Postgres OnLine Journal: October 2008

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/

From the Editors

When it rains it Drizzles

MySQL is turning out to be one big soap opera as far as I can tell and as Bruce Momjian has mentioned.

Lets go over some of the interesting episodes of this saga:

1. First Sun buys MySQL
2. Falcon storage engine creator Jim Starkey leaves MySQL/Sun
3. Brian Aker heads Drizzle which is a fork of MySQL that hopes to be a stream-lined implementation of MySQL that leaves out all that nonsense we don't need such as views and stored procs and targeting it self for running on the cloud, optimizing for massive concurrency, and ease of install. I presume Brian still works for MySQL/Sun though.
4. Michael "Monty" Widenius Leaves MySQL and Sun
   to help with this Drizzle thing, but will still be working on Maria according to Brian (see comment below).
5. And now Jay Pipes leaves as well to work on Drizzle. He will still be a Sun staff engineer on Drizzle, but not the MySQL community leader.

I'm not sure what all these things say about the stability of the MySQL core. I mean should I stick with MySQL 5 or run for the Drizzle, but I think I'll stick with PostgreSQL where ever I can. PostgreSQL may not be quite as interesting from a soap opera perspective, but it seems a tad bit more dependable and I really like my views and stored functions.

Back to Table Of Contents When it rains it Drizzles Reader Comments

How to delete many functions *Intermediate*

There are a lot of functions in PostgreSQL and for the most part, they are nicely tucked away in the pg_catalog schema so they don't get mixed with your functions. There are a lot of contrib modules though and if you use a few of the big ones and just dump them in the public schema, the function list gets overwhelming and you have a hard time finding your own functions. To keep our sanity we tend to create a schema called util or something like that where we stuff our own personal functions for easy navigation and for larger contribs, we may put these in a separate schema altogether. Every once in a while, we screw up and put the functions in the wrong schema. Deleting these can become painful if there are a lot to delete.

Problem: How do you delete a butt load of functions without working up a sweat?

Solution:

Use the below code cautiously. We start off with our general hack of writing an sql statement to write lots of sql statements. We had hoped to use the more generic information_schema for this exercise, but the closest table we could find *information_schema.routines*, lists the names of the functions but not the arguments. You need the arguments in your drop since PostgreSQL supports function argument overloading.

```
SELECT 'DROP FUNCTION ' || ns.nspname || '.' || proname || '(' || oidvectortypes(proargtypes) || ');'
FROM pg_proc INNER JOIN pg_namespace ns ON (pg_proc.pronamespace = ns.oid)
WHERE ns.nspname = 'my_messed_up_schema'  order by proname;
```

The above will generate but not execute code that looks something like below so you can inspect the drops before executing:

```
DROP FUNCTION my_messed_up_schema.funcabc(int4,int4);
DROP FUNCTION my_messed_up_schema.funcdef(int4,date);
```

Back to Table Of Contents How to delete many functions Reader Comments

Why is my index not being used ***Beginner***

The age old question of *why or why is my table index not being used* is probably the most common question that ever gets asked even by expert database users.

In this brief article we will cover the most common reasons and try to order by statistical significance.

For the uninitiated, how do you know when an index is not being used? You use EXPLAIN, EXPLAIN ANALYZE or PgAdmin's nifty graphical Explain plan described in Reading PgAdmin Graphical Explain Plans.

Okay your query is not using an index, why and what can you do about it?

1. **Problem**: Outdated stats. Now this is less common of an issue if you have auto-vacuuming turned on, but if you have recently bulk loaded a table or added new indexes and then try to do a query, then the statistics may be out of date.

