Pl/pgsql

02 June 2015

14:04

Writing Functions

As with most databases, you can string a series of SQL statements together and treat them as a unit.

Different databases ascribe different names for this unit—stored procedures, modules, macros, prepared statements, and so on.

PostgreSQL calls them functions.

Aside from simply unifying various SQL statements, these units often add the capability to control the execution of the SQL statements through using procedural language (PL).

In PostgreSQL, you have your choice of languages when it comes to writing functions.

Often packaged along with binary installers are SQL, C, PL/pgSQL,PL/Perl, PL/Python.

Since version 9.2, you’ll also find plv8js, which will allow you to write procedural functions in JavaScript.

plv8js should be an exciting addition to web developers and a nice companion to the built-in JSON type.

You can always install additional languages such as PL/R, PL/Java, PL/sh, and even experimental ones geared for high-end processing and AI, such as PL/Scheme or PgOpenCL.

A list of available languages can be found here: Procedural [Languages](http://www.postgresql.org/docs/current/interactive/external-pl.html)

Anatomy of PostgreSQL Functions

Function Basics

Regardless which language you choose to write a particular function, they all share a similar structure.

Example: Basic Function Structure

CREATE OR REPLACE FUNCTION func\_name(arg1 arg1\_datatype)

RETURNS some\_type | setof sometype | TABLE (..) AS

$$

BODY of function

$$

LANGUAGE language\_of\_function

Functional definitions can include additional qualifiers to optimize execution and to enforce security.

• **LANGUAGE** has to be one you have installed in your database. You can get a list with the query: SELECT lanname FROM pg\_language;

• **VOLATILITY** defaults to VOLATILE if not specified. It can be set to STABLE, VOLATILE, or IMMUTABLE. This setting gives the planner an idea if the results of a function can be cached. STABLE means that the function will return the same value for the same inputs within the same query. VOLATILE means the function may return something different with each call, even with same inputs.

Functions that change data or are a function of other environment settings like time, should be marked as VOLATILE.

**IMMUTABLE** means given the same inputs the function is guaranteed to return the same result. The immutable setting is merely a hint to the query planner. It may choose to not cache if it concludes caching is less cost effective than recomputation.

However, if you mark a function as VOLATILE it will always recompute.

• **STRICT**. A function is assumed to be not strict unless adorned with STRICT. A strict function will always return NULL if any inputs are NULL and doesn’t bother evaluating the function so saves some processing.

When building SQL functions, you should be careful about using STRICT as it will prevent index usage as described in STRICT on SQL Functions

• **COST** is a relative measure of computational intensiveness. SQL and PL/pgSQL functions default to 100 and C functions to 1. This affects the order functions will be evaluated in a WHERE clause and also the likeliness of caching. The higher the value, the more costly the function is assumed to be.

• **ROWS** is only set for set returning functions and is an estimate of how many rows will be returned; used by planner to arrive at best strategy. Format would be ROWS 100.

• **SECURITY DEFINER** is an optional clause that means run under the context of the owner of the function. If left out, a function runs under the context of the user running the function.

Trusted and Untrusted Languages

Function languages can be divided into two levels of trust. Many—but not all—languages offer both a trusted and untrusted version.

• **Trusted**—Trusted languages are languages that don’t have access to the filesystem beyond the data cluster and can’t execute OS commands. They can be created by any user.

Languages like SQL, PL/pgSQL, PL/Perl are trusted. It basically means they can’t do damage to the underlying OS.

• **Untrusted**—Untrusted languages are those that can interact with the OS and even call on webservices or execute functions on the OS.

Functions written in these languages can only be created by super users, however, a super user can delegate rights for another user to use them by using the SECURITY DEFINER setting.

By convention, these languages have a u at the end of the name to denote that they’re untrusted—for instance, PL/PerlU, PL/PythonU.

Writing Functions with SQL

Writing SQL functions is fast and easy. Take your existing SQL statements, add a functional header and footer, and you’re done.

The ease does mean you’ll sacrifice flexibility. You won’t have fancy control languages to create conditional execution branches. You can’t have more than one SQL statement.

More restrictively, you can’t run dynamic SQL statements that you piece together based on parameters as you can in most other languages.

