(Kopfzeile)

Out now! PostgreSQL 8.4

Die neuen Features der Open Source Datenbank PostgreSQL 8.4

Nach einem Jahr Entwicklungszeit war es am 01.07.2009 soweit. Die „most advanced“ Open Source Datenbank PostgreSQL wurde in der Version 8.4 released. Was gibt es Neues? Was wurde verbessert? Der Artikel bietet Ihnen einen ersten Überblick und soll Ihnen Appetit auf mehr PostgreSQL verschaffen.

Andreas Wenk

Eigentlich war der Release der PostgreSQL 8.4 schon für März diesen Jahres geplant. Allerdings war das jüngste Baby der PostgreSQL Community noch nicht so weit um das Licht der Welt – oder besser die querys der Benutzer – zu erblicken. Deuten wir das als Beweis dafür, dass die Kernaussage vieler Open Source Projekte „it's done when it's done“ zutrifft, und zum anderen, dass Sie als Benutzer davon ausgehen können, ein extrem stabiles System zu erhalten. Das Entwickler Core Team und die Contributor geben sich erst zufrieden, wenn alle Tests erfolgreich durchlaufen und alle geplanten Features, soweit möglich, integriert sind.

Die neuste Version der PostgreSQL bietet wie bei jedem Major Release in fast allen Bereichen Verbesserungen, Erweiterungen und Neuerungen. Die Release Notes, zu finden unter [1], sind eine Lektüre allemal Wert. Natürlich können in diesem Artikel nicht alle Punkte behandelt werden. Aber lassen Sie uns einen Blick auf folgende Bereiche werfen.

Common Table Expressions (CTE's) und Recursive Queries

Unter *Common Table Expressions* versteht man eine temporäre Ergebnismenge, welcher eine Bezeichnung gegeben wird und von einer *SELECT*, *INSERT*, *UPDATE* oder *DELETE* Abfrage abgeleitet ist. Erstellt wird ein CTE prinzipiell durch die Nutzung des SQL Befehls *WITH* gefolgt von einem oder mehreren Subquerys (Unterabfragen). Das Ergebnis dieser Subquerys bildet dann die Grundlage für die eigentliche Abfrage. Das dient eindeutig der Übersicht, gerade wenn man sonst mehrere verschachtelte Subquerys benötigen würde. Außerdem lässt sich mehrere male auf ein Subquery zugreifen, ohne dieses jedes Mal ausführen zu müssen. Sehen wir uns gleich ein Beispiel an, aber zuvor erstellen wir eine Tabelle für eine beispielhafte Raumverwaltung für unsere Anschauungszwecke (diese Tabelle ist auch die Grundlage für die anderen Beispiele in diesem Artikel):

```
CREATE TABLE raumverwaltung (
    id int NOT NULL PRIMARY KEY,
    bezeichnung char varying(100),
    typ char varying (3),
    bewertung int NOT NULL,
    parent int NOT NULL,
    CONSTRAINT check_typ CHECK (typ = \
       ANY(ARRAY['sta'::bpchar,'geb'::bpchar,'rau'::bpchar])),
    CONSTRAINT check_bewertung (bewertung > 1 AND bewertung < 10)
)
```

Damit Sie mit den folgenden Abfragen etwas spielen können, sollten Sie ich die Mühe machen und einige Daten einfügen, wobei diese natürlich nach Möglichkeit unterschiedlich sein sollten. Die Tabelle erklärt sich größtenteils von selbst. In das Feld *parent* wird eine Beziehung zu einer anderen

Zeile in dieser Tabelle unter Verwendung der *id* hergestellt. Alternativ können Sie auch einen vorbereiteten dump von [2] herunterladen. Jetzt aber zurück zu unserem Beispiel:

```
WITH raumbewertung AS (
        SELECT floor(avg(bewertung)) AS bewertung
        FROM raumverwaltung
), raum AS (
        SELECT bezeichnung, bewertung
        FROM raumverwaltung
        WHERE bewertung > (SELECT bewertung FROM raumbewertung)
)
SELECT a.bezeichnung, a.bewertung, b.bewertung as durchschnitt
FROM raum a, raumbewertung b
WHERE a.bezeichnung IN (
        SELECT bezeichnung FROM raum
)
GROUP BY a.bezeichnung, a.bewertung, b.bewertung;
```

