SCHARP

at FRED HUTCH

Writing PostgreSQL Functions
and how to debug them

By Lloyd Albin

SCHARP
at FRED HUTCH

What we are going to cover

- Function Basics
  - *Procedural Languages*
  - *Function Behavior*
  - *NULL INPUT*
  - *Security*
  - *Syntax*
  - *Inline (Function) Syntax*
- Basic Example Functions
  - *Functional Indexes*
  - *Real World Example of Functional Indexes*
  - *Case Insensitive Indexes*
- Advanced Example Functions
  - *Working with View Definitions*
- Inline Example Functions
  - *Counting all the rows for all tables*
- Debugging Functions
  - *Debugging your Function*

SCHARP at FRED HUTCH

The Basics

Procedural Languages

Procedural Languages (pl)

- Pre-Installed Languages:
  - SQL * ** ***
  - C
  - internal
- Installable Languages that come with Postgres:
  - plpgsql * ** ***
  - plperl *
  - plperlu
  - pltcl *
  - plpython
- Other Downloadable Languages:
  - pljava
  - plphp
  - plpy
  - plr
  - plruby
  - plscheme
  - plsh
  - plv8 *
  - plcoffee * (Coffee Script)
  - plls * (LiveScript)
- Cloud Support
  - * Supported by Amazon
  - ** Supported by Google
  - *** Supported by Microsoft

SCHARP at FRED HUTCH

What Languages can be Installed

- With this query you are able to find out the languages that can be installed in your database.
- This is a good query to know if you are not the dba and need to find out what the server supports.
- Some of these may only be installable by the dba or superuser. See the next slide for one what may be installed by the database owner.

```
SELECT name, comment
FROM pg_available_extensions
WHERE comment LIKE '%language%';
```

Name	Comment
plperl	PL/Perl procedural language
plpgsql	PL/pgSQL procedural language

SCHARP at FRED HUTCH

What Languages can be Installed by the Database Owner

- This query will list the languages that may be installed by the database owner.

```
SELECT tmplname
FROM pg_pltemplate
WHERE tmpldbacreate IS TRUE;
```

tmplname
plperl
plpgsql

SCHARP at FRED HUTCH

How to Install a Language

- Since Postgres 9.1 most, if not all, languages have been made into “extensions’ and can use the “CREATE EXTENSION’ command.

```
CREATE EXTENSION plperl;
CREATE EXTENSION plpgsql;
```

SCHARP at FRED HUTCH

How to UnInstall a Language

- Since Postgres 9.1 most, if not all, languages have been made into “extensions’ and can use the “DROP EXTENSION’ command.

```
DROP EXTENSION plperl;
DROP EXTENSION plpgsql;
```

SCHARP at FRED HUTCH

Function Volatility

The Basics

Immutable

- IMMUTABLE indicates that the function cannot modify the database and always returns the same result when given the same argument values; that is, it does not do database lookups or otherwise use information not directly present in its argument list.
- If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.
- What this means is these types of functions can be used as a type converter or indexer.

SCHARP at FRED HUTCH

Stable

- STABLE indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements.
- This is the appropriate selection for functions whose results depend on database lookups, parameter variables (such as the current time zone), etc.
- It is inappropriate for AFTER triggers that wish to query rows modified by the current command.
- Also note that the current_timestamp family of functions qualify as stable, since their values do not change within a transaction.

SCHARP at FRED HUTCH

Volatile

- VOLATILE indicates that the function value can change even within a single table scan, so no optimizations can be made.
- Relatively few database functions are volatile in this sense; some examples are random(), currval(), timeofday().
- But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; some example are random(), currval(), timeofday()

NULL Input

The Basics

What to do with NULL?

- CALLED ON NULL INPUT – This is the default if you don't say anything. You will need to handle the NULL's within your code.
- RETURNS NULL ON NULL INPUT or STRICT – This will cause the function to return NULL when any of the input values are NULL. The body of the function is never executed.

Security

The Basics

SCHARP at FRED HUTCH

Security

- SECURITY INVOKER – This is the default. The function is to be executed with the privileges of the user that calls it.
- SECURITY DEFINER – This specifies that the function is to be executed with the privileges of the user that created it. This could be used to have a function update a table that the calling user does not have permissions to access, etc.