   Solution:
   ```
   vacuum analyze verbose sometable
   ```
   I add the verbose in because it is fun to see what vacuuming is doing and if you are as impatient as I am and you tables are big, its nice to get some confirmation that something indeed is happening. You can also just
   ```
   vacuum analyze verbose
   ```
   if you want to run against all tables.
2. **Problem**: The planner has decided its faster to do a table scan than an index scan : This can happen if a) your table is relatively small, or the field you are indexing has a lot of duplicates. **Solution**: Case in point, boolean fields are not terribly useful to index since 50% of your data is one thing and 50% is another. However they are good candidates to use for Partial indexes e.g. to only index data that is active.
3. **Problem**: You set up an index that is incompatible with how you are actually filtering a field. There are a couple of variants of this situation. The old

   -
     ```
     LIKE '%me'
     ```
     will never use an index, but
     ```
     LIKE 'me%'
     ```
     can possibly use an index.
   - The upper lower trap - you defined your index like:
     ```
     CREATE INDEX idx_faults_name ON faults USING btree(fault_name);
     ```
     , But you are running a query like this:
     ```
     SELECT * FROM faults where UPPER(fault_name) LIKE 'CAR%'
     ```
     Possible fix:
     ```
     CREATE INDEX idx_faults_name ON faults USING btree(upper(fault_name));
     ```
   - This is one that I wasn't even aware of. It is always nice to read about other peoples problems via newgroups because you discover all these problems you never even knew you had. If your server was init with a non-C locale, doing the above still doesn't work. This came up in Pgsql-novice newsgroup recently and Tom Lane provided an answer which works. This I suspect bites more people than is known. The solution Possible fix:
     ```
     CREATE INDEX idx_faults_uname_varchar_pattern ON faults USING btree(upper(fault_name)
     varchar_pattern_ops);
     ```
     However even with the above solution it appears you may still need the variant below for exact matches and IN clauses:

     We haven't quite proven if this an issue with database encoding, data itself or difference in versions of 8.2 vs 8.3. It seems for 8.3 for our particular dataset in UTF-8 the below is still needed for exact matches, however for similar use-case in a 8.2 database in SQL-ASCII, the varchar_pattern_ops is enough for both exact and LIKE matches.
     ```
     CREATE INDEX idx_faults_uname ON faults USING btree(upper(fault_name));
     ```
     e.g.
     ```
     SELECT fault_name from  faults
     WHERE upper(fault_name) IN('CASCADIA ABDUCTION', 'CABIN FEVER');
     ```
   - Dum newbie user - Just incompatible data type. E.g. doing something like creating an index on a date field and then doing a text compare with dates by casting your date to text.
4. Not all indexes can be used. Although PostgreSQL 8.1+ versions support what is known as Bitmap Index Scan, which allows multiple indexes on a table to be used in a query by creating in-memory bitmap indexes. If you have got many indexes, don't expect all possible candidate indexes to be used. Sometimes a table scan is just more efficient.
5. **Problem**: The planner isn't perfect. **Solution** Cry and pray for a brighter day. Actually I've been pretty impressed with PostgreSQL's planning abilities compared to other databases. Some would say, *If only I could provide hints, I could make this faster.* I tend to think hints are a bad idea and the best solution is just to make the planner better. The problem with hints is they take away one beauty of databases. That is the idea that the database knows the state of the data better than you do and constantly updates that knowledge.

Hints can quickly become stale where as a well-primed planner, will constantly be changing strategy as the database changes and that is really what makes database programming unique among various modes of programming.

Back to Table Of Contents Why is my index not being used Reader Comments

Simulating Row Number in PostgreSQL Pre 8.4 ***Intermediate***

PostgreSQL 8.4 will have a ROW_NUMBER() windowing function so this little hack will hopefully be unnecessary when 8.4 is in production.

Getting back to this exercise, this was actually inspired by Hubert's recent article Tips N' Tricks - setting field based on order. Why this inspired me, I guess because it stirred up memories about the often forgotten utility of arrays in PostgreSQL and I thought it would answer a question that was haunting me - *How do I assign sequential numbers to a list?*. The article just didn't quite read the way I expected it to and actually was answering another question I cared much less about, but it did get the juices flowing. So without much further ado.

Problem:

You have data that looks like this:

