Postgres OnLine Journal: May / June 2009

An in-depth Exploration of the PostgreSQL Open Source Database

Table Of Contents

A Product of Paragon Corporation
http://www.paragoncorporation.com/
http://www.postgresonline.com/

PostGIS 1.3.6 is out and new upcoming PostGIS book

This month is jam packed with a lot of PostGIS news.

PostGIS 1.3.6 is out

PostGIS 1.3.6 has been released. It is mostly a bug fix relase and is the first PostGIS that can be compiled under PostgreSQL 8.4 beta. Details can be found at PostGIS 1.3.6 release notes. We don't have Windows binaries ready yet, but expect to see that in the next week or so.

We are writing a PostGIS Book

Leo and I have been writing a PostGIS in Action book for the past couple of months, and now that it is finally listed on the Manning website, we can talk about it. We are working on our chapter 4 right now. If you are interested in learning PostGIS, check it out. The first chapter is free and with the Manning Early Access Program (MEAP), you can purchase the book now and have great influence on the direction of the book.

The book starts off hopefully with a gentle introduction to OpenGIS Consortium (OGC) spatial databases and concepts in general and PostgreSQL/PostGIS in particular. As we move further into the book, we cover more advanced ground. We plan to cover some of the new PostgreSQL 8.4 features in conjunction with PostGIS, writing stored functions to solve spatial problems and some of the other new exciting stuff and ancillary tools for PostGIS such as PgRouting, Tiger Geocoder, and WKT Raster.

Given all that ground, I suspect our estimate of 325 pages, may be a little low when all is said and done. It is funny that when we started out, we thought to ourselves -- "How can anyone fill up 325 pages." Turns out very easily especially once you start throwing in diagrams and pictures to demonstrate a point. Diagrams are kind of important to have when describing GIS and geometry concepts. So far its been fun and has forced us to sit down and walk thru all the things we took for granted and thought we understood but didn't. You realize just how little you understand when you try to explain something to someone else who really doesn't understand. So perhaps the process of explaining is the greatest of all learning experiences.

PGCon 2009 We'll be speaking

This year we are speaking at PgCon 2009 on PostGIS. Yes Josh Berkus asked politely and we thought, "Why the heck not!" We are looking forward to seeing all the faces we've only read about.

We are diligently preparing our presentations hoping not to make total fools of ourselves.

On a side note -- Paul Ramsey will be giving a workshop on spatial databases at Where 2.0. As you can see he has already started to make a fool of himself by demonstrating evil application architectures that would make any self-respecting database programmer's jaws drop.

PostGIS 1.4.0 not out yet

When will PostGIS 1.4.0 come out? I have no clue. We are still nursing this one. I guess because everyone on the PostGIS team has been so busy with other side activities or making fools of themselves. I'm hoping for late May or early June.

Back to Table Of Contents PostGIS 1.3.6 is out and new upcoming PostGIS book Reader Comments

Creating instance of custom type *Intermediate*

Someone asked me this recently and not playing with custom types much, I'm not sure this is the best solution. Anyrate here it goes. Note this works for PostgreSQL 8.2 and above. Note sure about 8.1 etc. I assume it probably does.

Let us say you created a custom type something like this:

```
CREATE TYPE my_type1 AS
  (name varchar(150),
   rotation_x double precision,
   rotation_y double precision,
   x_pos integer,
   y_pos integer
   );
```

First observe that a row object can be cast to a simple data type, because all table rows are really implemented as types in PostgreSQL. Therefore you can create an instance of this type by doing this:

```
SELECT CAST(ROW('motor', 0.5, 0, 10, 11) As my_type1) As an_instance;
```

Note you could write it using the more abbreviated PostgreSQL specific way, but I prefer the more verbose CAST since it exists in most relational databases

```
SELECT ROW('motor', 0.5, 0, 10, 11)::my_type1 As an_instance;
```

```
     an_instance
---------------------
 (motor,0.5,0,10,11)
```

If you wanted to select an item from this beast you would do:

```
SELECT (CAST(ROW('motor', 0.5, 0, 10, 11) As my_type1)).name As motor_name;
```

```
 motor_name
------------
 motor
(1 row)
```

If you wanted to select all the fields of the type you would do this:

```
SELECT (CAST(ROW('motor', 0.5, 0, 10, 11) As my_type1)).*;
```

```
 name  | rotation_x | rotation_y | x_pos | y_pos
-------+------------+------------+-------+-------
 motor |        0.5 |          0 |    10 |    11
```

