Postgres OnLine Journal: November 2008 / December 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/

PostGIS 1.3.4 is finally out the door

PostGIS 1.3.4 is finally out the door. This version has:

1. Support for 7.3, 7.4, 8.0, 8.1, 8.2, 8.3, 8.4 beta
2. GEOS 2.2.3, GEOS 3.0.3, GEOS 3.1 beta
3. Numerous bug fixes and speed improvements
4. Slightly better documentation
5. addition of function comments to help guide new users while in the psql or pgAdmin environment
6. One new function ST_AsGeoJSON to support javascript apis such as OpenLayers.

Some advanced GEOS functions such as ST_SimplifyPreserveTopology and ST_CoveredBy (both released in 1.3.3) will not be installed unless you are running GEOS 3.0 or above.

GEOS 3.0.3 also has numerous bug fixes such as a contains bug fix, compilation fixes to allow compiling on higher gcc. With that said, its highly recommended to upgrade to 3.0.3 as well.

Which minimal PostgreSQL version to support?

I suppose all side and core projects of PostgreSQL have to deal with this problem. Supporting numerous platforms and numerous versions of PostgreSQL is no easy undertaking and of course takes away resources that can be focused on bugs and new enhancements.

There is some talk recently in the PostGIS development team as to what minimal to support. I'm all for dropping 7.4.whatever and 8.0 for past releases (bug fix releases) and new major releases only supporting 8.2+ given our limited resources.

I'm curious what other PostgreSQL projects do and if PostgreSQL and other open source developers think this is wise.

How many PostGIS users out there are running something lower than 8.2. My general thinking is that there aren't that many and spending key resources on such a small crowd is just not worthwhile.

Back to Table Of Contents

PostGIS 1.3.5 out the door critical patch to 1.3.4 and Testing Enhancements

PostGIS 1.3.5 urgent upgrade if you are running 1.3.4

We were forced to release a 1.3.5 PostGIS upgrade as a result of a bug we accidentally introduced in 1.3.4 during our code cleanup. We apologize for any inconvenience this may have caused people. This bug affects the use of MULTILINESTRINGS and rears its ugly head by giving errors such as invalid circular line string when calling ST_Multi or another odd error when doing a Force collection on a MULTILINESTRING. This hits mapserver users using these geometry types the hardest.

More details of the issue can be gleaned from Paul Ramsey's blog. Warning: PostGIS 1.3.4 + Mapserver

Documentation-driven testing and how I learned to love XML and XSLT

A while back Kevin Neufeld came up with an efficient template for documenting each PostGIS function. This is the template we are using to rewrite the PostGIS 1.4 function reference section. The PostGIS 1.4 function reference section is almost complete. This allowed us to create special indexes sections and generate database level comments on functions with an XSL Transform.

This documentation enhancement helped at least partly solve another problem all thru the magic of XSLT. Given how much code we are refactoring and so, forth, how can we possibly test all 150 someodd spatial functions against all geometry types we support and continue to do so as our geometry types expand and functions expand?

It turned out not to be too hard to write an xsl transform script that takes our documentation as input and spits out what I shall call a **torture script** that for each function defined in our documentation exercises it for each geometry type we support. So the output is an approximate 1 MB file of sql battery of tests. The added benefits is that as we add new functions to our documentation, they are automatically tested and the script also flags when documented functions are no longer in use or errors in the documentation. So its a very interesting feedback loop.

It is still a work in progress, but was unfortunately too late in coming to test 1.3.4. Testing against 1.3.5 we uncovered some crashers and minor defects that have existed in this and prior versions which we plan to attend to in 1.3.6 and 1.4. We also uncovered another small thing we sort of broke with Curved types, but was of questionable logic to begin with that we temporarily patched in 1.3.5 but will revisit in later versions. Hopefully this added step to our testing process will save us from similar mistakes in the future.

Back to Table Of Contents PostGIS 1.3.5 out the door critical patch to 1.3.4 and Testing Enhancements Reader Comments

PostGIS 1.3.4 almost out the Door and 8.4 highlights

PostGIS 1.3.4 is almost out the door - Need testers

Well PostGIS 1.3.4 is almost out the door and we will be releasing an RC3 very shortly. As a developer in the group and also as a user of the product that is near and dear to me I would be really appreciative if people in the PostgreSQL community interested in PostGIS can test this out. Below is a clip of Mark Cave-Ayland's note to the postgis-dev group.

```
PostgreSQL 8.1+ - Win32
PostgreSQL 8.1+ - Solaris
PostgreSQL 7.3  - Linux

PostgreSQL 8.1+ - GEOS 2.2.3 - Linux
PostgreSQL 8.1+ - GEOS 3.0.3 - Linux
PostgreSQL 8.1+ - GEOS 3.1 - Linux

The plan here is to check Win32/Solaris both pass the new sed tests in
Makefile.config.in, make sure we haven't broken really old PostgreSQL
builds, and also make sure that GEOS 2.2, GEOS 3.0 and the very new GEOS
3.1 with prepared geometry doesn't break either.
```