```
name_id         last_name               first_name
jjones          Jones                   John
psmith          Smith                   Paul
wgates          Gates                   William
wgrant          Grant                   Wallace
```

And someone asks you to output it like this:

```
row_number      name_id                 last_name               first_name
1               wgates                  Gates                   William
2               wgrant                  Grant                   Wallace
3               jjones                  Jones                   John
4               psmith                  Smith                   Paul
```

```
--CREATE test data
CREATE TABLE people(name_id varchar(50) PRIMARY KEY, last_name varchar(50),
   first_name varchar(50));
INSERT INTO people(name_id, last_name, first_name)
(VALUES ('jjones', 'Jones', 'John'),
('psmith', 'Smith', 'Paul'),
('wgates', 'Gates', 'William')
   , ('wgrant', 'Grant', 'Wallace'));
```

Solutions: Here we propose 3 solutions and which you choose is based on preference, use-case and experience.

```
--Approach 1: The all in one WTF
--  Advantage: 1 step and no temp junk created
--  so safe to nest into other subqueries and reuse
--- Disadvantage: Messy to read,
--      could be slow for large sets,
--      and relies on a primary key
SELECT row_number, oldtable.*
FROM (SELECT * FROM people) As oldtable
CROSS JOIN (SELECT ARRAY(SELECT name_id
      FROM people
      ORDER BY last_name, first_name) As id)  AS oldids
CROSS JOIN generate_series(1, (SELECT COUNT(*)
              FROM people)) AS row_number
WHERE oldids.id[row_number] =  oldtable.name_id
ORDER BY row_number;

--Approach 2: Closer to Hubert's Examples
--Advantage: Easy to read - intuitive, doesn't rely on a primary key
--Disadvantage: Creates temp junk in the db
--      which means reusing in same session you must drop
--  and using in nested subquery results may be unpredictable
--I prefer the WTF
--I don't know what it is about explicitly creating stuff in the
```

```
--database even if it lasts for that one session
-- just irritates me - granted its much simpler and probably faster
CREATE TEMP sequence temp_seq;
SELECT nextval('temp_seq') As row_number, oldtable.*
FROM (SELECT * FROM people ORDER BY last_name,first_name) As oldtable;

--Approach 3:  This will work I think even in MySQL :)
--Advantage: Cross Platform - yeh
--Disadvantage:  Potentially slow correlated subquery for large datasets
--  and gets somewhat messy the more orderings you tack on
-- relies on your order by producing unique results
SELECT (SELECT COUNT(*) FROM people
    WHERE
    (COALESCE(people.last_name,'') || COALESCE(people.first_name,'')) <=
    (COALESCE(oldtable.last_name,'')
    || COALESCE(oldtable.first_name,''))) As row_number,
    oldtable.*
FROM (SELECT *
    FROM people
    ORDER BY
    last_name, first_name) As oldtable;
```

Back to Table Of Contents Simulating Row Number in PostgreSQL Pre 8.4 Reader Comments

Quick Guide to writing PLPGSQL Functions: Part 1 ***Beginner***

In this series we'll go over writing PLPGSQL stored functions. We shall follow up in a later issue with a one page cheat sheet.

The Anatomy of a PLPGSQL FUNCTION

All PLPGSQL functions follow a structure that looks something like the below.