Compound types

What if you had a compound type, how could you instantiate such a thing?

```
CREATE TYPE my_type2 AS
```

```
( compound_name varchar(150),
    right_item my_type1,
    left_item my_type1
);

SELECT CAST(ROW('superduper motor', charged, supercharged) as my_type2) As mycompound
FROM
(SELECT CAST(ROW('motor1', 0.5, 0, 10, 11) As my_type1) As charged,
CAST(ROW('motor2', 0, 0.5, 11, 12) As my_type1) As supercharged) As foo


                            mycompound
------------------------------------------------------------------
 ("superduper motor","(motor1,0.5,0,10,11)","(motor2,0,0.5,11,12)")
(1 row)
```

Pulling specific elements out

```
SELECT ((CAST(ROW('superduper motor', charged, supercharged) as my_type2)).right_item).name As
r_charged_name
FROM
(SELECT CAST(ROW('motor1', 0.5, 0, 10, 11) As my_type1) As charged,
CAST(ROW('motor2', 0, 0.5, 11, 12) As my_type1) As supercharged) As foo;

 r_charged_name
------------
 motor1
(1 row)
```

Custom Constructors, Operators for custom types

Scott Bailey created a nice custom type example demonstrating how to create custom types and operators for your custom types.

Timespan Custom Type

Simon Greener over at SpatialDbAdvisor also has some custom types up his sleeve for PostGIS work. Check out his Vectorization: Exploding a linestring or polygon into individual vectors in PostGIS.

Back to Table Of Contents Creating instance of custom type Reader Comments

Restore of functional indexes gotcha *Intermediate*

This has been bugging me for a long time and I finally complained about it and Tom Lane kindly gave a reason for the problem and that its by design and not a bug.

So I thought I would post the situation here without getting into too many embarassing specifics in case others have suffered from a similar fate and can learn from this.

The situation:

- You create a function lets call it **myniftyfunc()** in the public schema.
- Then you create another function that depends on myniftyfunc(), lets call it **mysuperniftyfunc()** also in public schema.
- Then because your function is such a super nifty function, you decide to create a functional index with that super function on your table that sits in mysuperdata schema - **mysuperdata.mysupertable**

Your super nifty function is doing its thing; your table is happy; the planner is spitting out your queries lightning fast using the super nifty index on your super table; The world is good.

One day you decide to restore your nifty database backup and to your chagrin, your nifty index is not there. The planner is no longer happily spitting out your queries lighting fast and everything has come to a painful crawl. Your super nifty index is gone. What happened to super nifty functional index?

I have to admit that I'm the type of person that assumes the public schema is always there and always in search_path and that my assumption is a flawed one. After all the public schema is there by default on new databases for convenience, but one can change it not to be in the search_path and in fact **pg_dump** does just that. So if everything you have is kept in public schema -- you don't run into this particular misfortune. If however you have your functions in public and your tables in different schemas, during restore -- the search path is changed to the schema being restored and your super functional indexes based on super functions that depend on other super functions fail because public is no longer in the search_path.

Below is a simple script to recreate the issue so its clear:

```
CREATE DATABASE superdata;
CREATE OR REPLACE FUNCTION myniftyfunc(myint integer) RETURNS integer AS
 $$ SELECT 1 + $1;$$
LANGUAGE 'sql' IMMUTABLE;

CREATE OR REPLACE FUNCTION mysuperniftyfunc(myint integer) RETURNS integer AS
$$ SELECT myniftyfunc($1); $$
LANGUAGE 'sql' IMMUTABLE;

CREATE SCHEMA mysuperdata;
CREATE TABLE mysuperdata.mysupertable(sid integer PRIMARY KEY, super_key integer);
CREATE INDEX idx_mysupertable_super_index
   ON mysuperdata.mysupertable USING btree (mysuperniftyfunc(super_key));

INSERT INTO mysuperdata.mysupertable(sid,super_key)
VALUES(1,1);

--Backup superdata
"C:\Program files\postgresql\8.3\bin\pg_dump" --host=localhost --port=5432 --username=postgres --format=plain
--verbose --file="C:\superdata.sql" superdata

--Restore
"C:\Program files\postgresql\8.3\bin\psql" -U postgres -h localhost -p 5432 -d superdata2 -f "C:\superduper.
sql"

--Get non-super cool error
psql:C:/superduper.sql:30: ERROR:  function myniftyfunc(integer) does not exist
```

```
LINE 1:  SELECT myniftyfunc($1);
                    ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
QUERY:   SELECT myniftyfunc($1);
CONTEXT:  SQL function "mysuperniftyfunc" during startup
```