We are especially looking for 7.3+ testers! Gasp and Sun Solaris 64-bit testers and 8.4 testers of any OS persuasion. I know Mark is a man of integrity and he promised this version would work even on 7.3 so I guess we have to stick to that. I don't think anyone personally cares about it working on that version but him, but we have to humor him :).

A couple of notes here of what has changed:

1. This now should compile with 8.4 dev
2. Lots of memory leak bugs closed to make upcoming GEOS 3.1 work well with PostGIS and major speed enhancements if you are using GEOS 3.1
3. AsGeoJSON and minor changes to other output function
4. Numerous speed improvements for ST_Intersects, ST_DWithin, ST_Contains,Within, etc
5. Numerous bug fixes to circular string support and increase in number of functions it works with.

Downloads of the source are at: http://postgis.refractions.net/download/ Choose the 1.3 snapshot or 1.3.4RC3 when we release it.

8.4 on the Way

8.4 seems to be moving along. Hubert Lubaczewski has some nice examples of what you can expect and boy is he on a roll. Below are some and I'm sure I missed others.

- sql-wrappable RETURNING which allows SQL functions to return list of records being added or deleted
- Suppressing redundant updates
- Database Level collation
- Common Table Expressions
- RETURNS TABLE

Back to Table Of Contents

Backing up Login Roles aka Users and Group Roles ***Beginner***

Sometimes when you are testing or setting up a server or just porting things to another server, you just want to install the same set of users as you had before without restoring any databases. In PostgreSQL, the users (Login Roles) and group roles are stored at the server level and only the permissions to objects are stored at the database level.

Question:

How do you restore just the users and roles without having to do a full pg_dumpall of your server?

Answer

Using the Command Line:

Remember that these executables pg_dumpall, psql are located in the bin of your postgresql install

Pre 8.3 syntax

```
pg_dumpall -h localhost -p 5432 -U postgres -v --globals-only > /path/to/useraccts.sql
```

8.3 Syntax

8.3 introduced the -f option to denote the file name which makes things a bit more predictable how they behave from OS to OS.

```
pg_dumpall -h localhost -p 5432 -U postgres -v -f "/path/to/useraccts.sql" --globals-only
```

Then to restore the accounts on the new server, open up the .sql file generated and delete all the accounts and stuff you don't want to bring over. Then just run the generate .sql file with psql something like

```
psql -h localhost -d postgres -U postgres -f "/path/to/useraccts.sql"
```

Using PgAdmin III

1. Connect to the Server
2. At Server level, right click the Server and choose **Backup Globals**
3. Browse to path you want and give a .sql extention

1. To restore - connect to Postgres db (or any db as a super user)
2. simply open the .sql file in the Query window
3. Delete all the stuff at the top like \connect etc
4. Run the script

Back to Table Of Contents Backing up Login Roles aka Users and Group Roles Reader Comments

Yum addendum for 8.3.5 and PgAgent *Beginner*

We had the pleasure of doing a fresh install of PostgreSQL 8.3.5 on RedHat EL4 box and when using the Yum repository, we noticed a couple of changes from last time we did this. This could have been an oversight in our documentation before.

Changes to Yum Install for 8.3.5?

In our April 2008 issue we had An Almost Idiot's Guide to PostgreSQL YUM and that article still seems to be surprisingly popular.

In the first step we had:

```
yum install postgresql
```

and that as I recall installed the postgresql server in addition to some client libraries.

For 8.3.5 fresh install it seems they are separated and to get the postgresql server you need to do:

```
yum install postgresql
yum install postgresql-server
```

We also throw in to get the development headers:

```
yum install postgresql-devel
```

PgAgent is now separate from PgAdmin III for Linux

We also described PgAgent in our January 2008/February 2008 issue, which is a task scheduling agent for PostgreSQL, which is similar in concept to Microsoft SQL Server Agent, but cross-platform so should fill Microsofties with a warm and fuzzy feeling.