 Your basic scalar returning SQL function is shown below

**Example SQL function to return key of inserted record**

create table logs2(log\_id serial PRIMARY KEY, user\_name varchar(20), description text, log\_ts timestamptz);

CREATE OR REPLACE FUNCTION ins\_logs(param\_user\_name varchar, param\_description text)

RETURNS integer AS

$$

INSERT INTO logs2(user\_name, description) VALUES($1, $2)

RETURNING log\_id;

$$

LANGUAGE 'sql' VOLATILE;

To call the function, we would execute:

SELECT ins\_logs('peter given', 'this is a test') As new\_id;

Similarly, you can update data with an SQL function and return a scalar or void as shown below

**Example SQL function to update a record**

CREATE OR REPLACE FUNCTION upd\_logs(log\_id integer, param\_user\_name varchar,

param\_description text)

RETURNS void AS

$$

UPDATE logs2 SET user\_name = $2, description = $3, log\_ts = CURRENT\_TIMESTAMP

WHERE log\_id = $1;

$$

LANGUAGE 'sql' VOLATILE;

To execute:

SELECT upd\_logs(1,'joe', ' another test');

Prior to 9.2, SQL functions could only use the ordinal position of the input arguments in the body of the function. After 9.2, you have the option of using named arguments, for example you can write param\_1, param\_2 instead of $1, $2.

Functions in almost all languages can return sets. SQL functions can also return sets.

There are three common approaches of doing this, using ANSI-SQL standard RETURNS TABLE syntax, using OUT parameters, or returning a composite data type.

The RETURNS TABLE approach requires PostgreSQL 8.3 or above, but is closer to what you’ll see in other relational databases. In the next example, we demonstrate how to write the same function returning tables and out parameters.

**Examples of function returning sets**

**Using returns table:**

CREATE FUNCTION sel\_logs\_rt(param\_user\_name varchar)

RETURNS TABLE (log\_id int, user\_name varchar(20), description text, log\_ts timestamptz) AS

$$

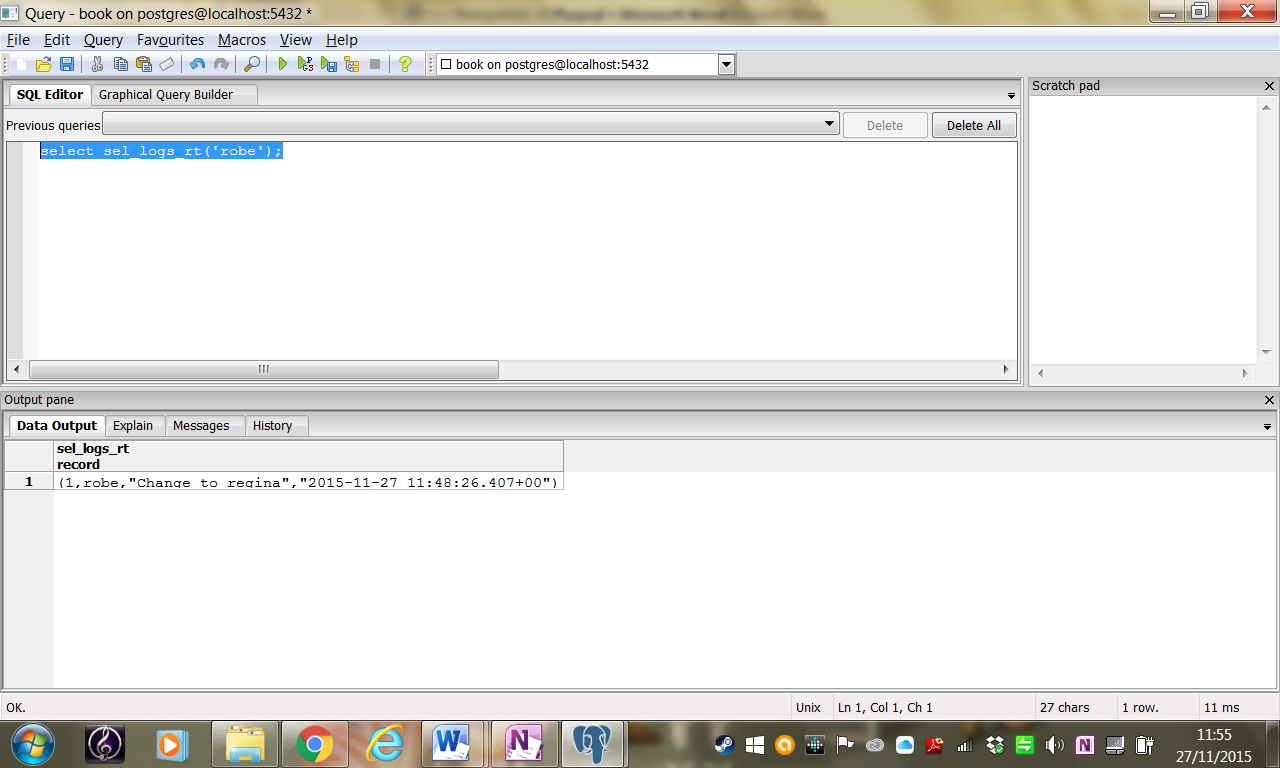
SELECT log\_id, user\_name, description, log\_ts FROM logs2 WHERE user\_name = $1;