Normally I do my backup in compressed format, but did it in plain to demonstrate the problem and here is what pg_dump produces.

```
--
-- PostgreSQL database dump
--

-- Started on 2009-06-18 21:45:59

SET statement_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = off;
SET check_function_bodies = false;
SET client_min_messages = warning;
SET escape_string_warning = off;

--
-- TOC entry 6 (class 2615 OID 1086067)
-- Name: mysuperdata; Type: SCHEMA; Schema: -; Owner: postgres
--

CREATE SCHEMA mysuperdata;


ALTER SCHEMA mysuperdata OWNER TO postgres;

SET search_path = public, pg_catalog;

--
-- TOC entry 21 (class 1255 OID 1086065)
-- Dependencies: 3
-- Name: myniftyfunc(integer); Type: FUNCTION; Schema: public; Owner: postgres
--

CREATE FUNCTION myniftyfunc(myint integer) RETURNS integer
        LANGUAGE sql IMMUTABLE
        AS $_$ SELECT 1 + $1;$_$;


ALTER FUNCTION public.myniftyfunc(myint integer) OWNER TO postgres;

--
-- TOC entry 22 (class 1255 OID 1086066)
-- Dependencies: 3
-- Name: mysuperniftyfunc(integer); Type: FUNCTION; Schema: public; Owner: postgres
--

CREATE FUNCTION mysuperniftyfunc(myint integer) RETURNS integer
        LANGUAGE sql IMMUTABLE
        AS $_$ SELECT myniftyfunc($1); $_$;


ALTER FUNCTION public.mysuperniftyfunc(myint integer) OWNER TO postgres;

SET search_path = mysuperdata, pg_catalog;
-- this is not a bug, but would be super
cool if public or whatever the default search path of the database was in here.

SET default_tablespace = '';

SET default_with_oids = false;
```

```
--
-- TOC entry 1465 (class 1259 OID 1086068)
-- Dependencies: 6
-- Name: mysupertable; Type: TABLE; Schema: mysuperdata; Owner: postgres; Tablespace:
--

CREATE TABLE mysupertable (
        sid integer NOT NULL,
        super_key integer
);


ALTER TABLE mysuperdata.mysupertable OWNER TO postgres;

--
-- TOC entry 1735 (class 0 OID 1086068)
-- Dependencies: 1465
-- Data for Name: mysupertable; Type: TABLE DATA; Schema: mysuperdata; Owner: postgres
--

COPY mysupertable (sid, super_key) FROM stdin;
1       1
\.

--
-- TOC entry 1734 (class 2606 OID 1086072)
-- Dependencies: 1465 1465
-- Name: mysupertable_pkey; Type: CONSTRAINT; Schema: mysuperdata; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY mysupertable
        ADD CONSTRAINT mysupertable_pkey PRIMARY KEY (sid);


--
-- TOC entry 1732 (class 1259 OID 1086073)
-- Dependencies: 22 1465
-- Name: idx_mysupertable_super_index; Type: INDEX; Schema: mysuperdata; Owner: postgres; Tablespace:
--

CREATE INDEX idx_mysupertable_super_index ON mysupertable USING btree (public.mysuperniftyfunc(super_key));


--
-- TOC entry 1740 (class 0 OID 0)
-- Dependencies: 3
-- Name: public; Type: ACL; Schema: -; Owner: postgres
--

REVOKE ALL ON SCHEMA public FROM PUBLIC;
REVOKE ALL ON SCHEMA public FROM postgres;
GRANT ALL ON SCHEMA public TO postgres;
GRANT ALL ON SCHEMA public TO PUBLIC;


-- Completed on 2009-06-18 21:45:59

--
-- PostgreSQL database dump complete
--
```

Solution (workaround):

The work around for this situation is to either explicitly qualify the functions you are using within another or for PostgreSQL 8.3 -- add to your function set search_path=public

Solution 1:

```
CREATE OR REPLACE FUNCTION mysuperniftyfunc(myint integer) RETURNS integer AS
```

```
$$ SELECT public.myniftyfunc($1); $$
LANGUAGE 'sql' IMMUTABLE;
```

Solution 2: (only works for PostgreSQL 8.3+)

```
CREATE OR REPLACE FUNCTION mysuperniftyfunc(myint integer) RETURNS integer AS
$$ SELECT myniftyfunc($1); $$
LANGUAGE 'sql' IMMUTABLE;

ALTER FUNCTION mysuperniftyfunc(integer) SET search_path=public;
```

Of course neither of these solutions is particularly satisfying if you are a package author. If you are and that is how this mess started. You want people to be able to install your functions in whatever schema they like and if they wanted to use it globally they would add it to their database default search path. Though that is arguable. Perhaps all packages should live in specifically named schemas.

