Table des matières

PostgreSQL 8.4

Cet article, écrit par Guillaume Lelarge, a été publié dans le hors-série 44 du magazine GNU/Linux Magazine France, hors-série dédié à PostgreSQL. Il est disponible maintenant sous licence Creative Commons.

Plus d'un an après la sortie de la version 8.3, les développeurs de PostgreSQL ont sorti la version 8.4. Très exactement le 1er juillet. Les plus attentifs remarqueront que le planning initial a été largement débordé, de quatre mois au minimum. Ceci est dû à l'attente de deux patchs ajoutant des fonctionnalités très intéressantes à la réplication de PostgreSQL (esclave en lecture seule, réplication à la transaction prêt). Ces derniers n'étant manifestement pas prêts, il a été jugé préférable de les repousser à une prochaine version. Néanmoins, que cela ne vous refroidisse pas trop. La version 8.4 apporte un très grand nombre de nouvelles fonctionnalités, et des performances encore améliorées. Même si elles ont souvent pour cible l'administrateur, les utilisateurs et les développeurs y trouveront aussi leur compte.

Petit aparté : Tous les exemples ci-dessous ont été réalisés à partir de la base de données de fluxbb, le moteur de forums utilisé sur http://forums.postgresql.fr. Les noms des utilisateurs du forum ont été rendus anonymes.

Pour l'utilisateur

Il y a deux grosses nouveautés pour les utilisateurs : les fonctions WINDOW et la clause WITH. Mais cela ne s'arrête pas là, comme nous allons le voir.

Fonctions WINDOW

Introduite avec la norme SQL:2003 et améliorée avec la norme SQL:2008, cette fonctionnalité se trouvait supportée par les grands noms des bases de données propriétaires : Oracle, SQL Server et Sybase. Aucun SGBD libre ne le proposait… jusqu'à maintenant, grâce au développement d'Hitoshi Harada de la société FORCIA.

Cette fonctionnalité similaire aux agrégats est trop complexe pour la résumer en une phrase. Disons qu'elle permet de faire des opérations d'agrégats sur des sous-ensembles du résultat. Prenons immédiatement un exemple. Fluxbb dispose d'une table appelée fbb_posts, contenant l'ensemble des messages enregistrés sur tous les forums. Si je souhaite connaître le nombre de messages postés par chaque utilisateur, je vais utiliser la fonction d'agrégat `count()` avec la clause `GROUP BY` sur le nom de l'utilisateur :

```
fluxbb=# SELECT poster, count(*) FROM fbb_posts
fluxbb-# GROUP BY 1 ORDER BY 2 DESC;
 u3    |  515
 u312  |   97
 u310  |   58
 u20   |   43
 u130  |   35
[…]
(199 lignes)
```

Rien de nouveau jusque là. Disons maintenant que je veux avoir le pourcentage de messages par utilisateur. Dans ce cas, il faut que je connaisse le nombre de messages postés par un utilisateur, mais aussi le nombre de messages postés par tous les utilisateurs. Il est possible de s'en sortir avec une sous-requête :

```
SELECT poster,
  count(*), 100.0*count(*)/(SELECT count(*) FROM fbb_posts)
FROM fbb_posts
GROUP BY 1 ORDER BY 2 DESC;
```

Le problème de cette requête est que PostgreSQL doit faire deux parcours séquentiels de fbb_posts. Utilisons maintenant une requête window :

```
fluxbb=# SELECT poster,
fluxbb-#   count(*), 100.0*count(*)/(sum(count(*)) OVER ())
fluxbb-# FROM fbb_posts
fluxbb-# GROUP BY 1 ORDER BY 2 DESC;
 poster | count |        ?column?
--------+-------+------------------------
 u3     |   515 |    30.9867629362214200
 u312   |    97 |     5.8363417569193742
 u310   |    58 |     3.4897713598074609
 u20    |    43 |     2.5872442839951865
 u130   |    35 |     2.1058965102286402
[…]
(199 lignes)
```

C'est moins compréhensible à première vue, mais ça a déjà l'avantage de ne demander qu'un seul parcours séquentiel. La clause d'une fonction window s'écrit de la façon suivante : `fonction() OVER (partition frame)`

La fonction peut être une fonction d'agrégat ou une fonction window, quelle soit interne ou créée par l'utilisateur. La partition est un ensemble de lignes fixes qui se déclare avec la clause `PARTITION BY`. La partition par défaut est la table entière (ce qui est le cas dans l'exemple ci-dessus). Quant au frame, il s'agit d'un ensemble de lignes mouvantes défini dans la partition. Les lignes peuvent se déplacer dans la partition pour établir un certain ordre, mais pas hors de cette partition. La clause `ORDER BY` permet la définition d'un frame.

Si on reprend notre exemple, la clause était la suivante : `sum(count(*)) over ()`. La fonction est donc `sum()` sur `count(*)`. On va donc additionner le résultat des `count(*)` de chaque ligne renvoyée. La clause `over` est vide, ce qui indique qu'il n'y a pas de frame, mais que la partition est la table entière. Ici, par table, on signifie le résultat de la requête :

```
SELECT poster, count(*)
FROM fbb_posts
GROUP BY 1 ORDER BY 2 DESC;
```

Autrement dit la requête exécutée sans la partie window. L'exemple `sum(count(*)) over ()` renvoie donc la somme des champs `count(*)` pour la table de travail représentée par la requête ci-dessus.