$$

LANGUAGE 'sql' STABLE;

Now try

select sel\_logs\_rt('joe');



Using OUT parameters:

CREATE FUNCTION sel\_logs\_out(param\_user\_name varchar, OUT log\_id int

, OUT user\_name varchar, OUT description text, OUT log\_ts timestamptz)

RETURNS SETOF record AS

$$

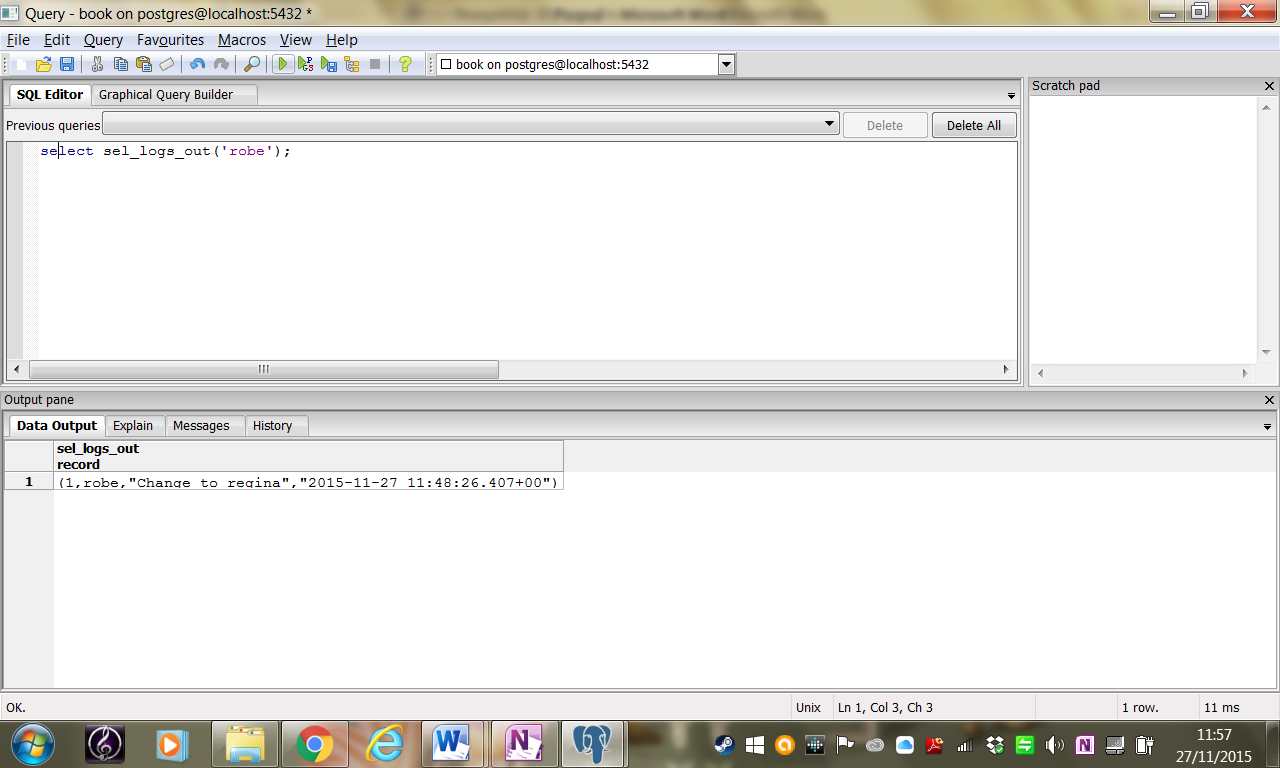
SELECT \* FROM logs2 WHERE user\_name = $1;

$$

LANGUAGE 'sql' STABLE;

Now try

select sel\_logs\_out('joe');



Writing PL/pgSQL Functions

When your functional needs exceed reaches beyond SQL, PL/pgSQL is the most common option.