Als Ergebnis erhalten wir:

```
 bezeichnung | bewertung | durchschnitt
-------------+-----------+--------------
 Muenchen    |         9 |            6
 Haus 1 B    |         9 |            6
 Haus 1 M    |         9 |            6
 Raum 2 M    |         7 |            6
 Haus 1 HH   |         8 |            6
(5 rows)
```

Was passiert hier? Das Ziel der Abfrage ist ein Ergebnis zu erhalten, welches eine Menge an Raumobjekten zurück gibt, die eine bessere Bewertung als der Durchschnitt aller Raumobjekte haben. Die Abfrage wird mit *WITH* eingeleitet wobei das erste Subquery in *bewertung* die nach unten gerundeten durchschnittliche Bewertung aller Raumobjekte vorhält. Im nächsten Subquery fragen wir dann die Bezeichnung des Raumobjekts ab, dessen Bewertung größer als die zuvor ermittelte durchschnittliche Bewertung ist. Dabei greifen wir auf das Ergebnis zu, indem wir wieder ein Subquery absetzen (unter Verwendung des SQL Befehls *IN*) und als 'Tabelle' den Namen des ersten Subquerys angeben - *raumbewertung*. Aha – ganz richtig erkannt. Sie können sich die Subquerys als temporäre Tabellen vorstellen, von welchen Sie ein Ergebnis abfragen. Zu guter Letzt kommt dann die eigentliche Abfrage, in welcher dann einfach alle Bezeichnungen aus dem Subquery *raum* abgefragt werden und mit einem *GROUP BY* Befehl dafür gesorgt wird, dass keine doppelten Einträge zurückgegeben werden.

Natürlich stellt das obige Beispiel einen einfachen Fall dar und ja, man kann das auch ohne diese Methode bewerkstelligen. Allerdings müssen Sie zugeben, dass das schon sehr der Übersicht dient.

Damit aber noch nicht genug mit CTE's. Die PostgreSQL bietet auch die Variante *WITH RECURSIVE*. Dadurch bietet sich die Möglichkeit innerhalb einer Abfrage nicht nur auf das eigene Ergebnis zuzugreifen, sondern auch darüber zu iterieren (*RECURSIVE* ist hier etwas irreführen entspricht aber dem SQL Standard). Dabei werden zwei Statements mit *UNION* bzw. *UNION ALL* zusammengeführt, wobei das zweite die Rekursion darstellt. Diese Art von Rekursion kann sehr gut

für die Darstellung einer Baumstruktur eingesetzt werden. Spannend! Wie geht das? Sehen wir es uns an:

```
WITH RECURSIVE raum(parent, bezeichnung, baum) AS (
        SELECT id, bezeichnung, ARRAY[id]
        FROM raumverwaltung
        WHERE id IN (1,2,3)
        UNION ALL
        SELECT a.id, a.bezeichnung, b.baum || ARRAY[a.id]
        FROM raumverwaltung a, raum b
        WHERE a.id != ANY (b.baum)
        AND a.parent = b.parent
)
SELECT *
FROM raum
ORDER BY baum;
```

Als Ergebnis erhalten wir:

```
parent | bezeichnung |   baum
--------+-------------+-----------
      1 | Hamburg     | {1}
      4 | Haus 1 HH   | {1,4}
      5 | Raum 1 HH   | {1,4,5}
      6 | Haus 2 HH   | {1,6}
      2 | Muenchen    | {2}
      7 | Haus 1 M    | {2,7}
      8 | Raum 1 M    | {2,7,8}
      9 | Raum 2 M    | {2,7,9}
      3 | Berlin      | {3}
     10 | Haus 1 B    | {3,10}
     11 | Raum 1 B    | {3,10,11}
(11 rows)
```