Back to Table Of Contents Restore of functional indexes gotcha Reader Comments

Running totals and sums using PostgreSQL 8.4 Windowing functions *Intermediate*

One thing that is pretty neat about windowing functions in PostgreSQL 8.4 aside from built-in windowing functions (row_number (), rank(), lead(), lag(), dense_rank(), percent_rank(), cume_dist(), first_value, last_value, nth_value) as documented in the manual Windowing Functions is that you can use windows with most aggregate functions (built-in or custom defined) as well as define your own specific windowing functions. In a later article, we'll demonstrate creating custom windowing functions.

In our PGCon 2009 PostGIS presentation one of the last slides demonstrates using lead() and lag() windowing functions to show a family's income level in the same record with the income levels of the next door neighbors in the fictitious town we created. This is not terribly useful unless you live in a somewhat dysfunctional neighborhood where everyone is concerned about how rich their neighbors are compared to themselves. Our town was very dysfunctional but mostly geographically dysfunctional. We will have much more useful use cases of this as applied to GIS in our upcoming PostGIS in Action book.

Hitoshi Harada and David Fetter did a presentation of this in PGCon 2009 which sadly we missed since we were giving our own presentation. Check out the PGCon2009 PostgreSQL 8.4 Windowing Functions Video. Also check out the slides at Introducing Windowing Functions.

Those who have used SQL Server 2005+, Oracle or IBM DBII are probably familar or have run into examples of Windowing functions in those products. Windowing in PostgreSQL 8.4 works more or less the same way. In a prior article, we demonstrated how to return running totals and sums using rudimentary SQL. To precelebrate the eminent arrival of PostgreSQL 8.4 and the current PostgreSQL 8.4 beta 2 release, we shall demonstrate the same exercise using the new ANSI SQL:2003 Windowing functionality built into the upcoming PostgreSQL 8.4.

```
--First create our table and dataset
CREATE TABLE orders(order_id serial PRIMARY KEY, customer_id integer,
      order_datetime timestamp, order_total numeric(10,2));

INSERT INTO orders(customer_id, order_datetime, order_total)
VALUES (1,'2009-05-01 10:00 AM', 500),
    (1,'2009-05-15 11:00 AM', 650),
    (2,'2009-05-11 11:00 PM', 100),
    (2,'2009-05-12 11:00 PM', 5),
       (3,'2009-04-11 11:00 PM', 100),
          (1,'2009-05-20 11:00 AM', 3);
```

Recall in our previous example we wanted our running total column to include the total of all purchases made by the customer prior to the current purchase. We accomplished this with a self join.

```
--non windowing version
SELECT n.customer_id, n.order_id, n.order_total,
   COALESCE(SUM(o.order_total),0) As past_order_total
FROM orders AS n LEFT JOIN orders AS o
   ON (o.customer_id = n.customer_id
        AND n.order_datetime > o.order_datetime)
GROUP BY n.customer_id, n.order_datetime, n.order_id, n.order_total
ORDER BY n.customer_id, n.order_datetime, n.order_id;
```

```
 customer_id | order_id | order_total | past_order_total
-------------+----------+-------------+---------------
           1 |        1 |      500.00 |             0.00
           1 |        2 |      650.00 |           500.00
           1 |        5 |        3.00 |          1150.00
           2 |        3 |      100.00 |             0.00
           2 |        4 |        5.00 |           100.00
           3 |        6 |      100.00 |             0.00
```

```
          1 |        8 |      650.00 |        500.00
          1 |       12 |        3.00 |       1150.00
          2 |        9 |      100.00 |             0
          2 |       10 |        5.00 |        100.00
          3 |       11 |      100.00 |             0
(6 rows)
```

The above shows an approach to write running sums using self-joins. If you do not have windowing functions this is a common approach. With windowing functions, you have the option of doing this. Note in our example we want to exclude the current order from our running total sum.

```
SELECT n.customer_id, n.order_id, n.order_total,
   SUM(n.order_total)
      OVER (PARTITION BY n.customer_id
               ORDER BY n.order_datetime) - n.order_total As past_order_total
FROM orders AS n
ORDER BY n.customer_id, n.order_datetime, n.order_id;


 customer_id | order_id | order_total | past_order_total
-------------+----------+-------------+------------------
           1 |        7 |      500.00 |             0.00
           1 |        8 |      650.00 |           500.00
           1 |       12 |        3.00 |          1150.00
           2 |        9 |      100.00 |             0.00
           2 |       10 |        5.00 |           100.00
           3 |       11 |      100.00 |             0.00
```

An equal statement to the above that uses pure windowing with no subtract would probably use something like a RANGE BETWEEN x PRECEEDING AND Y FOLLOWING etc. Unfortunately PostgreSQL 8.4 doesn't appear to support the Oracle like:
BETWEEN x PRECEDING AND y FOLLOWING
BETWEEN x PRECEDING AND y PRECEDING
As described in Oracle Windowing functions therefore we had to put the - n.order_total to achieve more or less the same result as our previous. Also note we are making the assumption in our prior version that a given customer will not place 2 orders at the same time. Otherwise the results would be different between the 2.