Syntax

The Basics

Create Function Syntax

- This is the basic Syntax for a Function.

```
CREATE OR REPLACE FUNCTION (
argname text
)
RETURNS numeric AS
$body$
…
$body$
LANGUAGE 'sql'
IMMUTABLE | STABLE | VOLATILE
RETURNS NULL ON NULL INPUT
SECURITY DEFINER
;
```

SCHARP at FRED HUTCH

Additional Return Types

- void – This is for when the trigger should not return anything. It is just doing some background process for you.
- trigger – This must be set for all trigger functions.
- boolean, text, etc – This is for a single values being passed back.
- SET OF schema.table – This is for returning multiple rows of data. This can either point to an existing table or a composite type to get the table/field layout.

SCHARP at FRED HUTCH

Inline Syntax

Bulk Delete part of the Table

Inline Function Syntax

- Inline functions are the same as normal functions that have no input parameters and returns void.
- But if you want to send input or output you can get around the issues by read/writing tables, even if they are temp tables.

```
DO $body$
…
$body$;
```

```
DO $body$
DECLARE
…
BEGIN
…
END
$body$
LANGUAGE 'plpgsql'
;
```

SCHARP at FRED HUTCH

Functional Indexes

Basic Example

The Problem

- We receive faxes that are multi-page TIFF's. The TIFF file name are called the raster id. We have data where we have the full path of the file name, the raster id and the raster id with page number.

- Examples:
  - 0000/000000
  - 0000/0000001111
  - /studydata/studyname/0000/000000

SCHARP at FRED HUTCH

Finding the Raster ID

- The first thing to do, is to be able to find the Raster ID, no matter which format is supplied.
- IMMUTABLE is required to be able to use this function as part of an index.
- This function was written to fit on this page. Some of the other items to put inside the real function would be:
  - We should check the return to make sure a '/' is in the correct spot and through an error if it is not. "R" is also valid.
  - If the raster is less then 23 characters to return an error, except for when it is 11 or 15 in length.

```
CREATE FUNCTION find_raster (raster varchar)
RETURNS VARCHAR(11) AS
$$
BEGIN
    CASE length(raster)
        WHEN 11 THEN
            -- Format: 1234/567890
            -- Returns: 1234/567890
RETURN raster;
        WHEN 15 THEN
            -- Format: 1234/5678901234
            -- Returns: 1234/567890
RETURN substr(raster, 1, 11);
        ELSE
            -- Format: /study_data/study_name/1234/567890
            -- Returns: 1234/567890
RETURN substr(raster, length(raster)-10, 11);
    END CASE;
END;
$$
LANGUAGE plpgsql
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

SCHARP at FRED HUTCH

Examples of the find_raster Function

```
-- Test returning of Raster ID when Submitting Raster ID
SELECT find_raster('1234/567890');
-- Returns: 1234/567890

-- Test returning of Raster ID when Submitting Raster ID
with 4 Digit Page Number
SELECT find_raster('1234/5678901234');
-- Returns: 1234/567890

-- Test returning of Raster ID when Submitting Filename
that includes Raster ID
SELECT find_raster('/study_data/study_name/1234/567890');
-- Returns: 1234/567890
```

SCHARP at FRED HUTCH

Create Tables for Testing

- In the real tables there would much more data, this is just the minimum amount for showing off this feature.

```
-- Format: 0000/000000
CREATE TABLE raster (
    raster VARCHAR(11)
);

-- Format: 0000/0000000000
CREATE TABLE raster_page (
    raster VARCHAR(15)
);

-- Format: /study_data/study_name/0000/0000000000
CREATE TABLE raster_file (
    raster VARCHAR
);
```

SCHARP at FRED HUTCH

Creating Test Data

- We have real data, but to actually be able to demo this live and not show real data, we can use the microseconds for each command to create data for us. Creating two tables with 100,000 records and one table with 500,000 takes about a minute.