Zuerst bestimmen wir, welche Felder unser *WITH* Statement zurückgeben soll (*parent*, *bezeichnung*, *baum*). Dann werden alle Zeilen zurück gegeben, die entweder 1, 2 oder 3 als id beinhalten. Das ist nur ein workaround um das Ergebnis anschaulicher zu machen. Das Ergebnis ist also die Grundlage für den *UNION ALL* Befehl.Im zweiten *SELECT* Statement müssen ja per Definition von *UNION* die gleichen Spalten abgefragt werden. wobei wir das Array aus dem ersten *SELECT* Statement mit einem Array aus der *id* der Tabelle *raumverwaltung* konkatenieren. Die Einschränkungen geben an, dass wir keine Zeilen wollen, bei denen die *id* der Tabelle *raumverwaltung* im Array der Spalte *baum* ist und aber alle Zeilen, bei denen die *parent id's* übereinstimmen. Das Ergebnis ist eine ziemlich brauchbare Baumstruktur (das obige *WITH RECURSIVE* Beispiel ist 'inspired by' einem Beispiel von Andreas 'ads' Scheerbaum).

Natürlich stellen die Beispiele nur die prinzipielle Technik dar. Es gibt mit Sicherheit viele Fälle, in welchen diese Techniken sehr gut angewendet werden können. Weitere Informationen zu CTE's und Recursive Queries erhalten Sie in der online Dokumentation der PostgreSQL unter [3].

WINDOW Functions

Ins deutsche übersetzt heißen *WINDOW Functions* Fenster Funktionen. Zugegebenermaßen klingt das ziemlich doof, sagt aber ziemlich genau aus um was es hier geht. *WINDOW Functions* bieten die Möglichkeit in einem Ergebnis zusätzlich ein Aggregat von zusammengehörenden Zeilen zu erstellen. Dabei wird allerdings jede einzelne Zeile bewahrt und nicht gleichartige Zeilen gruppiert. Das kann mit normalen Aggregat Funktionen erreicht werden. Das bedeutet also, dass angegeben werden muss, welcher Blick durch's Fenster auf welche zusammengehörigen Zeilen gegeben werden soll. Ein denkbar einfaches Beispiel ist folgendes:

```
SELECT *,
        avg(bewertung) OVER (PARTITION BY typ) durch_bewert,
        min(bewertung) OVER (PARTITION BY typ) min_bewert,
        max(bewertung) OVER (PARTITION BY typ) max_bewert
FROM raumverwaltung
ORDER BY typ DESC, bewertung DESC;
```

Das Ergebnis:

```
id | bezeichnung | typ | bewertung |parent|durch_bewert       |min_bewert|max_bewert
---+-------------+-----+-----------+------+-------------------+----------+----------
2  | Muenchen    | sta |         9 |  2   |6.0000000000000000 |      3   |       9
3  | Berlin      | sta |         6 |  3   |6.0000000000000000 |      3   |       9
1  | Hamburg     | sta |         3 |  1   |6.0000000000000000 |      3   |       9
9  | Raum 2 M    | rau |         7 |  7   |4.2500000000000000 |      2   |       7
5  | Raum 1 HH   | rau |         4 |  4   |4.2500000000000000 |      2   |       7
11 | Raum 1 B    | rau |         4 |  10  |4.2500000000000000 |      2   |       7
8  | Raum 1 M    | rau |         2 |  7   |4.2500000000000000 |      2   |       7
10 | Haus 1 B    | geb |         9 |  3   |7.7500000000000000 |      5   |       9
7  | Haus 1 M    | geb |         9 |  2   |7.7500000000000000 |      5   |       9
4  | Haus 1 HH   | geb |         8 |  1   |7.7500000000000000 |      5   |       9
6  | Haus 2 HH   | geb |         5 |  1   |7.7500000000000000 |      5   |       9
    (11 rows)
```

Anhand der Ausgabe wird ziemlich klar, was gemeint ist. Der Wert *durch_wert* wird auf Grundlage aller Zeilen mit *typ = sta*, *typ = rau* und *typ = geb* berechnet. Genauso die Werte für *min_bewert* und *max_bewert*. Außerdem ist gut erkennbar, dass beliebig viele *WINDOW Functions* in ein einzelnes Statement eingebaut werden können.