```
CREATE OR REPLACE FUNCTION fnsomefunc(numtimes integer, msg text)
   RETURNS text AS
$$
DECLARE
   strresult text;
BEGIN
   strresult := '';
   IF numtimes > 0 THEN
      FOR i IN 1 .. numtimes LOOP
         strresult := strresult || msg || E'\r\n';
      END LOOP;
   END IF;
   RETURN strresult;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE
SECURITY DEFINER
  COST 10;

--To call the function we do this and it returns ten hello there's with
carriage returns as a single text field.
SELECT fnsomefunc(10, 'Hello there');
```

The basic make-up of a PLPGSQL function is as follows:

1. There is the function interface that defines the args and the return type
2. There is the body which in modern versions of PostgreSQL (8+) the preferred encapsulation is dollar quoting vs. using a single quote
3. Within the body: There is a declaration of variables section which is optional
4. Then there is a BEGIN END structure that defines the meat of the function. Unlike sql functions which currently require you to refer to variables by their ordinal position $1, $2, $3 etc. in PLPGSQL you can refer to variables by there name.
5. After the body, like all PostgreSQL functions, is noted the Language and a tag that denotes how it should be cached. In this case we have noted IMMUTABLE meaning that the output of the function can be expected to be the same if the inputs are the same. Other options are STABLE - meaning it will not change within a query given same inputs and VOLATILE such as functions involving random () and CURRENT_TIMESTAMP that can be expected to change output even in the same query call.
6. PostgreSQL 8.3 introduced the ability to set costs and estimated rows returned for a function. For a scalar function the rows is not applicable so we leave that out for this simple example. The cost is relative to other functions and defaults to 100 unless you change it. Nuances of COST and caveats are outlined in our New Features for PostgreSQL Stored Functions
7. Note also the clause after the caching model is sometimes the words SECURITY DEFINER which means the function is run under the context of the owner of the function. This means the function can do anything the owner of the function has security to do even if the person running the function does not have those rights. This portion applies not just to PLPGSQL functions but any. If this clause is left out, then a function runs under the security context of the person running the function.

   For users coming from SQL Server - this is similar in concept to SQL Server 2005 - EXECUTE AS OWNER (leaving Security definer out is equivalent to EXECUTE As CALLER in sql server). Note SQL Server 2005 has an additional option called EXECUTE As 'user_name' which PostgreSQL lacks that allows you to run under a named user that need not be the owner of the function.

   For MySQL users, SECURITY DEFINER exists as well and works more or less the same as it does in PostgreSQL.
8. Pretty much all the functions you can write in PostgreSQL whether SQL, PLPGSQL or some other language can use recursion. We'll go over an example of that in another part of this series.

Conditional Logic

PLPGSQL has a couple of conditional logic structures. In the above we saw the simple IF THEN. There also exists IF .. ELSIF ..ELSIF END IF, IF ..ELSE ..END IF. We shall demonstrate by making dumb changes to our above.

```
CREATE OR REPLACE FUNCTION fnsomefunc(numtimes integer, msg text)
   RETURNS text AS
$$
DECLARE
   strresult text;
BEGIN
   strresult := '';
   IF numtimes = 42 THEN
      strresult := 'Right you are!';
   ELSIF numtimes > 0 AND numtimes < 100 THEN
      FOR i IN 1 .. numtimes LOOP
         strresult := strresult || msg || E'\r\n';
      END LOOP;
   ELSE
      strresult := 'You can not do that. Please don''t abuse our generosity.';
      IF numtimes <= 0 THEN
         strresult := strresult || ' You are a bozo.';
      ELSIF numtimes > 1000 THEN
         strresult := strresult || ' I do not know who you think you are.
               You are way out of control.';
      END IF;
   END IF;
   RETURN strresult;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