So to get PgAgent

1. Download from http://www.postgresql.org/ftp/pgadmin3/release/pgagent/
2. Extract - Copy bin/pgagent file to /usr/bin/
3. Make sure to mark as executable with
   ```
   chmod 775 /usr/bin/pgagent
   ```
4. Run
   ```
   psql -U postgres -d postgres -f pgagent.sql
   ```
5. Install PgAgent as a service. See next section.

PgAgent on CentOS/Red Hat EL as a service

Below is a sample script we hacked together from some samples we have seen. Works for us at anyrate, but we are predominantly microsofties though less so than we were before, so more professional Linux users, feel free to butt in.

```
#!/bin/bash
#
#   /etc/rc.d/init.d/pgagent
#
# Starts the pgagent daemon
#
# chkconfig: - 65 35
# description: PgAgent Postgresql Job Service
# processname: pgagent
# Source function library.
. /etc/init.d/functions


RETVAL=0
prog="PgAgent"

start() {
   echo -n $"Starting $prog: "
   daemon "pgagent hostaddr=127.0.0.1 dbname=postgres user=postgres"
```

```
    RETVAL=$?
    echo
}

stop() {
    echo -n $"Stopping $prog: "
    killproc /usr/bin/pgagent
    RETVAL=$?
    echo
}

#
#   See how we were called.
#
case "$1" in
  start)
    start
    ;;
  stop)
    stop
    ;;
  reload|restart)
    stop
    start
    RETVAL=$?
    ;;
  status)
    status /usr/bin/pgagent
    RETVAL=$?
    ;;
  *)
    echo $"Usage: $0 {start|stop|restart|reload|status}"
    exit 1
esac

exit $RETVAL
```

1. Then copy the above script to a text file called **pgagent** -- no extension
2. Upload to folder on server */etc/rc.d/init.d*. The path may vary slightly
3. On server do a
   `chmod 755 /etc/rc.d/init.d/pgagent`
   to make it executable.
4. to Add to start up services:
   `chkconfig pgagent on`
5. To test you can do a
   `service pgagent start`

It should automatically start on its own during server boot ups.

Back to Table Of Contents Yum addendum for 8.3.5 and PgAgent Reader Comments

Quick Guide to writing PLPGSQL Functions: Part 3 - NOTICES, RECURSION, and more *Intermediate*

In this third part of our PLPGSQL Quick Guide series, we shall delve into writing recursive functions. Before we do that, we shall demonstrate a very important but trivial feature in PostgreSQL and that is the RAISE NOTICE feature. There are more elegant ways of debugging, but this is the simple brain dead way of doing so.

RAISE

RAISE Notices in plpgsql are generally used for two reasons:

- As a simple debugging tool to output state variables in a function call.
- As a WARNING to a user to inform them of important things such as this function is deprecated and should not be used or they are using something in an incorrect way.

A simple example of notices and recursion is shown below. Admittedly I couldn't come up with a more pointless example to demonstrate recursion:

```
CREATE OR REPLACE FUNCTION fnsomefunnote(param_numcount integer)
  RETURNS integer AS
$$
DECLARE
BEGIN
   IF param_numcount > 0 THEN
      RAISE NOTICE 'Yo there I''m number %, next: %', param_numcount, param_numcount -1;
      RETURN fnsomefunnote(param_numcount - 1);
   ELSE
      RETURN param_numcount;
   END IF;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

```
SELECT fnsomefunnote(4);

Returns 0 and also notices - if you are looking at this in PgAdminIII
you shall see this on the messages tab

NOTICE:  Yo there I'm number 4, next: 3
NOTICE:  Yo there I'm number 3, next: 2
CONTEXT:  PL/pgSQL function "fnsomefunnote" line 5 at RETURN
NOTICE:  Yo there I'm number 2, next: 1
CONTEXT:  PL/pgSQL function "fnsomefunnote" line 5 at RETURN
PL/pgSQL function "somefunnote" line 5 at RETURN
NOTICE:  Yo there I'm number 1, next: 0
CONTEXT:  PL/pgSQL function "fnsomefunnote" line 5 at RETURN
PL/pgSQL function "fnsomefunnote" line 5 at RETURN
```

RAISE also has other variants namely **DEBUG(1-5)**, **LOG**, **INFO**, **EXCEPTION**

DEBUG, LOG, and INFO are just different levels of NOTICE and only vary depending on which logs they get written to and if they get written to client. By default NOTICE is always written to the client. These are controlled by the postgresql.conf *client_min_messages* and *log_min_messages*.

RAISE EXCEPTION is slightly different. It both displays an error message and halts further excecution of the stored function.

Below is a slightly different variant of the above: Also note here we follow the general practice of having a single point of return. Having a single point of return tends to make your code easier to read and debug.