Ein anderer Ansatz ist das Verwenden eines Subquerys:

```
SELECT * FROM
        (SELECT *, rank() OVER (PARTITION BY typ ORDER BY bewertung DESC)
         FROM raumverwaltung) as rank
ORDER BY rank;
```

Das Ergebnis ist folgendes:

```
 id | bezeichnung | typ | bewertung | parent | rank
----+-------------+-----+-----------+--------+------
  9 | Raum 2 M    | rau |         7 |      7 |    1
 10 | Haus 1 B    | geb |         9 |      3 |    1
  2 | Muenchen    | sta |         9 |      2 |    1
  7 | Haus 1 M    | geb |         9 |      2 |    1
 11 | Raum 1 B    | rau |         4 |     10 |    2
```

```
  5 | Raum 1 HH   | rau |         4 |      4 |    2
  3 | Berlin      | sta |         6 |      3 |    2
  4 | Haus 1 HH   | geb |         8 |      1 |    3
  1 | Hamburg     | sta |         3 |      1 |    3
  8 | Raum 1 M    | rau |         2 |      7 |    4
  6 | Haus 2 HH   | geb |         5 |      1 |    4
(11 rows)
```

In diesem Beispiel wurde noch die Funktion *rank()* eingebunden und innerhalb eines Subquerys ausgeführt. Als Ergebnis erhalten wir ein nach *rank* sortiertes Ergebnis.

Schließlich können wir die beiden vorgestellten Techniken auch kombinieren. Dazu noch ein Beispiel:

```
SELECT * FROM (
        WITH sub_raum AS (
                SELECT bezeichnung, bewertung, typ, avg(bewertung) OVER \
                  (PARTITION BY typ) AS avg_bewertung
                FROM raumverwaltung
                GROUP BY bezeichnung, bewertung, typ
        )
        SELECT bezeichnung, bewertung, typ, avg_bewertung, rank() OVER \
           (PARTITION BY typ ORDER BY bewertung DESC) AS rank
        FROM sub_raum
) AS erg WHERE avg_bewertung > 5;
```

Als Ergebnis erhalten wir:

```
 bezeichnung | bewertung | typ |   avg_bewertung    | rank
-------------+-----------+-----+--------------------+------
 Haus 1 M    |         9 | geb | 7.7500000000000000 |    1
 Haus 1 B    |         9 | geb | 7.7500000000000000 |    1
 Haus 1 HH   |         8 | geb | 7.7500000000000000 |    3
 Haus 2 HH   |         5 | geb | 7.7500000000000000 |    4
 Muenchen    |         9 | sta | 6.0000000000000000 |    1
 Berlin      |         6 | sta | 6.0000000000000000 |    2
 Hamburg     |         3 | sta | 6.0000000000000000 |    3
(7 rows)
```

Gerade für statistische Aufgaben bieten die *WINDOW Functions* viele Möglichkeiten und sind nun in der PostgreSQL 8.4 verfügbar. Benutzer der Oracle, DB oder des SQL Server kannten dies bereits schon.

Spalten basierte Privilegien

Die PostgreSQL basiert schon seit langem auf einem Rollen Rechtesystem. Dabei erhält jede Rolle für jedes einzelne Datenbankobjekt wie Tabellen, Funktionen oder Sequenzen Privilegien. Schön ist auch, dass jede Rolle Mitglied jeder anderen Rolle sein kann und die Privilegien der parent Rolle erbt (so lange nicht anders angegeben). Seit der Version 8.4 können nun auch Privilegien für einzelne Spalten einer Tabelle Privilegien vergeben werden. Dabei haben allerdings die Rechte auf die gesamte Tabelle Vorrang. Als Beispiel haben wir eine Rolle *angucker* erstellt, welche nun nur

eineingeschrängte *SELECT* Privilegien auf die Spalten *bezeichnung* und *typ* für die Tabelle *raumverwaltung* erhält:

```
php_mag_6_09=# CREATE ROLE angucker LOGIN PASSWORD 'hallo123';
php_mag_6_09=# GRANT SELECT (bezeichnung, typ) ON raumverwaltung TO angucker;
php_mag_6_09=' \c php_mag_6_09 angucker
Password for user angucker:
psql (8.4.0)
You are now connected to database "php_mag_6_09" as user "angucker".
php_mag_6_09=> SELECT * FROM raumverwaltung;
ERROR:  permission denied for relation raumverwaltung
php_mag_6_09=> SELECT bezeichnung FROM raumverwaltung;
 bezeichnung
-------------
 Hamburg
 Muenchen
 Berlin
 Haus 1 HH
 Raum 1 HH
 Haus 2 HH
 Haus 1 M
 Raum 1 M
 Raum 2 M
 Haus 1 B
 Raum 1 B
(11 rows)
```

Im Zusammenhang mit Rechten sei übrigens noch erwähnt, dass es nun ein zu vergebendes Privileg *TRUNCATE* gibt.