Named windows

One other feature that is nice about window functions is that you can name windows and reuse them, also have windowing different from your basic sort order, and have multiple windowing functions in the same query. Below is a demonstration of this where we sequentially order our orders by order_datetime in descending order and then reuse our existing window to get an ordering for each customer partitioning such that we have a row_number recount for each customer by ascending time.

```
--multiple windows and multiple window function calls
SELECT row_number() OVER(ORDER BY order_datetime DESC) As rtime_d,
n.customer_id, row_number() OVER(window_custtime) As cr_num, n.order_id, n.order_total,
   SUM(n.order_total) OVER (window_custtime) - n.order_total As past_order_total
FROM orders AS n
WINDOW window_custtime AS (PARTITION BY n.customer_id
                    ORDER BY n.order_datetime)
ORDER BY  n.order_datetime, n.order_id, n.customer_id;


 rtime_d | customer_id | cr_num | order_id | order_total | past_order_total
---------+-------------+--------+----------+-------------+------------------
       6 |           3 |      1 |       11 |      100.00 |             0.00
       5 |           1 |      1 |        7 |      500.00 |             0.00
       4 |           2 |      1 |        9 |      100.00 |             0.00
       3 |           2 |      2 |       10 |        5.00 |           100.00
       2 |           1 |      2 |        8 |      650.00 |           500.00
       1 |           1 |      3 |       12 |        3.00 |          1150.00
```

```
(6 rows)
```

Below is the PgAdmin graphical explain plan from PgAdmin III 1.10 Beta 3

Observe that using the same window results in one Windows Agg which is a bit clearer in the next example.

```
--using same window twice results in only one windowagg call
SELECT row_number()  OVER(window_custtime) As rtime_d,
n.customer_id, lead(order_id) OVER(window_custtime) As cr_num, n.order_id, n.order_total
FROM orders AS n
WINDOW window_custtime AS (PARTITION BY n.customer_id
                    ORDER BY n.order_datetime)
ORDER BY  n.order_datetime, n.order_id, n.customer_id;
```

```
 rtime_d | customer_id | cr_num | order_id | order_total
---------+-------------+--------+----------+-------------
       1 |           3 |        |        5 |      100.00
       1 |           1 |      2 |        1 |      500.00
       1 |           2 |      4 |        3 |      100.00
       2 |           2 |        |        4 |        5.00
       2 |           1 |      6 |        2 |      650.00
       3 |           1 |        |        6 |        3.00
(6 rows)
```

Back to Table Of Contents

Planner Statistics ***Beginner***

You'll often hear the term planner statistics thrown around by database geeks. *Did you update your statistics*. This lingo isn't even limited to PostgreSQL, but is part and parcel to how most decent databases work. For example in PostgreSQL you do a `vacuum analyze` to update your planner statistics in addition to cleaning up dead space. In SQL Server you do an `UPDATE STATISTICS`. In MySQL you do an `ANALYZE TABLE` or a more invasive `OPTIMIZE TABLE`.

Normally all this "update your stats so your planner can be happy" is usually unnecessary unless you just did a bulk load or a bulk delete or you are noticing your queries are suddenly slowing down. These stat things are generally updated behind the scenes by most databases on an as needed basis.

What makes SQL really interesting and a bit different from procedural languages is that it is declarative (like functional and logical programming languages) and relies on the database planner to come up with strategies for navigating the data. Its strategy is not fixed as it is in procedural languages. A big part of this strategy is decided on by the query planner which looks at distributions of data. Given different WHERE conditions for similar queries, it could come up with vastly different strategies if one value has a significantly higher distribution in a table than another. This is also the mystery of why it sometimes refuses to use an index on a field because it has decided a table scan is more efficient and also why some people consider HINTS evil because they pollute the imperative nature of the language.

So what do these stat things look like exactly? Well you can see these things in PostgreSQL by interrogating the pg_stats view. The pg_stats view exists for all currently supported versions of PostgreSQL (7.4+). Not sure if it existed in prior versions.