```
CREATE OR REPLACE FUNCTION fnsomemorefunnote(param_numcount integer)
  RETURNS integer AS
```

```
$$
DECLARE result integer;
BEGIN
   IF param_numcount < 0 THEN
      RAISE EXCEPTION 'Negative numbers are not allowed';
   ELSIF param_numcount > 0 THEN
      RAISE NOTICE 'Yo there I''m number %, next: %', param_numcount, param_numcount -1;
      result := fnsomemorefunnote(param_numcount - 1);
   ELSE
      RAISE INFO 'Alas we are at the end of our journey';
      result := param_numcount;
   END IF;
   RETURN result;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;


--------
SELECT fnsomemorefunnote(n)
FROM generate_series(-1,2) As n;

ERROR:  Negative numbers are not allowed

********** Error **********

ERROR: Negative numbers are not allowed
SQL state: P0001
---------

--no result is returned--


SELECT fnsomemorefunnote(n)
FROM generate_series(0,2) As n;
--result--
0
0
0


--messages--
INFO:  Alas we are at the end of our journey
NOTICE:  Yo there I'm number 1, next: 0
INFO:  Alas we are at the end of our journey
CONTEXT:  PL/pgSQL function "fnsomemorefunnote" line 7 at assignment
NOTICE:  Yo there I'm number 2, next: 1
NOTICE:  Yo there I'm number 1, next: 0
CONTEXT:  PL/pgSQL function "fnsomemorefunnote" line 7 at assignment
INFO:  Alas we are at the end of our journey
CONTEXT:  PL/pgSQL function "fnsomemorefunnote" line 7 at assignment
PL/pgSQL function "fnsomemorefunnote" line 7 at assignment

Total query runtime: 16 ms.
3 rows retrieved.
```

Storing values in Temp variables, FOUND and more recursion

Sometimes you have the need to store intermediate results in a temp variable from a query to use for later processing. Here is a simple example of that.

FOUND is a state variable that contains either true or false if the last result returning query returned records or not.

Below is an example that demonstrates, FOUND, storing sql values in temp variables, and more recursion.

This example is the plpgsql equivalent to our SQL Server Transact-SQL article called Using SQL Server 2000's User Defined Function to solve the Tree Problem

Also check out Using PostGreSQL User-Defined Functions to solve the Tree Problem

```
CREATE TABLE employees(employeeid varchar(50) PRIMARY KEY, managerid varchar(50));
--8.2+ syntax for 8.1 and below need to do individual insert statements
INSERT INTO employees(employeeid, managerid)
```

```
VALUES ('Diana', Null), ('Peter', 'Diana'),
   ('Nancy', 'Peter'), ('John', 'Nancy');

CREATE TYPE reportsTo AS
  (employeeid varchar(50),
   depth integer);


--8.3 syntax
CREATE OR REPLACE FUNCTION fn_reportsTo (param_employeeid varchar(50), param_depth integer)
RETURNS SETOF reportsTo
 AS
 $$
 DECLARE
  var_managerid varchar(50);
  var_next_depth integer;
 BEGIN
   var_next_depth := param_depth + 1;
   SELECT managerid INTO var_managerid FROM employees WHERE employeeid = param_employeeid;
   IF FOUND AND var_managerid IS NOT NULL AND var_managerid > ''
    AND param_depth < 6 AND var_managerid <> param_employeeid THEN
/***We stop if a person is their own manager or  a person has no manager or
We've  exceeded a depth of 5 (to prevent potential infinite recursion ***/
      RETURN QUERY
         SELECT employeeid, param_depth
            FROM employees M
            WHERE M.employeeid = param_employeeid
         UNION ALL
            SELECT employeeid, depth FROM fn_reportsTo(var_managerid, var_next_depth);
   ELSE
      RETURN QUERY SELECT param_employeeid As employeeid, param_depth As depth;
   END IF;
 END;
$$
LANGUAGE 'plpgsql' STABLE;


SELECT *
   FROM fn_reportsTo('John');


 employeeid | depth
------------+-------
 John       |     0
 Nancy      |     1
 Peter      |     2
 Diana      |     3
```

Overloading Function names

PostgreSQL unlike Microsoft SQL Server, supports function name and argument overloading. Which in a nutshell means you can have multiple functions with the same name as long as they take a different number of arguments and/or different argument types. The functions don't even need to be written in the same language. With that said we can make our function a bit nicer by doing this.