Compliquons un peu l'exemple avec une clause over non vide. Disons que je veuille récupérer le nombre de messages envoyés par chaque utilisateur, avec un décompte par mois. C'est là-aussi un simple calcul d'agrégat :

```
fluxbb=# SELECT poster, to_yyyymm(posted), count(*)
fluxbb-# FROM fbb_posts
fluxbb-# GROUP BY 1, 2
fluxbb-# ORDER BY 2, 3;
 poster | to_yyyymm | count
--------+-----------+-------
 u42    | 2008-09   |     1
 u23    | 2008-09   |     1
[...]
 u16    | 2008-09   |     5
 u3     | 2008-09   |    15
 u139   | 2008-10   |     1
 u28    | 2008-10   |     1
[...]
 u77    | 2008-10   |     1
[...]
(305 lignes)
```

Note : La fonction `to_yyyymm()` n'existe pas, c'est une fonction utilisateur. Fluxbb enregistre les dates et heures au format epoch. J'ai donc ajouté cette fonction avec l'ordre suivant :

```
CREATE OR REPLACE FUNCTION to_yyyymm(integer)
RETURNS text
LANGUAGE sql
IMMUTABLE
AS $$
SELECT to_char('1970-01-01 00:00:00-00'::timestamp + ($1 || ' seconds')::interval, 'YYYY-MM')
$$;
```

Maintenant, si je veux aussi avoir le pourcentage calculé auparavant, mais cette fois pas pour la totalité des messages, mais mois par mois, on comprend bien qu'il va falloir faire des partitions par mois, donc suivant le résultat de la fonction to_yyyymm(). La requête va être transformée ainsi :

```
fluxbb=# SELECT poster, to_yyyymm(posted), count(*) AS message_par_mois,
fluxbb-#   100*count(*)/(sum(count(*)) over (PARTITION BY to_yyyymm(posted)))
fluxbb-#   AS pourcentage_par_mois
fluxbb-# FROM fbb_posts
fluxbb-# GROUP BY 1, 2
fluxbb-# ORDER BY 2, 3;
 poster | to_yyyymm | message_par_mois |  pourcentage_par_mois
--------+-----------+------------------+------------------------
 u23    | 2008-09   |                1 |     2.1739130434782609
 u42    | 2008-09   |                1 |     2.1739130434782609
 u5     | 2008-09   |                1 |     2.1739130434782609
 u40    | 2008-09   |                2 |     4.3478260869565217
 u28    | 2008-09   |                2 |     4.3478260869565217
 u20    | 2008-09   |                2 |     4.3478260869565217
 u30    | 2008-09   |                3 |     6.5217391304347826
 u17    | 2008-09   |                3 |     6.5217391304347826
 u19    | 2008-09   |                3 |     6.5217391304347826
 u32    | 2008-09   |                3 |     6.5217391304347826
 u16    | 2008-09   |                5 |    10.8695652173913043
 u22    | 2008-09   |                5 |    10.8695652173913043
 u3     | 2008-09   |               15 |    32.6086956521739130
 u30    | 2008-10   |                1 | 0.46948356807511737089
 u139   | 2008-10   |                1 | 0.46948356807511737089
 u155   | 2008-10   |                1 | 0.46948356807511737089
[...]
(305 lignes)
```

Une clause WINDOW est aussi disponible pour pouvoir utiliser plusieurs fois dans la même requête la définition complexe d'une partition et d'une frame jointe. La requête précédente aurait pu s'écrire :

```
SELECT poster, to_yyyymm(posted), count(*) AS message_par_mois,
  100*count(*)/(sum(count(*)) over w) AS pourcentage_par_mois
FROM fbb_posts
GROUP BY 1, 2
WINDOW w AS (PARTITION BY to_yyyymm(posted))
ORDER BY 2, 3;
```

Je n'ai utilisé que des fonctions d'agrégats mais il faut savoir qu'il existe de nombreuses autres fonctions de type window : `row_number()`, `rank()`, dense_rank(), percent_rank(), cume_dist(), ntile(), lag(), lead(), first_value(), last_value(), nth_value(). Certaines sont très faciles à comprendre, d'autres moins. Le manuel de PostgreSQL les explique en détails et je ne saurais que trop vous conseiller de regarder la première partie de la vidéo de la conférence d'Hitoshi Harada et de David Fetter, présentée au PGCon 2009 (http://hosting3.epresence.tv/fosslc/1/watch/130.aspx), où tout ceci est très clairement expliqué.

Clause WITH

Plus connue sous le nom de « Common Table Expression », cette clause permet dans une même requête de définir une sous-requête nommée qui sera référencée dans d'autres clauses de la requête. Non seulement cela supprime l'utilisation de tables temporaires pour certaines opérations, mais cela permet en plus de créer des requêtes récursives qui peuvent parcourir un arbre ou une liste chaînée de façon particulièrement efficace.

Prenons un exemple. La requête ci-dessus me donne tous les utilisateurs qui ont posté sur le forum avec le pourcentage des messages par mois de chaque utilisateur. Disons que je souhaite filtrer sur un ou plusieurs utilisateurs. Avant la 8.4, je pouvais le faire en déclarant une vue et en utilisant cette vue avec un filtre sur l'utilisateur souhaité. Cela nécessite le droit de créer des objets. Je pouvais aussi faire une sous-requête du style :

```
SELECT *
FROM
(SELECT poster, to_yyyymm(posted), count(*) AS message_par_mois,
  100*count(*)/(sum(count(*)) over w) AS pourcentage_par_mois
FROM fbb_posts
GROUP BY 1, 2
WINDOW w AS (PARTITION BY to_yyyymm(posted))
ORDER BY 2, 3) AS stats
WHERE poster='u3';
```

Ça se fait. Ce n'est pas très lisible, mais c'est utilisable. Si vous devez réutiliser la même requête plusieurs fois, ça devient beaucoup moins lisible et encore moins maintenable. De plus, vous exécuterez autant de fois la sous-requête qu'elle sera mentionnée. Arrive donc la fonctionnalité CTE. La requête ci-dessus s'écrit ainsi :

```
WITH stats AS
(SELECT poster, to_yyyymm(posted), count(*) AS message_par_mois,
  100*count(*)/(sum(count(*)) over w) AS pourcentage_par_mois
FROM fbb_posts
GROUP BY 1, 2
WINDOW w AS (PARTITION BY to_yyyymm(posted))
ORDER BY 2, 3)
SELECT * FROM stats WHERE poster='u3';
```