Details of the fields are here: pg_stats fields

Below is a sample query from one of our test databases and the output.

```
SELECT attname As colname, n_distinct,
            array_to_string(most_common_vals, E'\n') AS common_vals,
            array_to_string(most_common_freqs, E'\n') As dist_freq
 FROM pg_stats
 WHERE schemaname = 'assets' and tablename = 'land';
```

```
        colname     | n_distinct |    common_vals    | dist_freq
-----------------+------------+-------------------+-----------
 pid             |         -1 |                   |
 land_name       |         -1 |                   |
 land_type       |         13 | park              | 0.104587
                                                       : college           : 0.0990826
                                                       : highschool        : 0.0899083
                                                       : hospital          : 0.0899083
                                                       : 2 family          : 0.0862385
                                                       : 3 family          : 0.0825688
                                                       : elementary school : 0.0788991
                                                       : commercial        : 0.0770642
                                                       : government        : 0.0752294
                                                       : condo             : 0.0697248
                                                       : police station    : 0.0623853
                                                       : 1 family          : 0.0458716
```

```
                                                    : vacant land       : 0.0385321
 the_geom        |         -1 |                    |
 land_type_other |  -0.166972 |                    |
(5 rows)
```

So the above tells us that land_type has 13 distinct values with park the most common value in this table with 10 percent of the sampled data being that and that pid, the_geom and name are fairly distinct.

Keep in mind sampling may not be the all the records and in general is not and sampling can be indirectly controlled per column with ALTER TABLE *tablename* ALTER COLUMN *column_name*.

For columns with no specific sampling set, you can check the default with:

```
show default_statistics_target;
```

The statistics setting controls the number of items that can be stored in the common_vals, frequencies, and histogram_bounds fields arrays for each table and that would indirectly increase the sampling of records. Prior to 8.4 -- this was defaulted to 10 though can be increased by changing the postgresql.conf file. At 8.4 this the default setting was increased to 100.

Observe what happens to our data set when we up the statistics to as high as we can set for land_type_other and then reanalyze.

```
ALTER TABLE assets.land
 ALTER COLUMN land_type_other SET STATISTICS 10000;

 vacuum analyze assets.land;

 SELECT attname As colname, n_distinct,
           array_to_string(most_common_vals, E'\n') AS common_vals,
           array_to_string(most_common_freqs, E'\n') As dist_freq
FROM pg_stats
WHERE schemaname = 'assets' and tablename = 'land';
```

```
         colname     | n_distinct |           common_vals            | dist_freq
-----------------+------------+----------------------------------+-------------
 land_type       |         13 | park                             | 0.104587
                                                                 : college                          : 0.0990826
 the_geom        |         -1 |                                  |
 land_type_other |  -0.166972 | {"3 family",college}             | 0.00550459
                                                                 : {condo,park}                     : 0.00550459
                                                                 : {highschool,"police station"}    : 0.00550459
                                                                 : {"1 family",government,hospital} : 0.00366972
                                                                 : {"1 family",hospital}            : 0.00366972
                                                                 : {"2 family","police station"}    : 0.00366972
                                                                 : {college,commercial}             : 0.00366972
                                                                 : {college,government}             : 0.00366972
                                                                 : {college,"police station"}       : 0.00366972
                                                                 : {commercial,condo}               : 0.00366972
                                                                 : {government}                     : 0.00366972
                                                                 : {highschool,park}                : 0.00366972
                                                                 : {hospital,park}                  : 0.00366972
                                                                 : {park,"police station"}          : 0.00366972
 land_name       |         -1 |                                  |
 pid             |         -1 |                                  |
```

Back to Table Of Contents

Managing disk space using table spaces ***Beginner***

We have covered this briefly before, but its an important enough concept to cover again in more detail.

Problem: You are running out of disk space on the drive you keep PostgreSQL data on

Solution:

Create a new tablespace on a separate drive and move existing tables to it, or create a new tablespace and use for future tables.

What is a tablespace and how to create a tablespace

A tablespace in PostgreSQL is similar to a tablespace in Oracle and a filegroup in SQL Server. It segments a piece of physical disk space for use by the PostgreSQL process for holding data. Below are steps to creating a new tablespace. Tablespaces have existed since PostgreSQL 8.0.

More about tablespaces in PostgreSQL is outlined in the manual PostgreSQL 8.3 tablespaces

While it is possible to create a table index on a different tablespace from the table, we won't be covering that.