```
CREATE OR REPLACE FUNCTION fn_reportsTo(param_employeeid varchar(50))
RETURNS SETOF reportsTo
AS
$$
   SELECT * FROM fn_reportsTo($1, 0);
$$
LANGUAGE 'sql' STABLE;
```

By doing the above we kill 2 birds with one stone:

1. We no longer have to pass in a goofy 0 for depth.
2. We can now use a unique feature of functions written in SQL and C that allows set returning functions in those languages to be used in the SELECT clause as described in our Trojan SQL Function Hack - A PL Lemma in Disguise

   Which allows us to do this:

```
SELECT e.employeeid As emp, (fn_reportsTo(employeeid)).*
  FROM employees E
  ORDER BY e.employeeid, depth;

 emp   | employeeid | depth
-------+------------+-------
 Diana | Diana      |     0
 John  | John       |     0
 John  | Nancy      |     1
 John  | Peter      |     2
 John  | Diana      |     3
 Nancy | Nancy      |     0
 Nancy | Peter      |     1
 Nancy | Diana      |     2
 Peter | Peter      |     0
 Peter | Diana      |     1
```

What is in store in 8.4

8.4 has a couple of things cooking to advance SQL in PostgreSQL and improved features for PL/PGSQL. Below is just a sampling.

1. David Fetters Trees and More http://fetter.org/Trees_and_More_WEST_2008.pdf - which demonstrates how to write recursive SQL statements with upcoming Common Table Expressions in 8.4
2. Hubert's Waiting for 8.4 - pl/* srf functions in selects http://www.depesz.com/index.php/2008/11/03/waiting-for-84-pl-srf-functions-in-selects/ - this one is a pretty important one I think. It means no longer a need for a trojan like hack for 8.4+
3. Hubert's Waiting for 8.4 - RETURNS TABLE http://www.depesz.com/index.php/2008/08/04/waiting-for-84-returns-table/ Again this feature I consider to be pretty important. Especially for people coming from Microsoft SQL Server backgrounds from a comfort stand-point (look at how similar SQL Servers table returning functions look). It also means no longer needing to create a custom type as we did above if you want to return a custom set of fields.

Back to Table Of Contents

Fusion Charts and PostgreSQL Part 1: Database Analysis of USDA DB ***Beginner***

In our Product Showcase section of this issue, we introduced Fusion Charts which is a flash-based charting product that makes beautiful flash charts. It comes in both a free and a non-free more bells and whistles version.

In this 3-part series article we shall demonstrate using this with a PostgreSQL database, building a simple dashboard with ASP.NET and PHP. We shall demonstrate both C# and VB.NET both using the PostgreSQL NPGSQL driver.

For this first part we shall simply load the database, do a quick analysis of what we've got to report on and create some views to help us with our PHP and ASP.NET apps that will follow in parts 2 and 3.

We will be testing this on 8.3, but since the database is an old one, it should work just fine on older versions of PostgreSQL. We'll try to refrain from using new features of PostgreSQL.

Loading the USDA Database

For the database, we shall be using the **usda** (USDA The US Department of Agriculture's public domain food and nutrient database) which you can download from http://pgfoundry.org/projects/dbsamples/.

Now for the load

1. Extract the tar.gz file.
2. Launch your shell - if you are on windows you'll want to use the Start->Program Files->PostgreSQL *someversion*->Command Prompt
3. ```
   psql -h localhost -p 5432 -U postgres -c "CREATE DATABASE usda WITH ENCODING='UTF8';"
   psql -h localhost -p 5432 -f /path/to/usda.sql -U postgres -d usda
   psql -h localhost -p 5432 -U postgres -c "CREATE ROLE usda_app LOGIN PASSWORD 'foodie' NOSUPERUSER
   INHERIT NOCREATEDB NOCREATEROLE;
   ```