PL/pgSQL stands apart from SQL in that you can declare local variables using DECLARE, you can have control flow, and the body of the function needs be enclosed in a BEGIN..END block.

To demonstrate the difference, we have rewritten the example above as a PL/pgSQL function.

**Example: Function to return a table using PL/pgSQL**

CREATE OR REPLACE FUNCTION sel\_logs\_rt2(param\_user\_name varchar)

RETURNS TABLE (user\_name varchar(20), description text, log\_ts timestamptz) AS

$$

BEGIN

RETURN QUERY

SELECT user\_name, description, log\_ts

FROM logs2 WHERE user\_name = param\_user\_name;

END;

$$

LANGUAGE 'plpgsql' STABLE;

Test with

select sel\_logs\_(‘joe’);

The manual for PL/pgsql can be found here: <http://www.postgresql.org/docs/8.3/static/plpgsql.html>

Writing PL/Python Functions

Python is a slick language with a vast number of available libraries. PostgreSQL is the only database we know of that’ll let you compose functions using Python.

PostgreSQL 9.0+ supports both Python 2 and Python 3.

You can have both plpython2u andplpython3u installed in the same database, but you can’t use them in the same session.

This means that you can’t write a query that contains both plpython2u and plpython3u-written functions.

**Exercise:**

In order to use PL/Python, you first need to install Python on your server. For Windows and Mac OS, Python installers are available at <http://www.python.org/download/>.

For Linux/Unix systems, Python binaries are usually available via the various distros. For details, refer to <http://www.postgresql.org/docs/current/interactive/plpython.html>

**Homework**

**Install Python for your version of PostgreSQL and test it with the Python functions below**

**You should install a minor version of Python that matches what your plpythonu extensions were compiled against.**

**For example, if your plpython2u is compiled against 2.7, then you’ll need to install Python 2.7.**

**See the Installation Notes in the installed Postgres documentation directory.**

**I installed python 3.3 from the python site and copied dll to postgresql/lib folder**

**I then issued the** CREATE LANGUAGE plpython3u command

After you have Python on your server, proceed to install the PostgreSQL Python extension using the commands below:

CREATE EXTENSION plpython3u;

(You will find a third extension called plpythonu, which is an alias for plpython2u and intended for backwards compatibility.)

Make sure you have Python properly running on your server before attempting to install the extension or else you will run into errors.

Basic Python Function

PostgreSQL automatically converts PostgreSQL datatypes to Python datatypes and back. PL/Python is capable of returning arrays and composite types.

You can use PL/Python to write triggers and create aggregate functions. This is demonstrated in the Postgres OnLine Journal, in PL/Python <http://www.postgresonline.com/journal/index.php?/archives/99-Quick-Intro-to-PLPython.html>

Python allows you to perform feats that aren’t possible in PL/pgSQL.

In the next Example, we demonstrate how to write a PL/Python function that does a text search of the online PostgreSQL document resource site.

**Example: Searching PostgreSQL docs using PL/Python**

CREATE OR REPLACE FUNCTION postgresql\_help\_search(param\_search text)

RETURNS text AS

$$

import urllib, re

response = urllib.urlopen('http://www.postgresql.org/search/?u=%2Fdocs%2Fcurrent%2F&q=' +param\_search)

raw\_html = response.read()

result = raw\_html[raw\_html.find("<!-- docbot goes here -->"):raw\_html.find("<!--pgContentWrap -->") - 1]

result = re.sub('<[^<]+?>', '', result).strip()

return result

$$

LANGUAGE plpython3u SECURITY DEFINER STABLE;

* Import the libraries we’ll be using.
* Web search concatenating user input parameters.
* Read response and save html to a variable called raw\_html.
* Save the part of the raw\_html that starts with <!-- docbot goes here --> and ends just before the beginning of <!-- pgContentWrap -->.
* Strip HTML and white space from front and back and then re-save back to variable called result.
* Return final result.

Calling Python functions is no different than functions written in other languages.

In the next Example, we use the function we created in the last Example to output the result with three search terms.

**Example: Using Python function in a query**

SELECT search\_term, left(postgresql\_help\_search(search\_term), 125) As result FROM (VALUES

('regexp\_matc'), ('pg\_trgm'), ('tsvector')) As X(search\_term);

search\_term | result

regexp\_match | Results 1-7 of 7.