```
-- Now lets create a function to create a lot of data.
CREATE OR REPLACE FUNCTION create_data ()
RETURNS INTEGER AS
$$
DECLARE
  count integer;
  r raster%rowtype;
BEGIN
  count := 0;
  LOOP
    INSERT INTO raster
      VALUES (to_char(clock_timestamp(), '0MS/US'));
      count := count + 1;
      EXIT WHEN count >= 100000;
  END LOOP;
  INSERT INTO raster_file
    SELECT '/study_data/study_name/' || raster.raster
    FROM raster;
  count := 5;
  LOOP
    INSERT INTO raster_page
      SELECT raster.raster ||
      substr(to_char(count, '0009'),2,4) FROM raster;
    count := count - 1;
    EXIT WHEN count = 0;
  END LOOP;
RETURN 1;
END;
$$
LANGUAGE plpgsql;

SELECT create_data();
```

SCHARP at FRED HUTCH

What a Normal Join Looks Like: raster_page

Hash Left Join (cost=12430.05..23339.48 rows=140556 width=88) (actual time=4334.613..4980.760 rows=540 loops=1)

  Hash Cond: ((public.raster.raster)::text = substr((raster_page.raster)::text, 1, 11))

  -> Limit (cost=931.03..932.90 rows=100 width=40) (actual time=117.630..118.284 rows=100 loops=1)

        -> Seq Scan on raster (cost=0.00..1060.56 rows=56956 width=40) (actual time=0.025..65.829 rows=50100 loops=1)

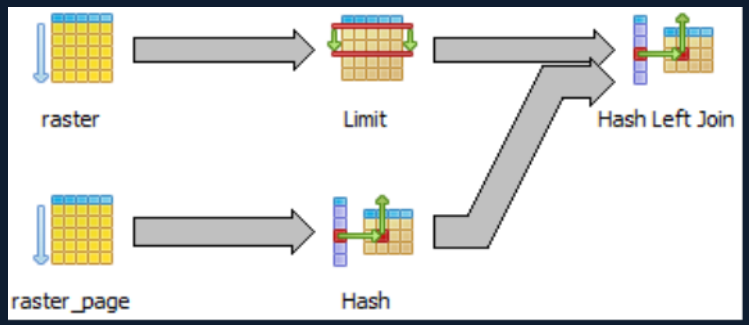
  -> Hash (cost=5514.12..5514.12 rows=281112 width=48) (actual time=4215.010..4215.010 rows=500000 loops=1)

        -> Seq Scan on raster_page (cost=0.00..5514.12 rows=281112 width=48) (actual time=0.028..1001.646 rows=500000 loops=1)

Total runtime: 4982.527 ms

Total runtime: 4.982527 s

```sql
SELECT raster.*, raster_page.*
FROM (
  SELECT * FROM raster OFFSET 50000 LIMIT 100
) raster
LEFT JOIN raster_page
  ON raster.raster = substr(raster_page.raster, 1, 11);
```



SCHARP at FRED HUTCH

What a Normal Join Looks Like: raster_file

Hash Left Join (cost=4516.50..10754.24 rows=50000 width=75) (actual time=1309.087..1445.892 rows=109 loops=1)

  Hash Cond: ((public.raster.raster)::text = substr((raster_file.raster)::text, (length((raster_file.raster)::text) - 10), 11))

  -> Limit (cost=745.50..746.99 rows=100 width=12) (actual time=128.356..129.015 rows=100 loops=1)

        -> Seq Scan on raster (cost=0.00..1491.00 rows=100000 width=12) (actual time=0.025..75.858 rows=50100 loops=1)

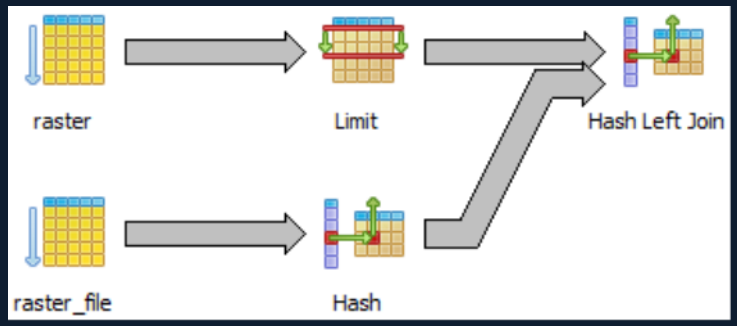
  -> Hash (cost=1788.00..1788.00 rows=100000 width=35) (actual time=1180.239..1180.239 rows=100000 loops=1)

        -> Seq Scan on raster_file (cost=0.00..1788.00 rows=100000 width=35) (actual time=0.141..222.528 rows=100000 loops=1)

Total runtime: 1446.589 ms

Total runtime: 1.446589 s