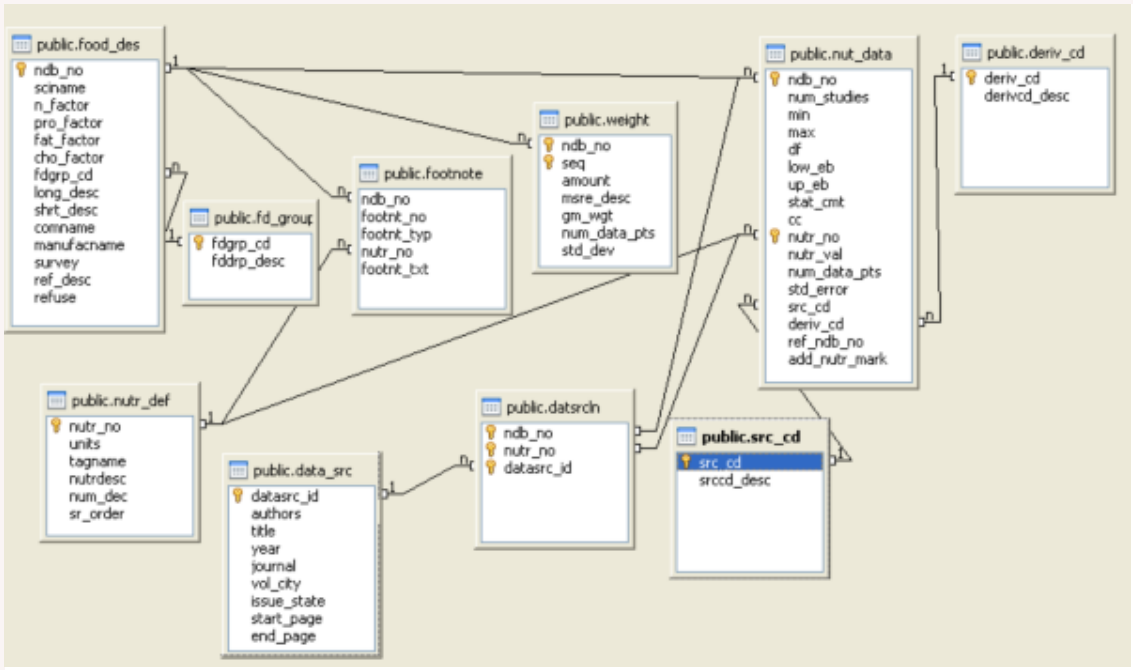
Now this database turns out to be somewhat rich for a demo db. It helps to look at the relationships in a relational designer to get a breath of what metrics we can glean from here. So lets itemize what we have here.

We covered installing Open Office's PostgreSQL SDBC driver in Using OpenOffice Base 2.3.1 with PostgreSQL

Installing the driver is the same in OpenOffice 3.0

Analyzing what we have

Below is a snapshot of what our loaded database relationships look like in OpenOffice 3.0
   ```



Now lets see what we have here and what interesting questions we can answer and graph from this data. Since we have no instructions to go by aside from the relationships and the data we see, we'll make some educated guesses.

Tables

- **data_src** -- appears to be a list of some food research articles.
- **datsrcln** -- correlates articles to nutritional data (nut_data) as we can see from relationships.
- **deriv_cd** -- some sort of lookup table to denoting short names - that define if nutritional is derived, calculated or analytical.
- **fd_group** -- lookup table defining food groups
- **food_desc** -- catalog of various foods
- **foot_note** -- general notes about a food stats. Only has one record so can't be that interesting.
- **nut_data** -- various stats collected about each food.
- **nutr_def** -- lookup table defining the various stat types about food. Things like protein, lactose, etc. concentration
- **src_cd** -- superclass of source of information - is it analytical, derived etc.
- **weigth** -- different unit types and gram weight of 1 unit of each for each food.

What do we want to know

Looking at the data, some interesting questions come to mind. We may expand on this as we get into the app. Anyone with other thoughts are free to suggest.

- What are the top 10 foods that provide the most protein, caffein, sugar, etc, pick your favorite metric?
- Which food groups have the highest average concentration of protein, caffeine, sugar, pick your favorite metric?

Creating Views and granting rights

For this section note we are living out with metrics like 18:0, 14:0. I have no clue what those things are.

Create a view to provide basic food stats by metric

```
CREATE OR REPLACE VIEW vwfdgrpstats AS
 SELECT g.fddrp_desc AS item_name, g.fddrp_desc AS food_group,
 nutr_def.nutrdesc AS metric, round(avg(nut_data.nutr_val)::numeric, 3) AS avgnutritionalvalue,
 nutr_def.units
   FROM food_des f
   JOIN fd_group g ON f.fdgrp_cd = g.fdgrp_cd
   JOIN nut_data ON nut_data.ndb_no = f.ndb_no
   JOIN nutr_def ON nutr_def.nutr_no = nut_data.nutr_no
   WHERE nutr_def.nutrdesc NOT LIKE '%:%'
  GROUP BY g.fddrp_desc, nutr_def.nutrdesc, nutr_def.units;

GRANT SELECT ON TABLE vwfdgrpstats TO usda_app;
```

Create view to get average stats about food groups by unit and metric