Einige Weitere Neuerungen im Bereich SQL

Eine nützliche Erweiterung gibt es für den Befehl *TRUNCATE*. Zum einen gibt es ab der Version 8.4 ein Privileg *TRUNCATE* welches einer Rolle zugewiesen werden kann. Zum anderen kann mit dem Befehl *TRUNCATE tabelle_1, tabelle_2, tabelle_n RESTART IDENTITY* bewirkt werden, dass die Sequenze/n der Tabelle/n zurückgesetzt werden. *CONTINUE IDENTITY* ändert im Gegensatz dazu nichts an dem Stand der Sequenze/n, wobei das das Standard Verhalten ist. Ausserdem gibt es den Parameter *ONLY* der bewirkt, dass sich die Änderungen nicht auf Kindtabellen auswirken. Beachten Sie unbedingt die Hinweise in der online Dokumentation bzgl. *TRUNCATE* und MVCC [4] . Der Parameter *ONLY* ist nun übrigens ebenfalls für den Befehl *LOCK* verfügbar.

Ein sehr nützlicher SQL Befehl ist *CREATE TABLE tabelle AS*. Jetzt kann *EXPLAIN* auch für diesen Befehl genutzt werden. Ein kleines Beispiel:

```
php_mag_6_09=# EXPLAIN CREATE TABLE raumverwaltung_2 AS
php_mag_6_09=# SELECT * FROM raumverwaltung WHERE id > 5 AND typ = 'rau';
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on raumverwaltung  (cost=0.00..14.35 rows=1 width=246)
```

```
   Filter: ((id > 5) AND ((typ)::text = 'rau'::text))
(2 rows)
```

Naja, nicht wirklich spektakulär aber beispielhaft. Außerdem wurde noch der Parameter *WITH [NO] DATA* hinzugefügt. Damit kann angegeben werden, ob nur die Struktur des Ergebnis eines zu Grunde liegenden Statements, oder aber auch die Daten für die neue Tabelle genutzt werden. Das Standard Verhalten ist auch die Daten zu kopieren.

Das Thema *query planning* und die Erstellung entsprechender Pläne ist in der PostgreSQL eines der wichtigsten Themen für *query optimization*. In jedem Major Release sind deshalb auch immer Verbesserungen am Planer integriert um letztlich eine bessere Performance durch bessere Pläne zu erreichen. Für die Analyse und Ausgabe der Pläne wird *EXPLAIN* genutzt. In der aktuellen Version hat sich das Verhalten von *EXPLAIN VERBOSE statement* insofern geändert, als das für jede *plan node* die resultierenden Spalten ausgegeben werden. Zum Beispiel:

```
php_mag_6_09=# EXPLAIN VERBOSE SELECT * FROM raumverwaltung;
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on raumverwaltung  (cost=0.00..12.90 rows=290 width=246)
   Output: id, bezeichnung, typ, bewertung, parent
(2 rows)
```

Es gibt noch weitere Neuerungen für die Befehle ALTER, LIMIT, CREATE, SELECT und andere. Sehen Sie dafür bitte das Release Notes unter [1] durch.

Neuerungen bei PL/pgSQL

PL/pgSQL ist die prozedurale Sprache der PostgreSQL. Auch hier gibt es einige Neuerungen, die hier kurz angerissen werden sollen.

Zu erst soll erwähnt werden, dass nun auch CASE Strukturen möglich sind und somit aufwändige IF THEN ELSE Konstrukte vereinfacht werden können. Ein Beispiel könnte so aussehen:

```
CASE
   WHEN typ = 'sta' THEN
          _typ := 1;
   WHEN typ = 'geb' THEN
          _typ := 2;
   WHEN typ = 'rau' THEN
          _typ := 3;
   ELSE
          _typ := 0;
END CASE;
```