```sql
SELECT raster.*, raster_file.*
FROM (
  SELECT * FROM raster OFFSET 50000 LIMIT 100
) raster
LEFT JOIN raster_file
  ON raster.raster = substr(raster_file.raster,
    length(raster_file.raster)-10, 11);
```



Adding the Indexes

```
CREATE INDEX [ name ] ON table ( expression )
```

expression

- An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

```
CREATE INDEX raster_raster_idx
  ON raster find_raster(raster);

CREATE INDEX raster_file_raster_idx
  ON raster_file find_raster(raster);

CREATE INDEX raster_page_raster_idx
  ON raster_page find_raster(raster);
```

SCHARP at FRED HUTCH

What the New Join Looks Like: raster_page

Nested Loop Left Join (cost=745.75..84448.59 rows=250000 width=88) (actual time=117.717..140.916 rows=540 loops=1)

  -> Limit (cost=745.50..746.99 rows=100 width=40) (actual time=116.171..116.685 rows=100 loops=1)

        -> Seq Scan on raster (cost=0.00..1491.00 rows=100000 width=40) (actual time=0.026..65.717 rows=50100 loops=1)

  -> Index Scan using raster_page_raster_idx on raster_page (cost=0.25..180.76 rows=2500 width=48) (actual time=0.168..0.204 rows=5 loops=100)

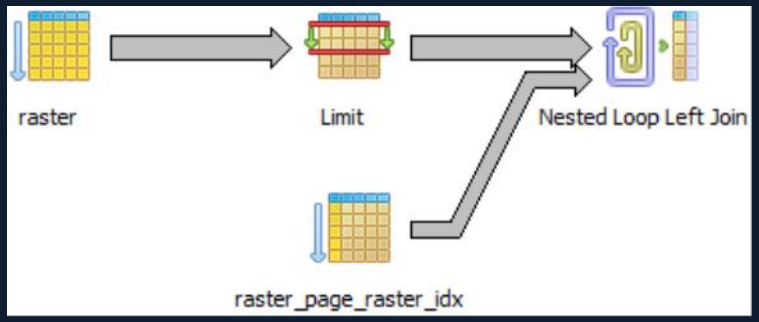
        Index Cond: ((public.raster.raster)::text = (find_raster(raster_page.raster))::text)

Total runtime: 141.809 ms
Total runtime: 0.141809 s

35 Times Faster

Before Indexes:
Total runtime: 4982.527 ms
Total runtime: 4.982527 s

```sql
SELECT raster.*, raster_page.*
FROM (
  SELECT * FROM raster OFFSET 50000 LIMIT 100
) raster
LEFT JOIN raster_page
  ON raster.raster = find_raster(raster_page.raster);
```



SCHARP at FRED HUTCH

What the New Join Looks Like: raster_file

Nested Loop Left Join (cost=745.75..18567.32 rows=50000 width=75) (actual time=117.587..128.616 rows=109 loops=1)

  -> Limit (cost=745.50..746.99 rows=100 width=12) (actual time=116.578..117.111 rows=100 loops=1)

        -> Seq Scan on raster (cost=0.00..1491.00 rows=100000 width=12) (actual time=0.022..65.574 rows=50100 loops=1)

  -> Index Scan using raster_file_raster_idx on raster_file (cost=0.25..46.94 rows=500 width=35) (actual time=0.091..0.102 rows=1 loops=100)

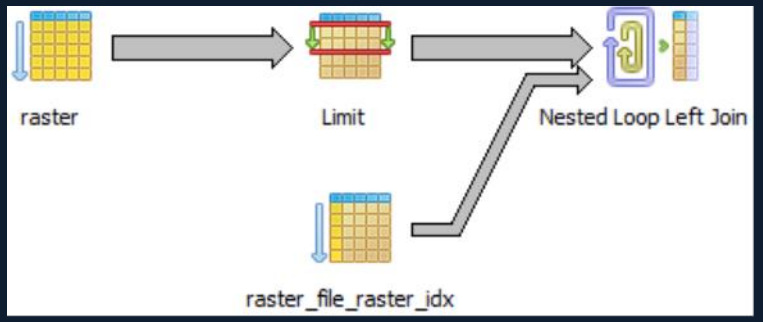
        Index Cond: ((public.raster.raster)::text = (find_raster(raster_file.raster))::text)

Total runtime: 129.261 ms
Total runtime: 0.129261 s

11 Times Faster

Before Indexes:
Total runtime: 1446.589 ms
Total runtime: 1.446589 s