```
CREATE OR REPLACE VIEW vwfoodstats AS
 SELECT initcap(f.shrt_desc) AS item_name, f.shrt_desc AS food_name,
 g.fddrp_desc AS food_group, nutr_def.nutrdesc AS metric,
 round(avg(nut_data.nutr_val::numeric), 3) AS avgnutritionalvalue,
 nutr_def.units
   FROM food_des f
   JOIN nut_data ON nut_data.ndb_no = f.ndb_no
   JOIN nutr_def ON nutr_def.nutr_no = nut_data.nutr_no
   JOIN fd_group g ON f.fdgrp_cd = g.fdgrp_cd
     WHERE nutr_def.nutrdesc NOT LIKE '%:%'
  GROUP BY f.shrt_desc, g.fddrp_desc, nutr_def.nutrdesc, nutr_def.units;
```

Back to Table Of Contents Fusion Charts and PostgreSQL Part 1: Database Analysis of USDA DB Reader Comments

Fusion Charts for Sprucing up Data

This product is not specifically a PostgreSQL product but it is one that we use frequently with many of our database apps so we felt our obligation to blog about it in the context of databases. Lets face it, when you have a database, somebody will come to you one day and demand to see their data in sparkling colors, because why have data if you can't see it in sparkling colors. They might not know what the data is telling them, but at least it will look damn good when charted in 3D.

This is when you should whip out something like Fusion Charts. This is just a small part of a three part series. In our application arena, we shall demonstrate using Fusion Charts in PHP as well as ASP.NET and of course display PostgreSQL data using it. We shall only be demonstrating the Free version. If you really insist on Oracle, MySQL, IBM DB11, SQL Server, SQLite or FireBird or some other flavor of db, you can perform the same trick with slight variation. You just need data you want to chart.

What is Fusion Charts?

Fusion Charts is a Macromedia Flash-based charting component that makes extremely attractive 2D and 3D charts. It comes in 2 flavors.

- Fusion Chart v3 Fusion Charts v3 which comes with about 45 different chart types to choose from and numerous other extra bells and whistles for animation and integration.
- Fusion Charts v2 Free - The free version is free for commercial use as well, but has only 22 chart types and limited animation features. This version actually satisfies most of our charting needs.

The differences between the Free and Non-free versions are outlined in Free vs. v3

Surprisingly, even the Free version contains a Gantt Chart which is a real rarity among free charting software.

Cool Features

What we really love about this product

- The Charts are clean and breathtakingly beautiful
- The XML data format it requires is relatively easy to produce
- Both the Free and Commercial come packaged with APIs for ASP.NET, PHP, ASP

Some simple demonstrations can be seen here.

Stay tuned for our Application demonstrations of how pretty data can look when dressed up in charts.

Back to Table Of Contents Fusion Charts for Sprucing up Data Reader Comments

PostgreSQL 8.3 PLPGSQL Cheatsheet Overview

To finish off our PL/PGSQL tutorial series, we are providing a PL/PGSQL cheat sheet.

Below is a Thumbnail view of the PostgreSQL 8.3 PL/PGSQL cheat sheet that covers both 8.3 new features and past core PL/PGSQL features. This is by no means comprehensive, but are the features we use most often.

PDF Portrait version 8.5 x 11" of this cheatsheet is available at PostgreSQL 8.3 PL/PGSQL Cheatsheet 8.5 x 11 and also available in PDF A4 format and HTML.

Back to Table Of Contents

Reader Comments

PostGIS 1.3.5 out the door critical patch to 1.3.4 and Testing Enhancements

Marc

The new documentation is wonderfull and the way you all adressed the issues with 1.3.4 was great, too.
Kudos to the postgis team.

Backing up Login Roles aka Users and Group Roles

Gurjeet Singh

--globals-only option also dumps the non-default tablespaces, if you have any... So those need to be removed too, from the final file.

Yum addendum for 8.3.5 and PgAgent

Devrim GÜNDÜZ

Hmm. I think I can steal this script to add pgagent to my yum repository ;)

Fusion Charts and PostgreSQL Part 1: Database Analysis of USDA DB

Chris Swingley

> For this section note we are living out with metrics like 18:0, 14:0.
> I have no clue what those things are.

These are fatty acids. For example, 18:3 (ALA), 20:5 (EPA) and 22:6 (DHA) are three of the more important Omega-3 fatty acids (see the Wikipedia article: http://en.wikipedia.org/wiki/Omega-3_fatty_acids). One useful thing to do might be to create a new nutrient definition for Omega-3 and Omega-6 fatty acids, and then create new rows in the nut_data table by summing the relevant individual fatty acids.

Cheers

Chris

Fusion Charts for Sprucing up Data

digicon

We just pulled the trigger on purchasing FusionCharts for a project and love its breadth of charting, widgets, and maps. Good stuff.

Official PostgreSQL 8.3 PL/pgSQL Documentation URL: http://www.postgresql.org/docs/8.3/static/plpgsql.html

We cover only a subset of what we feel are the most useful constructs that we could squash in a single cheatsheet page

commonly used

[1] New in this release.

FUNCTION CACHING