C'est plus lisible. Ça évite d'être bloqué si vous ne pouvez pas créer de tables temporaires ou de vues. C'est aussi plus efficace si vous voulez faire des jointures, vu que vous pouvez utiliser le nom stats comme celui de n'importe quel autre table. Disons par exemple que je veux les statistiques de u3 et u4 :

```
WITH stats AS
(SELECT poster, to_yyyymm(posted), count(*) AS message_par_mois,
  100*count(*)/(sum(count(*)) over w) AS pourcentage_par_mois
FROM fbb_posts
GROUP BY 1, 2
WINDOW w AS (PARTITION BY to_yyyymm(posted))
ORDER BY 2, 3)
SELECT * FROM stats WHERE poster='u3'
UNION ALL
SELECT * FROM stats WHERE poster='u4';
```

L'exemple est mauvais, je vous l'accorde volontiers. Il aurait été beaucoup plus simple d'utiliser l'opérateur IN. Néanmoins, cela montre que l'on peut utiliser le nom de la table virtuelle stats autant de fois qu'on le souhaite.

En résumé, cette fonctionnalité développée par Yoshiyuki Asaba permet :

- d'utiliser une table temporaire non coûteuse (par exemple une énumération) ;
- d'utiliser une définition de table par les utilisateurs qui n'ont pas le droit de créer des objets comme une table temporaire ou une vue.

Par exemple :

```
fluxbb=# WITH stats AS
(SELECT poster, to_yyyymm(posted), count(*) AS message_par_mois,
  100*count(*)/(sum(count(*)) over w) AS pourcentage_par_mois
FROM fbb_posts
GROUP BY 1, 2
WINDOW w AS (PARTITION BY to_yyyymm(posted))
ORDER BY 2, 3),
groupes (id, titre) AS (VALUES (1, 'administrateur'), (2, 'modérateurs'), (3, 'invité'), (4, 'membre'))
SELECT groupes.titre, stats.*
FROM stats, fbb_users, groupes
WHERE message_par_mois>10
  AND poster=username AND group_id=groupes.id;
     titre      | poster | to_yyyymm | message_par_mois | pourcentage_par_mois
----------------+--------+-----------+------------------+----------------------
 administrateur | u3     | 2008-09   |               15 |  32.6086956521739130
 membre         | u20    | 2008-10   |               11 |   5.1643192488262911
 membre         | u107   | 2008-10   |               13 |   6.1032863849765258
 membre         | u130   | 2008-10   |               16 |   7.5117370892018779
 administrateur | u3     | 2008-10   |               67 |  31.4553990610328638
 administrateur | u3     | 2008-11   |               24 |  27.2727272727272727
```

WITH garantit en plus que la requête ne sera évaluée qu'une fois, ce qui nous permet d'avoir de bonnes performances et le même ensemble de données, mais ce qui interdit aussi d'utiliser des index pour cette requête.

L'autre gros intérêt de cette clause est de permettre l'exécution de requêtes récursives. Pour cela, il faut ajouter le mot clé RECURSIVE à la clause WITH. La requête WITH RECURSIVE est composée de deux parties : la base et la récursion, les deux jointes par un UNION ALL. La partie base initie le travail récursif, la partie récursion sera exécutée autant que fois que la récursion l'exigera. Du coup, attention aux récursions infinies. Ce type de requête est surtout intéressant pour les données de type listes, arbres, graphes.

Nouveau module contrib

La version 8.4 apporte plusieurs modules contrib, dont un à destination des utilisateurs : citext.

Ce module a pour but de permettre l'utilisation d'un champ texte insensible à la casse. Autrement dit, il est possible d'ajouter n'importe quel texte dans ce champ, la recherche ne prendra pas en compte la casse des caractères.

Pour notre exemple, prenons en considération la table des groupes de Fluxbb. fbb_groups contient une colonne indiquant le titre du groupe, g_title :

```
fluxbb=# SELECT g_id, g_title FROM fbb_groups;
 g_id |    g_title
```

```
------+---------------
    1 | Administrators
    2 | Moderators
    3 | Guest
    4 | Members
(4 lignes)
```

Si j'essaie de récupérer les lignes qui commencent par un m minuscule, je n'obtiendrais aucune réponse, alors que pour les lignes avec un M majuscule, j'obtiendrais deux lignes :

```
fluxbb=# SELECT g_id, g_title FROM fbb_groups WHERE g_title LIKE 'm%';
 g_id | g_title
------+---------
(0 ligne)

fluxbb=# SELECT g_id, g_title FROM fbb_groups WHERE g_title LIKE 'M%';
 g_id |  g_title
------+------------
    2 | Moderators
    4 | Members
(2 lignes)
```

Ajoutons maintenant une colonne de type citext à cette table et plaçons-y le contenu de la colonne g_title :

```
fluxbb=# ALTER TABLE fbb_groups ADD COLUMN g_title_2 citext;
ALTER TABLE
fluxbb=# UPDATE fbb_groups SET g_title_2=g_title;
UPDATE 4
fluxbb=# SELECT g_id, g_title, g_title_2 FROM fbb_groups;
 g_id |    g_title     |   g_title_2
------+----------------+----------------
    1 | Administrators | Administrators
    2 | Moderators     | Moderators
    3 | Guest          | Guest
    4 | Members        | Members
(4 lignes)
```

Le contenu est bien identique. Faisons les mêmes tests que précédemment mais sur la nouvelle colonne :

```
fluxbb=# SELECT g_id, g_title, g_title_2 FROM fbb_groups WHERE g_title_2 like 'm%';
 g_id |  g_title   | g_title_2
------+------------+------------
    2 | Moderators | Moderators
    4 | Members    | Members
(2 lignes)

fluxbb=# SELECT g_id, g_title, g_title_2 FROM fbb_groups WHERE g_title_2 like 'M%';
 g_id |  g_title   | g_title_2
------+------------+------------
    2 | Moderators | Moderators
    4 | Members    | Members
(2 lignes)
```

La recherche, pour une colonne de type citext, ne tient pas compte de la casse.