Diese Schreibweise ist wesentlich kürzer, einfacher und für Liebhaber der *CASE* Kontrollstruktur in PHP sehr einfach zu adaptieren. Etwas merkwürdig mag die Verwendung von ELSE am Ende sein. Dies ist gleichbedeutend mit *DEFAULT* in PHP und sollte immer angegeben werden (zumindest des guten Stils wegen).
Eine weitere Neuerung ist die Möglichkeit dem Befehl EXECUTE dynamisch Variablen zu übergeben. Dazu folgendes Beispiel:

```
CREATE OR REPLACE FUNCTION factor(int) RETURNS RECORD AS
$$
```

```
DECLARE
        _factor numeric;
        _typ char(3);
        _id ALIAS FOR $1;
        _ret record;
BEGIN
        SELECT typ INTO _typ FROM raumverwaltung
        WHERE id = _id;

        CASE
                WHEN _typ = 'sta' THEN
                        _factor := 3;
                WHEN _typ = 'geb' THEN
                        _factor := 2;
                WHEN _typ = 'rau' THEN
                        _factor := 1;
                ELSE
                        _factor := 0;
        END CASE;

        EXECUTE 'SELECT bezeichnung, bewertung, (bewertung * $1) as factor
                 FROM raumverwaltung
                 WHERE id = $2'
        INTO _ret
        USING _factor, _id;

        RETURN _ret;
END
$$ LANGUAGE plpgsql;
```

Das Ergebnis könnte so aussehen:

```
php_mag_6_09=# SELECT factor(4);
       factor
--------------------
 ("Haus 1 HH",8,16)
(1 row)
```

EXECUTE wird innerhalb von PL/pgSQL genutzt um dynamische Statements auszuführen. Dass nun auch Variablen dynamische übergeben werden können, erweitert diese Funktionalität immens und bringt viele Möglichkeiten mit sich. Das Gleiche kann im Übrigen auch bei dem Befehl *RETURN QUERY EXECUTE* getan werden.

Der letzte Abschnitt stellt nur einen kleinen Einblick in die Neuerungen der PostgreSQL 8.4 für PL/pgSQL dar. Natürlich ist das noch lange nicht das Ende der Fahnenstange. Auch hier sei wieder auf die Release Notes unter [1] verwiesen. Kommen wir jetzt zu Neuerungen bei *psql*.

psql – die PostgreSQL shell

Auch das CLI der PostgreSQL hat sein Fett wegbekommen. psql ist ein sehr komfortables Tool um extrem schnell mit der PostgreSQL zu kommunizieren. Tendenziell hat man mit einem Command Line Interface mehr zu tun, wenn man von UNIX artigen Betriebssystemen wie Linux oder MAC OS X kommt. Ist dem so, erwartet man natürlich solche Nettigkeiten wie *tab completion* oder eine *history*. Beides hat psql.

Um die Arbeit effizient und einfach zu gestalten, verfügt psql über einen shortcut Befehlssatz. Die Befehle beginnen immer mit einem \ Zeichen. Neu ist der sehr reduzierte Startbildschirm. Eine erste Befehlsübersicht erhält man nun mit Eingabe von *help*. Die Übersicht erhält man mit \?

```
php_mag_6_09=# \?
General
  \copyright             show PostgreSQL usage and distribution terms
  \g [FILE] or ;         execute query (and send results to file or |pipe)
  \h [NAME]              help on syntax of SQL commands, * for all commands
  \q                     quit psql

Query Buffer
  \e [FILE]              edit the query buffer (or file) with external editor
  \ef [FUNCNAME]         edit function definition with external editor
[…]
Informational
  (options: S = show system objects, + = additional detail)
  \d[S+]                 list tables, views, and sequences
  \d[S+]  NAME           describe table, view, sequence, or index
  \da[+]  [PATTERN]      list aggregates
  \db[+]  [PATTERN]      list tablespaces
[...]
```

Gerade der Bereich *Informational* und die Ausgaben wurden überarbeitet. Zum Beispiel ist die Ausgabe aller vorhandenen Datenbanken im Cluster mit \l nun wesentlich größer und lässt sich durch Angabe der Option + noch um weitere Informationen erweitern. Außerdem sind einige Funktionen wie \des[+] (foreign servers), \deu[+] (user mappings) oder \dew[+] (foreign-data wrappers) hinzu gekommen. Großartig ist auch der neue shortcut Befehl \ef um Funktionen zu bearbeiten.