Below are steps to creating one

1. First create a folder on an available disk in your filesystem using an filesystem server administrative login
2. Next give full rights to the postgres server account (the one the daemon process runs under) or you can change the owner of the folder to the postgres account (in linux you can use chown postgres and on windows just use the administrative properties panel for that folder.
3. Then launch psql or pgadmin III and connect as a super user
4. At query window type:

   For Linux you will have something like this

   ```
   CREATE TABLESPACE mynewtablespace LOCATION /path/to/mynewfolder
   ```

   For windows you will have something like this:

   ```
   CREATE TABLESPACE mynewtablespace LOCATION 'D:/mynewfolder';
   ```

Moving existing tables to the new tablespace

You move existing tables with the ALTER TABLE command. You can do the below replacing the my... variables with your specific one. mytableschema if you are not using schemas for logical partitioning would simply be **public**.

1. Again connect via psql or PgAdmin III
2. ALTER TABLE *mytableschema.mytable* SET TABLESPACE *mynewtablespace*

Setting default location for new tables

As of PostgreSQL 8.2, PostgreSQL provides many options for setting the default location of newly created tables

- You can default location based on the database using:
  ```
  ALTER DATABASE mydatabase SET default_tablespace = mynewtablespace
  ```
  If you happened to have used mixed case when creating your database, you will need to quote the database name e.g

```
ALTER DATABASE "My_Database" SET default_tablespace = mynewtablespace;
```

- based on the user creating the table.
```
ALTER ROLE someuser SET default_tablespace = mynewtablespace;
```
- Or temporarily for current session while you are creating a batch of tables using
```
SET default_tablespace = mynewtablespace;
```

Back to Table Of Contents Managing disk space using table spaces Reader Comments

Reader Comments

PostGIS 1.3.6 is out and new upcoming PostGIS book

BostonGIS Blog
As many people may have heard and as Mateusz kindly already commented on; Leo and I already mentioned in our Postgres OnLine Journal site, we are writing a book specifically focused on everything PostGIS. The hard copy version is do out around January o

---

Creating instance of custom type

Scott Bailey
Oh NOW you do a write up about it. I was working through this a few weeks ago and there really wasn't much info out there outside of the manual.

I just did a write up yesterday about adding a custom timespan type with associated constructors, functions and operators.

http://scottrbailey.wordpress.com/2009/05/19/timespan-postgresql/

Regina
Scott,

Neat. That was what I was going to cover next. custom constructor functions and operators, but since you already have - I'll just link to yours.

Restore of functional indexes gotcha

gregj
wouldn't a workaround , be forcing pg_dump to dump schema.functionname() always ?

Regina
If I understand you correctly, no that doesn't help. It is installing the functions in the right schema and the index can see the function perfectly fine. It is even smart enough to prefix the function with public in the index. In fact it works if you have no data in your table.

This particular issue happens because when pg_restore goes to build the index on the table in another schema, the search_path = yourtable_schema,pg_catalog

So since the functions above live in public (both functions actually do get installed before the tables but they are not in the same schema as the table or in pg_catalog), and the indexing process starts, the second function can no longer see the first function since its no longer in the search path. So the error message aside from not being super cool is kinda confusing because it makes you think the first function was never created. Its there just not visible by the second during the indexing process.

gregj
but than, if function name _in index_ was prefixed with schema name, it would find it!

Regina
Ah but it is - look at the pg_backup output:

CREATE INDEX idx_mysupertable_super_index ON mysupertable USING btree (public.mysuperniftyfunc(super_key));

but its when the create index process tries to call mysuperniftyfunc

mysuperniftyfunc can't find myniftyfunc

and breaks right here

SELECT myniftyfunc($1)

gregj
hmm, that's odd.
I would honestly think, that when you do specify schema explicitly - it doesn't need search paths at all. Kinda like FSs.

Regina
It doesn't unless thecalled function calls another function that is not schema qualified. So its a somewhat isolated issue. Except in normal database workload the function works fine since the schemas in use by the function are part of the the search_path of the db. Restore changes that so the default schemas are not necessarily in the search_path

matt peters
In your example above, add the following function:

```
create or replace function mysuperdata.myniftyfunc(myint integer) returns integer as
$_$
select mod($1*200,1000);
$_$
LANGUAGE sql immutable;
```

And lets give it some more data to make it interesting:

```
insert into mysupertable (sid, super_key) values(1,1), (2,200), (3,300);
```

Now, go ahead and pg_dump and restore to a database named tmp2. It works!