Quelques autres nouveautés

Ces trois nouveautés ne sont pas les seules pour les utilisateurs. En voici quelques-autres rapidement.

La recherche plein texte permettait une recherche rapide de la racine d'un mot. Si on voulait profiter de la rapidité de la recherche plein texte pour une recherche plus précise, il fallait en plus ajouter une clause qui ne pouvait pas bénéficier des index plein texte. Les développeurs de la recherche plein texte ont donc travaillé sur la recherche de préfixe. Cela permet à un utilisateur de rechercher par rapport au début d'un mot dans une requête de type plein texte.

Les clauses LIMIT et OFFSET avaient une grosse limitation. Elles n'acceptaient qu'un nombre entier. Les développeurs de PostgreSQL ont ajouté le support d'une sous-requête SQL.

```
fluxbb=# SELECT subject FROM fbb_topics
ORDER BY last_post DESC
LIMIT (SELECT count(*)/100 FROM fbb_topics);
                          subject
------------------------------------------------------------
 La réplication en 8.5: Un aperçu ?
 Connection à une base DB2 via un DBLINK est ce possible
 Poste DBA PostgreSQL en CDI
(3 lignes)
```

Peter Eisentraut, l'un des développeurs de la Core Team de PostgreSQL, a travaillé à améliorer le support du standard SQL. Pour cela, il a notamment ajouté la clause TABLE. « TABLE fbb_groups » revient à faire un « SELECT * FROM fbb_groups ». Cela étant dit, il reste encore à démontrer l'intérêt de cette fonctionnalité, en dehors du respect intégral de ce standard.

Un grand nombre de fonctions a été ajouté pour améliorer la gestion des tableaux :

- array_ndims() pour connaître le nombre de dimensions d'un tableau ;
- array_length() pour connaître la taille du tableau ;
- array_agg(), fonction d'agrégat permettant de créer un tableau à partir des valeurs des lignes de l'agrégat ;
- array_unnest() pour faire l'opération inverse (on récupère une ligne par valeur du tableau) ;
- array_fill() pour remplir un tableau.

Voici un exemple de chaque :

```
fluxbb=# SELECT array_ndims(array[1, 2, 3, 4]);
 array_ndims
-------------
           1
(1 ligne)

fluxbb=# SELECT array_ndims(array[[1,2],[3,4]]);
 array_ndims
-------------
           2
(1 ligne)

fluxbb=# SELECT array_length(array[1, 2, 3, 4], 1);
 array_length
--------------
            4
(1 ligne)

fluxbb=# SELECT array_length(array[[1,2],[3,4]], 1);
 array_length
--------------
            2
(1 ligne)

fluxbb=# SELECT g_title FROM fbb_groups;
    g_title
----------------
 Administrators
 Moderators
 Guest
 Members
(4 lignes)

fluxbb=# SELECT array_agg(g_title) FROM fbb_groups;
                array_agg
------------------------------------------
 {Administrators,Moderators,Guest,Members}
(1 ligne)

fluxbb=# SELECT unnest(array[1,2,3,4]);
 unnest
--------
      1
      2
      3
      4
(4 lignes)

fluxbb=# SELECT array_fill('a'::text, array[2]);
 array_fill
------------
 {a,a}
(1 ligne)
```

```
fluxbb=# SELECT array_fill('a'::text, array[2,3]);
    array_fill
-----------------
 {{a,a,a},{a,a,a}}
(1 ligne)
```

Peter Eisentraut a aussi ajouté le début de l'infrastructure pour SQL/MED. Cet autre standard SQL doit fournir une API pour accéder à distance à des SGBD.

L'outil en ligne de commande psql a reçu son lot d'améliorations. Il est possible de se connecter à des serveurs de versions antérieures et d'utiliser les méta-commandes sans problèmes. Les méta-commandes ont aussi beaucoup évoluées. Par exemple, certaines commandes \d affichaient les objets utilisateurs et systèmes par défaut alors que d'autres n'affichaient que les objets utilisateurs. Maintenant, \d n'affiche que les objets utilisateurs. Il faut ajouter l'option S pour avoir en plus les objets systèmes. Ce n'est pas une grosse amélioration en soi, mais ce petit détail ajoute un peu de cohérence.

Pour le développeur

Le langage PL/pgsql a été amélioré par de nombreuses petites fonctionnalitées intéressantes.

Il est enfin possible d'indiquer des valeurs par défaut pour les arguments d'une fonction. Pour cela, il faut utiliser le mot clé DEFAULT ou l'opérateur = après le nom de l'argument. Par exemple:

```
fluxbb=# CREATE OR REPLACE FUNCTION get_topic(p_id integer = 10)
RETURNS text
LANGUAGE plpgsql AS
$$
DECLARE
  s text;
BEGIN
  SELECT subject INTO s FROM fbb_topics WHERE id=p_id;
  RETURN s;
END;
$$;
CREATE FUNCTION
fluxbb=# SELECT get_topic(2);
       get_topic
------------------------
 Nouveau site français !
(1 ligne)

fluxbb=# SELECT get_topic();
                 get_topic
---------------------------------------------
 Problème lié à l'archivage des fichiers WAL
(1 ligne)

fluxbb=# SELECT get_topic(10);
                 get_topic
---------------------------------------------
 Problème lié à l'archivage des fichiers WAL
(1 ligne)
```

En cas d'ambiguité entre deux fonctions, PostgreSQL renverra une erreur indiquant qu'il n'a pas pu choisir entre ces deux fonctions. Les arguments disposant de valeurs par défaut seront toujours les derniers arguments de la fonction.