SELECT fnsomefunc(42, 'Hello there');
SELECT fnsomefunc(200, 'Hello there');
SELECT fnsomefunc(5000, 'Hello there');
```

Control Flow

In the above example we saw a variant of the FOR LOOP - below are a listing of the other basic control structures. In part 2 we shall delve into using some of these.

The basic control flow structures available in PLPGSQL are:

- FOR *somevariable* IN (1 ...someendnumber) LOOP .. END LOOP;
- FOR *somevariable* IN REVERSE someendnumber .. 1 BY *somestep* LOOP .. END LOOP;
- FOR *somevariable* IN (*somesqlquery*) LOOP ..RETURN NEXT; .. END LOOP;
- LOOP ..logic statements EXIT .. EXIT WHEN .. CONTINUE WHEN .. END LOOP;
- WHILE ... LOOP ... END LOOP;
- EXCEPTION WHEN .... WHEN ..
- Introduced in 8.3 RETURN QUERY which can be in any LOOP like structure or stand alone

Back to Table Of Contents Quick Guide to writing PLPGSQL Functions: Part 1 Reader Comments

Quick Guide to writing PLPGSQL Functions: Part 2 *Beginner*

In this second part of our PLPGSQL Quick Guide series, we shall delve more into control flow. As we mentioned in the previous part, the following control flow constructs exist for PLPGSQL.

- FOR *somevariable* IN (1 ...someendnumber) LOOP .. END LOOP;
- FOR *somevariable* IN REVERSE someendnumber .. 1 BY *somestep* LOOP .. END LOOP;
- FOR *somevariable* IN EXECUTE(*somesqlquery*) LOOP ..RETURN NEXT; .. END LOOP;
- LOOP ..logic statements EXIT .. EXIT WHEN .. CONTINUE WHEN .. END LOOP;
- WHILE ... LOOP ... END LOOP;
- EXCEPTION WHEN .... WHEN ..
- Introduced in 8.3 RETURN QUERY which can be in any LOOP like structure or stand alone. This is covered in New Features of PostgreSQL Functions

In this section we shall demonstrate looping thru sets of records and writing a set returning function. In the next section after, we shall delve a little into recursive functions, doing table updates, and raising notices.

The FOR somevariable IN somesqlquery

This in pre-8.3 was the most common construct for looping thru records. It is still the only way to return a set of records where the query statement is dynamically changing within the procedure. Below is an example of such a thing.

```
--Just returning a string
CREATE OR REPLACE FUNCTION somefuntext(param_numcount integer)
  RETURNS text AS
$$
DECLARE
    result text := '';
    searchsql text := '';
    var_match text := '';
BEGIN
    searchsql := 'SELECT n || '' down'' As countdown
            FROM generate_series(' || CAST(param_numcount As text) || ', 1, -1) As n ';


    FOR var_match IN EXECUTE(searchsql) LOOP
        IF result > '' THEN
            result := result || ';' || var_match;
        ELSE
            result := var_match;
        END IF;
    END LOOP;
    RETURN result;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

--To use you would do this --
SELECT somefuntext(10);


---RESULT---
10 down;9 down;8 down;7 down;6 down;5 down;4 down;3 down;2 down;1 down
```

```
--Returning a set of strings
CREATE OR REPLACE FUNCTION somefun_settext(param_numcount integer)
  RETURNS SETOF text AS
$$
DECLARE
    result text := '';
    searchsql text := '';
    var_match text := '';
BEGIN
    searchsql := 'SELECT n || '' down'' As countdown
```

```
            FROM generate_series(' || CAST(param_numcount As text) || ', 1, -1) As n ';


    FOR var_match IN EXECUTE(searchsql) LOOP
        RETURN NEXT var_match;
    END LOOP;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

--To use you would do this
SELECT n
FROM somefuntext(10) As n;
```

```
---RESULT---
    n
---------
 10 down
 9 down
 8 down
 7 down
 6 down
 5 down
 4 down
 3 down
 2 down
 1 down
(10 rows)
```

```
--Returning a set of anonymous records
CREATE OR REPLACE FUNCTION somefun_recordset(param_numcount integer)
  RETURNS SETOF record AS
$$
DECLARE
    result text := '';
    searchsql text := '';
    var_match record;
BEGIN
    searchsql := 'SELECT n || '' down'' As countdown, n as integer
            FROM generate_series(' || CAST(param_numcount As text) || ', 1, -1) As n ';


    FOR var_match IN EXECUTE(searchsql) LOOP
        RETURN NEXT var_match;
    END LOOP;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