1. PostgreSQL: Documentation: Manuals: Pattern Matching [1.46]

...matching a POSIX regular

pg\_trgm|Results 1-8 of 8.

 1. PostgreSQL: Documentation: Manuals: pg\_trgm [0.66]

...pg\_trgm The pg\_trgm module provide

tsvector | Results 1-20 of 32.

Result pages: 1 2 Next

1. PostgreSQL: Documentation: Manuals: Text Search Functions

(3 rows)

Recall that PL/Python is an untrusted language without a trusted counterpart. This means it’s capable of interacting with the filesystem of the OS and a function can only be created by super users.

Our next example uses PL/Python to retrieve file listings from a directory.

Keep in mind that PL/Python function runs under the context of the postgres user account, so you need to be sure that account has adequate access to the relevant directories.

**Example: List files in directories**

CREATE OR REPLACE FUNCTION list\_incoming\_files()

RETURNS SETOF text AS

$$

import os

return os.listdir('c:\\')

$$

LANGUAGE 'plpython3u' VOLATILE SECURITY DEFINER;

You can run the function with the query below:

SELECT filename FROM list\_incoming\_files() As filename WHERE filename ILIKE '%.csv'

**Trigger Functions**

No database of merit should be without triggers to automatically detect and handle changes in data.

Triggers can be added to both tables and views.

PostgreSQL offers both statement-level triggers and row-level triggers. Statement triggers run once per statement, while row triggers run for each row called.

For instance, suppose we execute an UPDATE command that affects 1,500 rows. A statement-level trigger will fire only once, whereas the row-level trigger can fire up to 1,500 times.

More distinction is made between a BEFORE, AFTER, and INSTEAD OF trigger.

A BEFORE trigger fires prior to the execution of the command giving you a chance to cancel and change data before it changes data.

An AFTER trigger fires afterwards giving you a chance to retrieve revised data values. AFTER triggers are often used for logging or replication purposes.

The INSTEAD OF triggers run instead of the normal action. INSTEAD OF triggers can only be used with views.

BEFORE and AFTER triggers can only be used with tables.

To gain a better understanding of the interplay between triggers and the underlying command look at <https://www.postgresql.org/docs/8.3/trigger-definition.html>

PostgreSQL offers specialized functions to handle triggers. These trigger functions act just like any other function and have the same basic structure as your standard function.

Each trigger must have exactly one associated triggering function. To apply multiple triggering functions, you must create multiple triggers against the same event.

 You can use almost any language to write trigger functions, with SQL being the notable exception.

Our example below uses PL/pgSQL, which is by far the most common language for writing triggers.

You will see that we take two steps: First, we write the trigger function.

Next, we attach the trigger function to the appropriate trigger, a powerful extra step that decouples triggers from trigger functions.

**Example: Trigger function to timestamp new and changed records**

CREATE OR REPLACE FUNCTION trig\_time\_stamper() RETURNS trigger AS

$$

BEGIN

NEW.log\_ts := CURRENT\_TIMESTAMP;

RETURN NEW;

END;

$$

LANGUAGE plpgsql VOLATILE;

CREATE TRIGGER trig\_1

BEFORE INSERT OR UPDATE OF user\_name

ON logs2

FOR EACH ROW

EXECUTE PROCEDURE trig\_time\_stamper();

* Define the trigger function. This function can be used on any table that has a log\_ts column . It changes the value of the log\_ts field of the new record before returning. Trigger functions that change values of a row should only be called in the BEFORE event, because in the AFTER event, all updates to the NEW record will be ignored.
* The trigger will fire before the record is committed.
* Binds the trigger to the table.

Testing trigger

Select \* from logs2;

Note the timestamp on log\_id=1 record

update logs2 set user\_name=’jack’ where log\_id=1;

Select \* from logs2;

Note the timestamp on log\_id=1 record

The timestamp should have changed

HERE --- 21/11/18

Aggregates

You need to install the database from the book – postgresql\_book.sql. Copy it to the postgresql folder

**create database week10;**

**\c week10**

***\i postgresql\_book.sql***

***Or from command line prompt***

psql -Upostgres –dweek10 -1 -f postgresql\_book.sql

The postgresql\_book.sql is a plain text SQL backup that should be restored with psql.