Now, ready for a head-slapper? Execute the following:

```
tmp2=# set search_path to public, mysuperdata;

SET

tmp2=# select * from mysupertable where public.mysuperniftyfunc(super_key) = 201;

sid | super_key

-----+-----------

2 | 200

(1 row)


tmp2=# explain select * from mysupertable where public.mysuperniftyfunc(super_key) = 201;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on mysupertable (cost=0.00..1.04 rows=1 width=8)

Filter: ((1 + super_key) = 201)

(2 rows)


tmp2=# set search_path to mysuperdata, public;

SET

tmp2=# select * from mysupertable where public.mysuperniftyfunc(super_key) = 201;

sid | super_key

-----+-----------

(0 rows)


tmp2=# explain select * from mysupertable where public.mysuperniftyfunc(super_key) = 201;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on mysupertable (cost=0.00..1.05 rows=1 width=8)

Filter: (mod((super_key * 200), 1000) = 201)

(2 rows)
```

WOOOPS! This is not a build issue, it is a design flaw. Once you introduce schemas, you have to assume that search paths will be different. Your supernifty proc is therefore dependent on the user's search path. What I wonder is, if the data is inserted by different users with different search paths, what happens to the index? I imagine it is functionally corrupt.

So I changed my search_path, and inserted a bunch of records:

insert into mysupertable (sid, super_key) select sid, sid*100 from generate_series(4,500) ser(sid);

Then I changed it again and inserted another bunch of records:

insert into mysupertable (sid, super_key) select sid, sid*100 from generate_series(501,1000) ser(sid);

Then I VACUUM ANALYZE the table and did an explain. Now that the table is larger, and the cost of a sequence scan is more than using an index, the index showed up in the plan:

explain select * from mysupertable where mysuperniftyfunc(super_key) = mysuperniftyfunc(200);

QUERY PLAN

Seq Scan on mysupertable (cost=0.00..22.50 rows=5 width=8)

Filter: (mod((super_key * 200), 1000) = 0)

(2 rows)

Wait! that is the wrong function! Change search_paths and try again:

explain select * from mysupertable where mysuperniftyfunc(super_key) = mysuperniftyfunc(200);

QUERY PLAN

Seq Scan on mysupertable (cost=0.00..17.49 rows=4 width=8)

Filter: ((1 + super_key) = 201)

(2 rows)

Right function, wrong plan!

Lets do a VACUUM ANALYZE again, try the explain again:

explain select * from mysupertable where mysuperniftyfunc(super_key) = mysuperniftyfunc(200);

QUERY PLAN

Index Scan using idx_mysupertable_super_index on mysupertable (cost=0.00..8.27 rows=1 width=8)

Index Cond: ((1 + super_key) = 201)

(2 rows)

Do you follow what is happening? the statistics are being influenced by the search path. (So are the search results, by the way - try it!).

If you are a package author, you have bigger problems than PostgreSQL not rebuilding properly. You have created a situation where someone else's package and the unpredictability of any given user's search path will result in a) bad performance; b) the wrong data coming back; c) a functionally corrupt index(?)

The problem is that the design above implements schema, but only partially - you left a big gaping design hole in mysuperniftyfunc. There ARE design scenarios where it makes sense to NOT use explicit schema within functions, but they are generally the exception, and not the rule. In this scenario, I used a foil (mysuperdata.myniftyfunc) to illustrate the design flaw - but it was there from the beginning. The failure to restore is just one of the problems with the code.

Regina
Matt,
Very good points. I'm aware of these, I guess I was looking for a having my cake and eating it too kind of solution.

Normally and I'm sure I am different from other users. I use schemas to logically segregate my data (I don't go around changing search paths willy nilly and yes users can have their search paths individually set -- but for global functions I always make sure users have public or whatever in their paths just like pgcatalog is always there -- you can't get rid of it) and I define a specific search path for my database. Adding in the paths I want where I want tables to be used without schema qualification -- because its annoying to schema qualify stuff all the time and not terribly portable and hard to explain to users.

So I guess my basic point is if my database works happily given the search_paths I define, I expect it to work happily when I restore it as well. Yes I am a stupid user for expecting these things and I see very well the flaw in my logic for wanting these

things. But it doesn't change the fact that this is not a solution just more problems.

I don't really want to have to force people to install functions in a specific schema. Now of course if PostgreSQL had something like a
"this function references other functions in the schema it is stored in" which is a quite common scenario especially for packages (without having to explicitly define the schema these functions should be stored in), that would solve all my problems and I'm sure many package authors as well and would be logically consistent and not break your nicely elaboret example.

But PostgreSQL doesn't to my knowledge support this idea of the schema that the function is stored in can reference things in the schema it is in and that is the main problem I have and why I'm a very frustrated user.

Managing disk space using table spaces

Bernd Helmle

Since PostgreSQL 8.3 you can use the GUC temp_tablespaces to create temporary objects there, too. This also allows to specify locations where temporary files for sorting and so on can be located and separated from database storage. Multiple temp tablespaces can be configured for "balancing".