To call the above, you would do something like below. Note when calling an alias record set, we need to define the structure of the output. The benefit of an alias recordset is that you can have all sorts of queries use the same function as long as your calling query knows the expected structure. The downside is that you have to note the expected structure which quickly becomes annoying if it is always the same.

```
SELECT r.n , r.countdown
FROM somefun_recordset(10)
    As r(countdown text, n integer)
    ORDER BY r.n;
```

```
---RESULT --
 n  | countdown
----+-----------
  1 | 1 down
  2 | 2 down
  3 | 3 down
  4 | 4 down
  5 | 5 down
  6 | 6 down
  7 | 7 down
  8 | 8 down
  9 | 9 down
 10 | 10 down
```

In order to not have to specify the output structure of your query, you need to either use a named type or use a table as a type. NOTE: All tables and views in PostgreSQL have a corresponding row type that is autocreated for them so if your output results match a table type, you can use that table record type as output. Below we demonstrate creating a type to hold each row and using it.

```
CREATE TYPE somefun_type As (countdown text, n integer, somethingrandom numeric);
CREATE OR REPLACE FUNCTION somefun_setoftype(param_numcount integer)
  RETURNS SETOF somefun_type AS
$$
DECLARE
   result text := '';
   searchsql text := '';
   var_match somefun_type;
BEGIN
   searchsql := 'SELECT n || '' down'' As countdown, n as integer, random() As numeric
            FROM generate_series(' || CAST(param_numcount As text) || ', 1, -1) As n ';


   FOR var_match IN EXECUTE(searchsql) LOOP
      RETURN NEXT var_match;
   END LOOP;
END;
$$
LANGUAGE 'plpgsql' VOLATILE;
```

```
--Example run --
SELECT myfun.n, myfun.somethingrandom
FROM somefun_setoftype(10) As myfun
WHERE myfun.n IN(2,3);
```

```
 n |  somethingrandom
---+-------------------
 3 |  0.78656436316669
 2 | 0.818326753564179
```

Back to Table Of Contents

Reader Comments

When it rains it Drizzles

Brian Aker
Hi!

Jim continues to work with the Falcon team and was at the developer meeting. Monty and his team continue to work on the Maria engine (which has been their project for a couple of years now). Monty does not work on Drizzle, Maria is enough to keep him busy.

I am employed by Sun and Drizzle is what you describe it as. We have had a number of people working on it part time for a while now, some inside of Sun and many outside of. In Jay's case he can best describe it himself, but I think I can sum it up by saying that he wanted to do more development, and there is certainly a lot of active work being done on drizzle.

Soap Opera is a pretty dull one :)

Cheers,
-Brian

Regina
Thanks Brian,

That does make me feel much better. So the rains drizzle analogy is in line then; Just a light rain. :).

Thanks,
Regina

anon
baseless FUD. Mysql has over 100 engineers. a few changes does not impact that much

pcdinh
I love PostgreSQL but I don't think your comments above make sense. MySQL code quality and its future direction are quite stable and mature but it needs evolution that leads to staff move. Drizzle is a very good thing to see on the web.

Sun is a very big corporation and it can offer enough engineers to move MySQL forward.

What I want to see is after the merge, if MySQL is as community friendly as usual. It should.

pcdinh

Regina
Good point. I guess many people felt I over-reacted and thought I was just making a case for PostgreSQL.

I apologize. I didn't quite mean it that way. As I have said before - we make a lot of money on MySQL consulting and our PostgreSQL business is growing but still behind MySQL. My concern was actually more as a MySQL user than a PostgreSQL advocate.

I guess the main point of my article, is not that MySQL is not stable, but that when key people - seem to sway in another direction - it leaves a bit of a hazy cloud as to what the direction of the product is. Do I treat Drizzle as a separate entity altogether or as a subset of MySQL. Either way it seems to divide the group, though it will probably increase use among people who were not happy with the current MySQL direction.