```
IMMUTABLE
STABLE
VOLATILE
```

CONTROL FLOW

```
FOR i IN 1 ... numtimes LOOP
 statements
END LOOP;

FOR i IN REVERSE numtimes ... 1 LOOP
 statements
END LOOP;

FOR var_e IN EXECUTE('somedynamicsql') LOOP
 statements
 RETURN NEXT var_e;
END LOOP;

FOR var_e IN somesql LOOP
 statements
 RETURN NEXT var_e;
END LOOP;

IF condition THEN
  :
END IF;

IF condition THEN
  :
ELSE
  :
END IF;

IF condition. THEN
  :
ELSIF condition THEN
  :
ELSE
  :
END IF;

WHILE condition LOOP
  :
END LOOP;

LOOP
 -- some computations
 EXIT WHEN count > 100;
 CONTINUE WHEN count < 50;
 -- some computations for count IN [50 .. 100]
END LOOP;
```

RETURN constructs

```
RETURN somevariable
RETURN NEXT rowvariable
RETURN QUERY [1]
```

RAISE FAMILY

```
RAISE DEBUG[1-5]
RAISE EXCEPTION
RAISE INFO
RAISE LOG
RAISE NOTICE
```

EXCEPTION Handling

```
RAISE EXCEPTION 'Exception notice: %', var
EXCEPTION
 WHEN condition THEN
        do something or
        leave blank to ignore
END;
```

COMMON States and ERROR constants

```
FOUND
ROW_COUNT
division_by_zero
no_data_found
too_many_rows
unique_violation
```

Variable Setting

```
DECLARE
 somevar sometype :=  somevalue;
 somevar sometype
 curs1 refcursor;
 curs2 CURSOR FOR SELECT * FROM sometable;

somevar := somevalue

SELECT field1, field2
INTO somevar1, somevar2
FROM sometable WHERE .. LIMIT 1;
```

Return types

```
RETURNS somedatatype
RETURNS SETOF somedatatype
RETURNS refcursor
RETURNS trigger
RETURNS void
```

QUALIFIERS

```
SECURITY DEFINER
STRICT

COST cost_metric [1]
ROWS est_num_rows [1]
```

PLPGSQL FUNCTION SAMPLES

```
CREATE OR REPLACE FUNCTION fn_test(param_arg1 integer, param_arg2 text)
  RETURNS text AS
$$
DECLARE
        var_a integer := 0;
        var_b text := 'test test test';
BEGIN
        RAISE NOTICE 'Pointless example to demonstrate a point';
        RETURN var_b || ' - ' ||
                CAST(param_arg1 As text) || ' - '
                || param_arg2;
END
$$
  LANGUAGE 'plpgsql' STABLE;

SELECT fn_test(10, 'test');
```

--Example to RETURN QUERY --

```
CREATE OR REPLACE FUNCTION fnpgsql_get_peoplebylname_key(param_lname text)
  RETURNS SETOF int AS
$$
BEGIN
    RETURN QUERY SELECT name_key
        FROM people WHERE last_name LIKE param_lname;
END
$$
  LANGUAGE 'plpgsql' STABLE;
```

--Example using dynamic query --

```
CREATE OR REPLACE FUNCTION cp_addtextfield(param_schema_name text, param_table_name text,
        param_column_name text)
  RETURNS text AS
  $$
  BEGIN
        EXECUTE 'ALTER TABLE ' ||
                quote_ident(param_schema_name) || '.' || quote_ident(param_table_name)
                || ' ADD COLUMN ' || quote_ident(param_column_name) || ' text ';
        RETURN 'done';
  END;
$$
  LANGUAGE 'plpgsql' VOLATILE;
  SELECT cp_addtextfield('public', 'employees', 'resume');
```

--Perform action --

```
 CREATE OR REPLACE FUNCTION cp_updatesometable(param_id bigint,
        param_lname varchar(50), param_fname varchar(50))
  RETURNS void AS
$$
BEGIN
   UPDATE people SET first_name = param_fname, last_name = param_lname
         WHERE name_key = param_id;
END;
$$
  LANGUAGE 'plpgsql' VOLATILE SECURITY DEFINER;
```

--Sample logging trigger taken from docs

```
 CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);
 CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $$
    BEGIN
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), current_user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), current_user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), current_user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```