Il est aussi possible de déclarer des fonctions dont le nombre de paramètres n'est pas défini lors de la création. Les paramètres en question doivent tous être du même type, et seront fournis à la fonction sous la forme d'un tableau d'éléments.

```
fluxbb=# CREATE OR REPLACE FUNCTION get_topics(variadic p_id integer[])
RETURNS SETOF text
LANGUAGE plpgsql AS
$$
DECLARE
  r record;
BEGIN
  FOR r IN SELECT subject FROM fbb_topics, unnest(p_id) AS tmp(i) WHERE id=i
  LOOP
    RETURN NEXT r.subject;
  END LOOP;
  RETURN;
END;
$$;
fluxbb=# SELECT * FROM get_topics(10, 20);
                 get_topics
---------------------------------------------
 IMPORTATION des TABLES
 Problème lié à l'archivage des fichiers WAL
(2 lignes)

fluxbb=# SELECT * FROM get_topics(10, 15, 100);
                 get_topics
---------------------------------------------
 Problème avec pg_dump sur un bytea
 psql et création de table
 Problème lié à l'archivage des fichiers WAL
(3 lignes)
```

Une nouvelle structure de contrôle fait son apparition: CASE. Cette structure, déjà bien connue des développeurs, remplacera agréablement un ensemble de tests IF, seule façon de simuler CASE auparavant.

Une fonction accepte en type de retour une table. C'est une spécification du standard SQL et ça a le gros avantage de ne pas avoir à définir de paramètres OUT :

```
fluxbb=# CREATE OR REPLACE FUNCTION get_topics(variadic p_id integer[])
RETURNS TABLE (id integer, contenu text)
LANGUAGE sql AS
$$
  SELECT id, subject::text FROM fbb_topics, unnest($1) AS tmp(i) WHERE id=i;
$$;
CREATE FUNCTION
fluxbb=# SELECT * FROM get_topics(10, 20);
 id |                 contenu
----+---------------------------------------------
 20 | IMPORTATION des TABLES
 10 | Problème lié à l'archivage des fichiers WAL
(2 lignes)
```

La commande RAISE a bénéficié d'une grande attention. Auparavant, il était simplement possible de renvoyer un message ou une exception. Les nouveautés pour cette commande ajoutent des informations importantes : champs DETAIL et HINT, ainsi que le code d'erreur SQLSTATE.

```
fluxbb=# CREATE OR REPLACE FUNCTION get_topic(p_id integer = 10)
RETURNS text
LANGUAGE plpgsql AS
$$
DECLARE
  s text;
BEGIN
  SELECT subject INTO s FROM fbb_topics WHERE id=p_id;
  IF NOT FOUND
  THEN
    RAISE NOTICE 'identifiant % inconnu', p_id
    USING HINT = 'Toujours vérifier l''identifiant avant d''exécuter cette fonction',
    ERRCODE = 22023;
  END IF;
  RETURN s;
END;
$$;
fluxbb=# SELECT get_topic(10);
                 get_topic
---------------------------------------------
 Problème lié à l'archivage des fichiers WAL
(1 ligne)

fluxbb=# SELECT get_topic(1);
NOTICE:  identifiant 1 inconnu
ASTUCE : Toujours vérifier
l'identifiant avant d'exécuter cette fonction
 get_topic
-----------

(1 ligne)
```

D'autres nouveautés, moins importantes, sont apparues. Par exemple, PostgreSQL avait ajouté le support de la clause RETURN QUERY pour la 8.3, ce qui permettait de renvoyer le résultat d'une requête comme retour de la fonction. Malheureusement, cette requête ne pouvait pas être construite comme une

chaîne de caractères. Pavel Stehule a donc ajouté le support de la clause RETURN QUERY EXECUTE pour combler ce vide.

La fonction quote_nullable() facilite l'écriture de requêtes dynamiques.

La bibliothèque libpq a aussi bénéficié d'améliorations comme le support d'événements, une nouvelle erreur pour une requête vide et un meilleur support du SSL.

Pour l'administrateur

Cette version dispose de nombreuses améliorations pour l'administrateur : meilleure gestion des bases de données, nouvelle gestion de la fragmentation des tables, ajout d'une gestion des droits sur les colonnes, nouvelles statistiques, etc.

Paramètrage des tris et jeux de caractères par base de données

L'un des gros soucis avec PostgreSQL est que le tri alphabétique est configurable qu'au moment de l'initialisation du cluster. Cela sous-entend qu'il n'est pas non plus possible de le configurer par base de données.

PostgreSQL 8.4 change cela complètement. La création d'une base de données permet de spécifier deux nouveaux paramètres : LC_CTYPE pour le jeu de caractères utilisé et LC_COLLATE pour le tri alphabétique.

L'outil en ligne de commande psql a été modifié pour afficher ces nouvelles informations:

```
guillaume@laptop:~$ createdb --locale=C --template template0 base_en
guillaume@laptop:~$ psql -l
                                 Liste des bases de données
     Nom      | Propriétaire | Encodage |     Tri     | Type caract. |     Droits d'accès
--------------+--------------+----------+-------------+--------------+-----------------------
 base_en      | guillaume    | UTF8     | C           | C            |
 fluxbb       | fluxbb       | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 frida        | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 postgres     | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 stackoverflow | guillaume   | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 template0    | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/guillaume
                                                                    : guillaume=CTc/guillaume
 template1    | guillaume    | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/guillaume
                                                                    : guillaume=CTc/guillaume
(7 lignes)
```

Nouvelle gestion de la fragmentation des tables

Auparavant, lors d'une opération VACUUM, PostgreSQL indiquait la liste des blocs où il pouvait écraser des informations maintenant inutiles dans une structure en mémoire appelée FSM (pour Free Space Map, soit la carte des espaces libres). La taille de cette structure n'était pas dynamique : elle dépendait de deux paramètres (max_fsm_pages et max_fsm_relations), souvent peu considérés car mal compris. Malheureusement, si la structure était trop petite, certaines informations étaient oubliées, ce qui finissaient par aboutir à des tables trop grosses pour cause de fragmentation.

