage mémoire beaucoup plus important. En effet, au lieu d'être bloqué à 4 Go en 32 bits, le 64 bits permet d'adresser beaucoup plus de mémoire.

Avec une version de Windows soigneusement choisie, la version 9.0 pourra être installée sur un serveur 64 bits et bénéficier de cet espace d'adressage plus important. L'intérêt n'est pas spécialement pour le cache disque de PostgreSQL car la gestion de la mémoire partagée sous Windows est bien moins efficace que sous Unix, ce qui fait qu'on reste toujours bloqué à une valeur inférieure à 1 Go pour ce paramètre (plus exactement, vous pouvez mettre plus mais généralement les utilisateurs constatent une baisse des performances). Du coup, les processus serveur vont pouvoir se voir allouer beaucoup plus de mémoire de travail. Cela correspond aux paramètres work_mem et maintenance_work_mem.

De toute façon, malgré une utilisation plus importante de la mémoire, un serveur PostgreSQL sous Windows sera généralement plus lent que son équivalent sous Linux.

UTF-16 utilisé pour les traces

Autre amélioration au niveau de Windows, apportée par des développeurs japonais, les traces de PostgreSQL sous Windows peuvent être enregistrées au format UTF-16.

Modules contrib

Au niveau des modules contrib, on constate trois nouveaux modules et quatre modules bien améliorés. Nous avons déjà parlé de passwordcheck et de unaccent. Voyons le reste.

Faciliter les migrations de versions

Avec PostgreSQL, il existait auparavant deux manières de mettre à jour vers une nouvelle version majeure. La première consistait à sauvegarder chaque base de données, créer le nouveau répertoire des données, puis tout restaurer. Cette méthode, bien que simple, ne faisait pas que des heureux. Sauvegarder et restaurer 200 Go de données prend du temps. Beaucoup de temps. Ce qui rendait l'opération impossible pour certains.

La deuxième manière consiste à mettre en place un deuxième serveur contenant la nouvelle version de PostgreSQL et de répliquer les données entre le serveur contenant l'ancienne version (qui est donc le maître) et celui contenant la nouvelle (en tant qu'esclave) en utilisant un outil comme Slony. Dans ce cas, l'ancien serveur est toujours opérationnel pendant le transfert des données. Et une fois les données complètement synchronisées entre les deux serveurs, il ne reste plus qu'à basculer maître et esclave. C'est un processus un peu plus compliqué à mettre en place mais ça se fait malgré tout facilement.

Le gros avantage est que l'interruption de production est de quelques minutes. Le gros inconvénient est qu'il faut disposer d'un deuxième serveur disposant d'autant d'espace disque que l'ancien. C'est un problème qui peut vite se révéler très gênant car la majorité des serveurs qui seront mis à jour ainsi le seront parce que justement leur base est trop grosse pour le faire avec une opération de sauvegarde/restauration.

Vient donc pg_upgrade. Bruce Momjian, une des personnes de la Core Team de PostgreSQL, a décidé de coder cet outil depuis quelques temps déjà. Il était déjà disponible à la sortie de la 8.4, sous le nom de pg_migrator. Toute la phase de développement de la 9.0 a tenu compte de l'existence de cet outil. Des modifications ont été faites au niveau de PostgreSQL pour faciliter la vie de cet outil et pour permettre son intégration dans le code de PostgreSQL, sans parler des nombreuses corrections de bugs qui ont pu avoir lieu. Le gros intérêt de cet outil est qu'il ne va pas toucher aux fichiers contenant les tables utilisateurs vu qu'elles changent très rarement de structure entre version majeure. Il va travailler sur les catalogues systèmes, qui eux changent fréquemment mais sont peu impactés par la taille réelle de la base. Du coup, la migration d'un serveur contenant une base de 10 Mo et celle d'un serveur contenant une base de 10 To devrait se faire pratiquement aussi rapidement. Sur un exemple pris par Bruce Momjian lors d'une conférence donnée à San Francisco pour PGWest 2010, la migration d'une base de données de 150 Go contenant 850 tables a duré 5 heures avec la méthode traditionnelle (sauvegarde et restauration) alors qu'elle n'a pris que 44 secondes dans le meilleur cas avec pg_upgrade.

Gageons que cet outil va être de plus en plus utilisé.

Améliorations apportées d'autres modules

Plusieurs autres modules ont bénéficié d'améliorations.

Pgbench dispose maintenant d'une option (-j) pour réaliser du multithreading. La simulation de plusieurs clients en est facilitée.

Le module auto_explain, en plus de tracer un EXPLAIN automatique sur les requêtes en cours d'exécution, trace aussi la requête.

pg_stat_statement récupère plus d'informations, précisant ainsi le nombre de lectures dans le cache de PostgreSQL et hors du cache pour chaque requête récupérée dans la vue.

Quant au module hstore, il a bénéficié de tant d'améliorations qu'il serait difficile de les indiquer toutes ici.

Et la suite ?

La prochaine version sera la 9.1. Beaucoup de travail a déjà eu lieu sur cette version. Les grosses nouveautés devraient concerner les thèmes suivants :

- la réplication synchrone (pour rappel, la réplication interne de PostgreSQL est une réplication asynchrone) ;
- une réplication plus facile à mettre en place et à administrer (actuellement, mettre en place et surtout administrer la réplication interne demande une bonne connaissance de PostgreSQL) ;
- la suite de l'implémentation de SEPostgreSQL (une version encore plus sécurisée de PostgreSQL) ;
- le support de tables non tracées dans les journaux de transactions ;
- une gestion simple des extensions.

Conclusion

La version 9.0 mérite son numéro. L'ajout de la réplication en fait une version à part. Mais ce n'est pas la seule raison qui peut vous pousser à passer à cette version-là. Les autres nouveautés couvrent des domaines très étendues : plus de performance, plus de fonctionnalités utilisateurs et développeurs, une meilleure administration. De quoi faire plaisir à tous les profiles d'utilisateurs.

Si vous êtes intéressés par plus d'informations sur PostgreSQL et sur cette nouvelle version, je ne pourrais trop vous conseiller de venir à la première Session PostgreSQL. Elle aura lieu le 4 février 2011, à La Cantine. Vous trouverez plus d'informations sur le site des sessions (http://www.postgresql-sessions.org/). De plus, la communauté PostgreSQL organise des conférences au FOSDEM, qui aura lieu du 5 au 6 février 2011. Là-aussi, n'hésitez pas à venir. Le planning des conférences n'est pas encore connue mais il devrait être d'aussi bonne qualité que les années précédentes.