Administration

Die pg_hba.conf ist eine der zentralen Konfigurationsdateien. In diesem Zusammenhang gibt es einige wichtige Änderungen. Die Authentifizierungsmethode crypt() wurde gestrichen und ist somit nicht mehr verfügbar. Dagegen wurde eine neue Authentifizierungsmethode *cert* integriert. Diese besagt, dass gegen ein SSL Zertifikat authentifiztiert werden soll. In Verbindung damit wird das Zertifikat mit der Option *clientcert* angegeben. Weiterhin ist die Option *sameuser* bei der Authentifizierungsmethode *ident* entfallen. Die PostgreSQL geht davon aus, dass bei der *ident* Methode der User auf dem System vorhanden ist wenn keine *usermap* in der Datei pg_ident.conf angegeben wurde. Sehr hilfreich ist in der neusten Version auch, dass eine Fehler in der pg_hba.conf nicht erst beim Versuch eines Benutzers sich an der Datenbank anzumelden ausgegeben werden, sondern sobald diese aufgerufen und abgearbeitet wird.

Da der Einsatz von Rechnern mit mehreren CPU's immer häufiger wird, kann nun *pg_restore()* die Wiederherstellung eines Datenbankclusters auf mehrere Threads aufsplitten. Zuvor wurde jede Tabelle und jeder Index nacheinander wieder hergestellt. Durch die *multithread* Möglichkeit kann die Zeit für die Wiederherstellung natürlich erheblich reduziert werden.

Performance und Monitoring

Wie bereits schon erwähnt, wird bei jedem Majorrelease an der Performanceschraube gedreht. So auch dieses mal. Dazu zählen die Bereiche statistic collection, diverse Statement bzw. Join Optimierungen, Subselects, bulk inserts und einige andere. Außerdem wurden die Parameter max_fsm_pages und max_fsm_relations für die Free Space Map aus der postgresql.conf herausgenommen. Die beiden Parameter dienten dazu die Größe der Free Space Map zu steuern. Die Entwickler der PostgreSQL haben allerdings den gesamten Bereich für den Zugriff auf die FSM neu geschrieben, was beide Parameter überflüssig macht. Das Ganze hat zur Folge, dass es eine beträchtliche Performance Steigerung für den VACUUM Prozess gibt.

Für die Überwachung und das Monitoring des Systems gibt es auch Neuerungen. So gibt es jetzt z.B. einen View *pg_stat_user_functions* der Auskunft über die Nutuzung von *user defined functions* liefert oder Informationen über einen bestimmten Prozess durch die Ausgabe von *pg_stat_get_activity(pid) zu erhalten. Schön ist auch die Möglichkeit mit* pg_terminate_backend() einen Datenbankprozess zu terminieren.

Fazit

Die PostgreSQL Datenbank in der Version 8.4 ist nach wie vor die „The world's most advanced open source database“. Es gibt viele Neuerungen die es zum einen dem Programmierer erleichtern seine Statements zu schreiben, es dem Administrator erleichtern den Cluster zu managen und dem Benutzer ermöglichen eine bessere Performance zu bieten. Allerdings ist das bei weitem nicht das Ende. Die Entwickler der PostgreSQL haben schon die Version 8.5 im Visier und das erste Commit Fest für diese Version findet demnächst statt. Sie sind herzlich eingeladen die unterschiedlichen Mailinglists zu abbonieren um Fragen zu stellen, die Weiterentwicklung der PostgreSQL zu beobachten oder sogar mit neuen Features und Improvements aufzuwarten. Alle Mailinglisten finden Sie unter [5] .

Andreas Wenk ist Softwareentwickler und beschäftigt sich seit vielen Jahren mit Datenbanken. Seit drei Jahren intensiv mit der PostgreSQL. Mit Thomas Pfeiffer schreibt er gerade ein PostgreSQL Praxisbuch (Galileo Computing). Sie erreichen Ihn unter a.wenk@netzmeister-st-pauli.de.

Links & Literatur

[1] http://www.postgresql.org/docs/8.4/static/release-8-4.html
[2] http://www.netzmeister-st-pauli.de/php-magazin/6.09/index.html
[3] http://www.postgresql.org/docs/8.4/interactive/queries-with.html
[4] http://www.postgresql.org/docs/8.4/interactive/sql-truncate.html
[5] http://www.postgresql.org/community/lists/