Heikki Linnakangas a décidé de travailler là-dessus en faisant en sorte que cette structure soit dynamique. Pour cela, au lieu de la conserver en mémoire, elle est stockée sur disque, avec un fichier par table. Ce fichier a pour nom le numéro de la table (numéro correspondant à la valeur de la colonne relfilenode dans le catalogue système pg_class) suivi du suffixe « _fsm ». Comme pour les tables, chaque lecture d'un bloc de ce fichier sera placé dans le cache disque de PostgreSQL. Chaque écriture se fera de la même façon. Donc il utilise un mécanisme déjà bien rôdé. Le premier gros avantage de ce système, c'est qu'il fait disparaître les deux paramètres max_fsm_pages et max_fsm_relations. Il n'y a plus non plus de soucis de perdre une information. Vous n'avez donc plus qu'à vous assurer que vous faites suffisamment de VACUUM.

À ce sujet, cette innovation a permis l'implantation d'un autre mécanisme très intéressant : la carte de visibilité. Ce fichier, géré exactement comme le fichier des espaces libres, contient toutes les pages visibles par toutes les transactions en cours. Autrement dit, c'est la liste des pages ne contenant aucune donnée modifiée, donc ne nécessitant pas de VACUUM. Ce dernier a été amélioré pour tenir compte de cette information et ne parcourir que les pages vraiment modifiées. Néanmoins, pour éviter que la table ne soit jamais parcourue entièrement par un VACUUM, un nouveau paramètre est apparu : vacuum_freeze_max_age. Ce paramètre indique l'âge maximum d'une table avant qu'elle ne subisse un VACUUM complet.

Ces deux améliorations vont permettre d'améliorer les performances du VACUUM. D'autres améliorations sont même prévues pour les prochaines versions.

Droits sur les colonnes

PostgreSQL dispose de droits assez complets au niveau des objets qu'il est capable de gérer. L'un des objets manquants dans la liste était les colonnes. Ceci est corrigé en 8.4, comme le montre l'exemple suivant :

```
fluxbb=# CREATE TABLE t1 (id serial, c1 text, c2 text, c3 text, c4 text);
NOTICE:  CREATE TABLE créera des séquences implicites « t1_id_seq » pour la colonne serial « t1.id »
CREATE TABLE
fluxbb=# INSERT INTO t1 VALUES (1, 'a', 'A', 'à', ''), (2, 'e', 'E', 'é', 'è'), (3, 'i', 'I', 'î', 'ï');
INSERT 0 3
fluxbb=# SELECT * FROM t1;
 id | c1 | c2 | c3 | c4
----+----+----+----+----
  1 | a  | A  | à  |
  2 | e  | E  | é  | è
  3 | i  | I  | î  | ï
(3 lignes)

fluxbb=# CREATE USER glmf1 LOGIN;
CREATE rôle
fluxbb=# REVOKE SELECT ON TABLE t1 FROM glmf1;
REVOKE
fluxbb=# \c fluxbb glmf1
psql (8.4rc2)
Vous êtes maintenant connecté à la base de données « fluxbb »comme utilisateur « glmf1 ».
fluxbb=> SELECT * FROM t1;
ERREUR:  droit refusé pour la relation t1
fluxbb=> \c fluxbb guillaume
psql (8.4rc2)
Vous êtes maintenant connecté à la base de données « fluxbb »comme utilisateur « guillaume ».
fluxbb=# GRANT SELECT (id, c1) ON TABLE t1 TO glmf1;
GRANT
fluxbb=# \c fluxbb glmf1
psql (8.4rc2)
Vous êtes maintenant connecté à la base de données « fluxbb »comme utilisateur « glmf1 ».
fluxbb=> SELECT * FROM t1;
ERREUR:  droit refusé pour la relation t1
fluxbb=> SELECT id, c1 FROM t1;
 id | c1
----+----
  1 | a
  2 | e
  3 | i
(3 lignes)
```

Il est donc possible d'autoriser, ou non, la lecture ou l'écriture d'une colonne.

Restauration parallélisée

Un problème souvent constaté est la lenteur de la restauration. C'est principalement dû au fait que pg_restore est mono-processus. L'idée a donc été de paralléliser certaines opérations. pg_restore propose maintenant un mode activable par l'option -j en ligne de commande. Le principe est d'indiquer un nombre correspondant au nombre de processeurs sur le serveur. Par exemple « -j 4 » pour deux bi-core.

À noter que seul pg_restore propose cela, ce qui sous-entend que seules les sauvegardes en mode tar ou compressées peuvent bénéficier de cette amélioration.

Amélioration du TRUNCATE

L'opération TRUNCATE est une opération très intéressante. Elle est extrêmement rapide pour supprimer le contenu complet d'une table. Il lui manquait deux fonctionnalités : un droit spécifique et un trigger. C'est chose faite en 8.4.

Il est possible d'autoriser seulement certains utilisateurs à utiliser cette instruction sur certaines tables grâce au nouveau droit TRUNCATE.

Il est aussi possible maintenant de déclencher une procédure stockée suite à l'exécution de cette instruction. Évidemment, vu la nature intrinsèque de cette instruction, le trigger ne pourra être qu'un trigger en mode instruction (impossible donc de déclencher une procédure pour chaque ligne supprimée). Par contre, le reste des fonctionnalités des triggers est disponible.

Statistiques

Niveau statistiques, les nouveautés ne manquent pas.