It consists of the census data from America. You will need to copy this file to the scripts folder in the postgresql installation and load it from the command line with \i postgresql\_book.sql

This will install a census schema with three tables; facts, lu\_facts\_types, lu\_tracts and a staging schema with two tables; factfinder\_import, pop\_import

Also create the following view:

CREATE OR REPLACE VIEW census.vw\_facts AS

SELECT lf.fact\_type\_id, lf.category, lf.fact\_subcats, lf.short\_name

, f.tract\_id, f.yr, f.val, f.perc

FROM census.facts As f

INNER JOIN census.lu\_fact\_types As lf

ON f.fact\_type\_id = lf.fact\_type\_id;

Aggregates are another type of specialized function offered up by PostgreSQL.

In many other databases, you’re limited to ANSI-SQL aggregate functions such as MIN(), MAX(), AVG(), SUM(), and COUNT(). You can define your own aggregates in PostgreSQL.

You can write aggregates in almost any language, SQL included.

An aggregate is generally composed of one or more functions.

It must have at least a state transition function to perform the computation and optional functions to manage initial and final states.

See: <https://www.postgresql.org/docs/9.1/sql-createaggregate.html>

 Regardless of which languages you code the aggregate, the glue that brings them all together looks the same for all and is of the form:

CREATE AGGREGATE myagg(datatype\_of\_input)

(SFUNC=state\_function\_name, STYPE=state\_type, FINALFUNC=final\_func\_name,

INITCOND=optional\_init\_state\_value);

The final function is optional, but if specified must take as input the result of the state function.

The state function always takes as input the datatype\_of\_input and result of last state function call.

The initial condition is also optional. When present, it is used to initialize the state value.

We will demonstrate how to create a geometric mean aggregate function with SQL.

A geometric mean is the nth root of a product of n positive numbers ((x1\*x2\*x3...Xn)^(1/n)) so for example the geometric mean of the two numbers 2 and 8 is the square root of 2\*8 = 4 (the arithmetic root is 5)

It has various uses in finance, economics, and statistics.

A geometric mean may have more meaning than an arithmetic mean when the numbers are of vastly different scales.

A more suitable computational formula uses logarithm to convert a multiplicative process to an additive one (EXP(SUM(LN(x))/n)).

We’ll be using this method in our example.

For our geomeric mean aggregate, we’ll use two functions: a state function to add the logs and a final exponential function

We will also specify an initial condition of zero when we put everything together.

**Example: Geometric mean aggregate: State function**

CREATE OR REPLACE FUNCTION geom\_mean\_state(prev numeric[2], next numeric)

RETURNS numeric[2] AS

$$

SELECT CASE WHEN $2 IS NULL or $2 = 0 THEN $1

ELSE ARRAY[COALESCE($1[1],0) + ln($2), $1[2] + 1] END;

$$

LANGUAGE sql IMMUTABLE;

Our transition state function, as shown above, takes two inputs: the previous state passed in as a one-dimensional array with two elements and also the next element in the aggregation process.

If the next element is NULL or zero, the state function returns the prior state. Otherwise, it’ll return an array where the first element is the logarithmic sum and the second being the current count.

We will need a final function that takes the sum from the state transition and divides by the count.

**Example:. Geometric mean aggregate: Final function**

CREATE OR REPLACE FUNCTION geom\_mean\_final(numeric[2])

RETURNS numeric AS

$$

SELECT CASE WHEN $1[2] > 0 THEN exp($1[1]/$1[2]) ELSE 0 END;

$$

LANGUAGE sql IMMUTABLE;

Now we stitch all the pieces together in our aggregate definition.

**Example: Geometric mean aggregate: Putting all the pirces together**

CREATE AGGREGATE geom\_mean(numeric) (SFUNC=geom\_mean\_state, STYPE=numeric[], FINALFUNC=geom\_mean\_final, INITCOND='{0,0}');

Let’s take our geom\_mean() function for a test drive.

We’re going to compute a heuristic rating for the top five counties with least owner occupied homes in Massachusetts.

SELECT left(tract\_id,5) As county, geom\_mean(val) As div\_county

FROM census.vw\_facts

WHERE category = 'Population' AND short\_name != 'owner occupied'

GROUP BY county

ORDER BY div\_county DESC LIMIT 5;

county | div\_county

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

25025 | 89.3936028490478146

25013 | 85.9954645724462369

25017 | 77.3285846695027088

25027 | 77.1245786027256929

25009 | 76.0448526639188004