I guess my point is that its a little hazy right now and hard to tell what's going on.

True PostgreSQL has undergone many private forks, but the community doesn't seem that divided in a meaningful way. E.g. even though EnterpriseDb is a separate private product with its own objectives, I would say its very integrated with the PostgreSQL group and tries to keep in step with the rest of the PostgreSQL progression and not a complete fork like Drizzle is.

I would say Brian's comments were very helpful. If truth be known I like a lot of the MySQL core group and think they are really great people. The way news around the globe reads - its like they abandoned ship or something.

As to the FUD comment from the other guy - you can have a 1000 people on a project, but a 1000 people with no direction is worse than 2 people with a clear direction is my point. I just wasn't clear if those 100 engineers had a clear direction.

How to delete many functions

k_p_l
If you quote_ident(ns.nspname) and quote_ident(proname), you will handle functions with capital letters etc also.

Pythian Group Blog
Welcome to the 117th edition of Log Buffer, the weekly review of database blogs.
For those of you who don’t know me, My name is Nicklas Westerlund, and I’m a MySQL DBA with The Pythian Group. This is my first time writing Log Buffer, and I ...

Why is my index not being used

Tom Lane
You're right, a varchar_pattern_ops index won't work for IN (or more generally, plain "=") queries. This is because its equality operator is ~=~ ... which, in the original conception, didn't necessarily work like =. pattern_ops uses straight strcmp comparison while regular varchar comparison is based on strcoll, and in non-C locales strcoll can say that two strings are equal even if they aren't bitwise equal.

For the last release or so there's been a hard-wired requirement that varchar = be plain bitwise equality, which is what ~=~ does. So for 8.4 we have gotten rid of ~=~ and made regular = be the equality member of the varchar_pattern_ops opclass.

In short: as of 8.4 a pattern_ops index will be able to service plain "=" and hence IN queries, but in earlier releases you really do need two indexes if you're not using C locale.

David Fetter
You missed 0, the most common case, which is that the planner is smarter than the question, and using that index wouldn't have been the right thing anyhow.

Greg Smith
One of the common causes for using a sequential scan instead of an indexed one you didn't mention is that it will happen if the query is accessing a large portion of the table. Sometimes people wonder why the index isn't being used for a query that is accessing, say 30% of the rows. The reality is that if you're accessing that many, given how typical rows are clustered together you might as well avoid the overhead of using the index and just look at the whole thing.

With default parameters I believe that crossover point (where it's considered faster to just access the whole thing) is normally at 20% of the table--if the planner believes you're going to see more than that, it might as well just fetch the whole thing. Which is normally correct, but can be wrong particularly for very large tables where the data is packed tightly (a typical example is a large historical archive you're pulling a section out of). That sort of thing is where temporary changes to random_page_cost and enable_seqscan can be handy as pseudo-hints to the optimizer.

Simulating Row Number in PostgreSQL Pre 8.4

Richard Broersma Jr.
Couldn't the last example be refined using row wise comparison instead of field concatenation?

Regina
Richard,

You mean something like

ROW(t.last_name, t.first_name) ...

Hadn't thought about that. I guess that would work.

Quick Guide to writing PLPGSQL Functions: Part 1

Miguel Angel
Thanks for make a tutorial for this, i find really little information (apart from official postgresql documentation) for begin learning plpgsql without have used before any other database procedure language, i will wait for the next parts and the cheat sheet :).

Kuo, ChaoYi
Thanks to the teaching - I translated into Chinese in
http://postgresql-chinese.blogspot.com/2008/10/plpgsql-part-1.html

Regina
Thanks. I'll try to write the next part in next couple of days.

Fang
Link to "New Features for PostgreSQL Stored Functions" is dead...

Regina
Fang,

Thanks for the catch. Just fixed the link. Still working on the next part which should have sometime this week.