Commençons par la plus visible. Après les statistiques sur les tables, sur les index et sur les séquences, voici une statistique sur les fonctions. La vue s'appelle pg_stat_user_functions. Pour activer la récupération des statistiques sur les fonctions, il faut configurer le paramètre track_functions à on. Voici ce que donne la vue une fois la fonctionnalité activée et quelques fonctions exécutées:

```
fluxbb=> SELECT * FROM pg_stat_user_functions ;
 funcid | schemaname |  funcname  | calls | total_time | self_time
--------+------------+------------+-------+------------+-----------
```

```
  25547 | public    | get_topics |    2 |        26 |       26
  25540 | public    | get_topic  |   10 |        14 |       14
(2 lignes)
```

Les trois premières colonnes permettent d'identifier précisément la fonction concernée par les statistiques : OID de la fonction, nom du schéma de stockage et nom de la fonction. Les trois dernières colonnes sont les statistiques proprement dites : nombre d'appels, durée totale d'exécution (incluant le temps d'exécution des fonctions imbriquées), durée totale d'exécution de cette seule fonction. Ces trois colonnes sont très intéressantes pour connaître les fonctions très fréquemment exécutées et celles dont l'exécution dure longtemps, les deux étant à optimiser autant que possible.

La deuxième nouveauté concerne le fichier des statistiques. Avec les versions précédentes, PostgreSQL enregistrait les statistiques sur disque toutes les 500 ms. Cela pouvait avoir un fort impact au niveau des entrées/sorties. Deux modifications se sont chargées de ce problème. La première fait que le fichier n'est écrit que lorsque cela est nécessaire. La deuxième fait que le fichier est déplaçable. Plutôt que de se trouver dans $PGDATA/global, il est possible d'indiquer son emplacement avec le nouveau paramètre stats_temp_directory. Cela a le gros avantage de pouvoir indiquer un disque SSD ou un montage en mémoire, histoire d'accélerer les écritures. Il est à savoir que de toute façon, quand le serveur s'arrête, ce fichier est copié à son ancien emplacement.

Deux nouveaux modules contrib simplifient la vie dans certains cas. Leur installation est particulière. Le moyen de les activer ne revient pas à seulement exécuter un script SQL qui ajouterait des objets dans la base. Pour les activer, il faut aussi modifier le fichier de configuration de PostgreSQL.

Prenons pg_stat_statement. Ce module récupère la liste des requêtes les plus fréquemment exécutées. Pour l'activer, il faut configurer shared_preload_libraries ainsi:

```
shared_preload_libraries = 'pg_stat_statements'
```

Ensuite, certaines variables personnalisées sont disponibles. Une configuration de base donne ceci:

```
custom_variable_classes = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

pg_stat_statements.max indique le nombre maximum de requêtes à conserver. Plus vous augmentez ce paramètre, plus la quantité de mémoire partagée augmente. pg_stat_statements.track indique le type des requêtes que vous voulez surveiller : all pour toutes, top pour les requêtes émises par les clients et none pour ne pas les tracer. Ce paramétrage étant fait, il faut redémarrer PostgreSQL pour que le module puisse réclamer de la mémoire partagée. À partir de maintenant, PostgreSQL, par le biais du module, va conserver la trace de toutes les requêtes exécutées (sauf respect des conditions spécifiées par les variables).

Pour consulter les traces, il faut installer quelques objets dans la base de données :

```
guillaume@laptop:~/freeprojects/cvs.postgresql.org/head/contrib/pg_stat_statements$ psql -f /opt/postgresql-head/share/contrib/pg_stat_statements.sql fluxbb
SET
CREATE FUNCTION
CREATE FUNCTION
CREATE VIEW
GRANT
REVOKE
```

Ceci fait, il suffit de se connecter à la base de données et de regarder le contenu de la table pg_stat_statements:

```
guillaume@laptop:~/freeprojects/cvs.postgresql.org/head/contrib/pg_stat_statements$ psql fluxbb
psql (8.4rc2)
Saisissez « help » pour l'aide.

fluxbb=# SELECT true;
 bool
------
 t
(1 ligne)

fluxbb=# SELECT * FROM pg_stat_statements;
 userid | dbid  |                query                | calls | total_time | rows
--------+-------+-------------------------------------+-------+------------+------
     10 | 16386 | select pg_stat_statements_reset(); |     1 |   0.000276 |    1
     10 | 16386 | select * from pg_stat_statements;  |     1 |   9.3e-05 |    2
     10 | 16386 | select true;                       |     2 |   3.5e-05 |    2
(3 lignes)

fluxbb=#
```

Le module auto_explain a pour but d'exécuter un EXPLAIN de toutes les requêtes exécutées par les clients, et de stocker le résultat dans les traces. C'est particulièrement intéressant quand on veut comprendre pourquoi la requête exécutée par un client est si lente.

Divers

Évidemment, cela ne s'arrête pas là. La liste complète serait trop longue à fournir (et certainement ennuyante à lire), mais voici quelques autres ajouts intéressants. Les paramètres de l'autovacuum sont spécifiables par table. Ça existait avant, mais ça utilisait une table à part. L'information est maintenant donnée via les commandes CREATE/ALTER TABLE. Il est en plus possible de fournir ces informations pour les tables TOAST.

Le support de SSL a été grandement amélioré avec une gestion poussée des certificats, aussi bien du côté serveur que du côté client. L'authentification a bien évolué avec la suppression de la méthode crypt (pas suffisamment sécurisée), l'ajout de la méthode cert (certificat SSL), l'analyse du fichier pg_hba.conf pour afficher toutes les erreurs, etc.

L'ajout d'une ou plusieurs colonnes à une vue ne nécessite plus sa destruction puis re-création. Il existe maintenant un ordre CREATE OR REPLACE VIEW.

```
fluxbb=# CREATE VIEW v1 AS SELECT id, username FROM fbb_users;
CREATE VIEW
fluxbb=# SELECT * FROM v1 LIMIT 2;
 id  | username
-----+----------
 315 | u315
 316 | u316
(2 lignes)

fluxbb=# CREATE OR REPLACE VIEW v1 AS SELECT id, username, num_posts FROM fbb_users;
CREATE VIEW
fluxbb=# SELECT * FROM v1 LIMIT 2;
 id  | username | num_posts
-----+----------+-----------
 315 | u315     |         0
 316 | u316     |         0
(2 lignes)
```

Le EXPLAIN VERBOSE affiche les colonnes, ce qui permet de démontrer plus fortement le côté diabolique de l'étoile pour les performances.

```
fluxbb=# EXPLAIN VERBOSE SELECT * FROM fbb_users WHERE num_posts > 5;

                QUERY PLAN

---------------------------------------------------------------------------------------------------------------------------------------------------------- Seq Scan on fbb_users  (cost=0.00..23.93 rows=54 width=585)
   Output: id, group_id, username, password, email, title, realname, url, jabber, icq, msn, aim, yahoo, location, use_avatar, signature, disp_topics, disp_posts, email_setting, save_pass, notify_with_post, show_smilies, show_img, show_img_sig, show_av
   Filter: (num_posts > 5)
(3 lignes)
```

Deux nouvelles fonctions vont intéresser principalement les administrateurs : pg_conf_load_time() pour savoir quand la configuration a été relue la dernière fois, et pg_terminate_backend() pour fermer la connexion à un client (et donc arrêter le processus serveur lié à ce client).

La taille de la requête indiquée dans pg_stat_activity est enfin configurable grâce au nouveau paramètre track_activity_query_size, ce qui permet de dépasser le blocage à 1024 caractères.

Incompatibilité

Comme pour toute version majeure, il existe un certain nombre d'incompatibilité.

En tout premier lieu, il faut savoir que les types dates et heures sont maintenant codés avec des entiers sur 64 bits, et non plus des nombres à virgule flottante. C'était déjà possible auparavant, mais cela demandait de modifier une option de configuration avant la compilation. À noter que cette modification va poser un sérieux problème pour utiliser pg_migrator, un nouvel outil ayant pour but de vous permettre de passer d'une version majeure à une autre sans avoir à faire le cycle pg_dump / initdb / pg_restore. Pour l'utiliser, il faut que les options de compilation de la 8.4 soient identiques à celles de la 8.3. Donc il faudra désactiver cette option (–disable-integer-datetimes en argument de l'outil configure) pour permettre son utilisation. Et il y a fort peu de chances que les packageurs ajoutent cette option ou créent plusieurs versions de leur package.

Il est aussi à noter que les instructions TRUNCATE et LOCK s'appliquent à la table précisée ainsi qu'aux tables filles. Enfin, certains paramètres changent de valeurs par défaut : log_min_messages passe de notice à warning, max_prepared_transactions de 5 à 0, default_statistics_target de 10 à 100, constraint_exclusion de off à partition.

Petit bilan

Nous nous trouvons de nouveau avec une version très sérieuse, composée de nombreuses nouvelles fonctionnalités et augmentant d'autant l'intérêt de ce SGBD.

Par rapport à la version précédente, le processus de développement a été globalement mieux géré. Néanmoins, il faut avouer que tout n'a pas été rose,

avec notamment les deux patchs sensés fournir une réplication plus poussée en interne. Ces deux patchs n'ont pas été intégrés pour principalement deux raisons : parce qu'ils sont arrivés très tard dans le processus de développement et parce qu'il y a un manque de personnes suffisamment compétentes sur le code source de PostgreSQL pour pouvoir donner un avis fiable. Ce qui n'a pas arrangé les choses, c'est que ces personnes ont commencé par vider la queue des patchs en travaillant sur les patchs les plus simples, alors que d'autres personnes moins aguerries auraient pu s'en occuper. Ces erreurs ne sont pas graves en soi à condition qu'on cherche à y apporter des remèdes. Il est maintenant convenu que les patchs les plus lourds ont tout intérêt à être proposé tôt dans le processus de développement. Les développeurs les plus compétents ont accepté de se consacrer aux patchs les plus complexes, et laisser le traitement de la queue pleine de patchs simples à des personnes moins expérimentées.

Autre amélioration, concernant les tests, à la fin de chaque commit fest verra la sortie d'une version alpha de PostgreSQL. Évidemment, il est possible que certains commit fest en réchappent si la version est jugée trop instable. Néanmoins, l'arrivée des versions alpha permettra plus de tests, et donc des versions débuggées plus rapidement. Tout ceci devrait concourir à une version réellement exceptionnelle l'année prochaine.

Et le futur?

En terme de fonctionnalités, à quoi faut-il s'attendre pour la prochaine version ?

Sans trop prendre de risque, on peut penser que la réplication sera au cœur de cette version grâce aux patchs Hot Standby de Simon Riggs (esclave en lecture seule) et SyncRep de Fujii Masao (réplication à la transaction prête).

Les autres sujets chauds sont la mise à jour de version majeure en ligne (« upgrade in place »), une meilleure gestion des modules (pensez PostGIS notamment), une exécution parallélisée des requêtes. Mais beaucoup d'autres patchs sont en cours de développement. La prochaine commit fest, qui aura lieu le 15 juillet, comprend déjà une liste de 50 patchs.

Au niveau développement, le passage à git est très discuté. Il faut avouer que l'utilisation du très vieux CVS commence à faire grincer des dents chez certains développeurs.

Conclusion

Cette version apporte de nombreuses fonctionnalités très intéressantes pour les utilisateurs. Les administrateurs ne sont pas en reste et devraient beaucoup apprécier les améliorations apportées par la nouvelle gestion de la fragmentation des tables.