```sql
SELECT raster.*, raster_file.*
FROM (
  SELECT * FROM raster OFFSET 50000 LIMIT 100
) raster
LEFT JOIN raster_file
  ON raster.raster = find_raster(raster_file.raster);
```



SCHARP at FRED HUTCH

Real Word Example

Real World Example of Functional Indexes

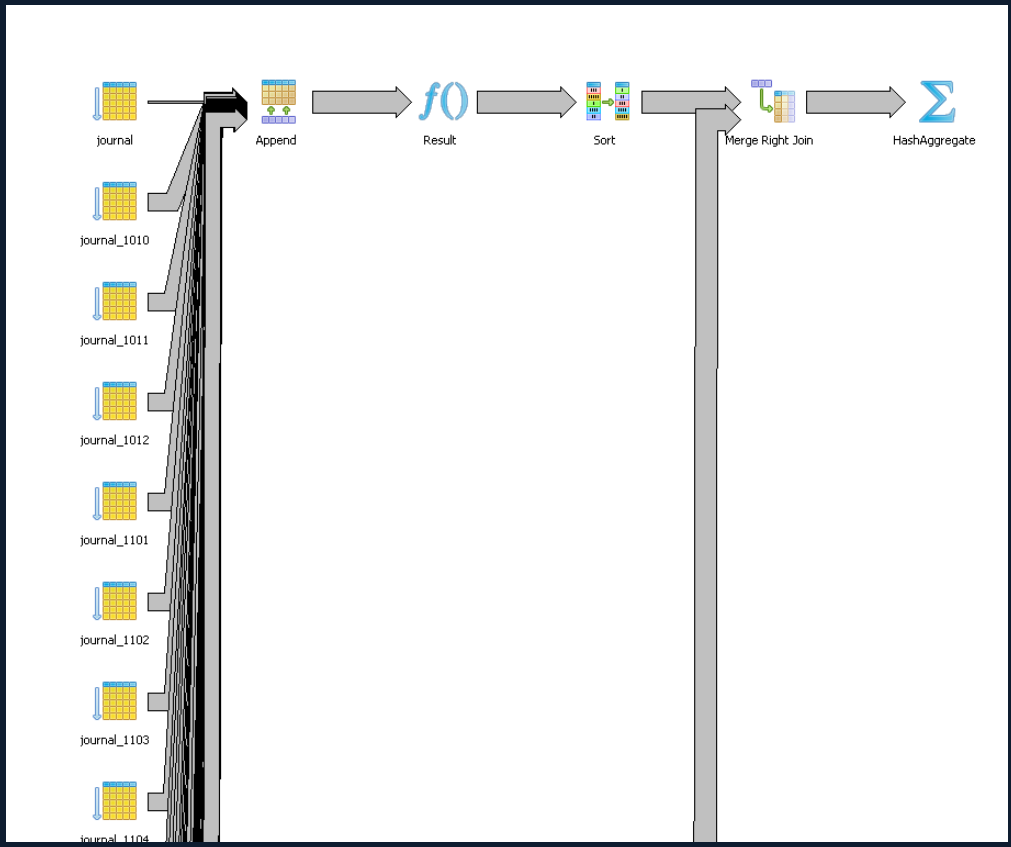
Joins between Journal & Faxlog

- Journal used in testing is 10/1997 to 4/2012, more than the 18 months on production.

Total runtime: 197813.222 ms
Total runtime: 197.813222 s
Total runtime: 3.296887033 m

Journal – 16,384,930 Rows of Data
Faxlog – 1,244,754 Rows of Data

```
SELECT DISTINCT journal.username
FROM protimp.faxlog
LEFT JOIN protimp.journal
  ON substring(faxlog.fax_loc FROM LENGTH(faxlog.fax_loc)
    - 8) = substring(journal.dfraster from 1 for 9)
WHERE faxlog.fax_time >= '1/1/2012'
  AND faxlog.fax_time < '1/2/2012';
```



SCHARP at FRED HUTCH

Joins between Journal & Faxlog

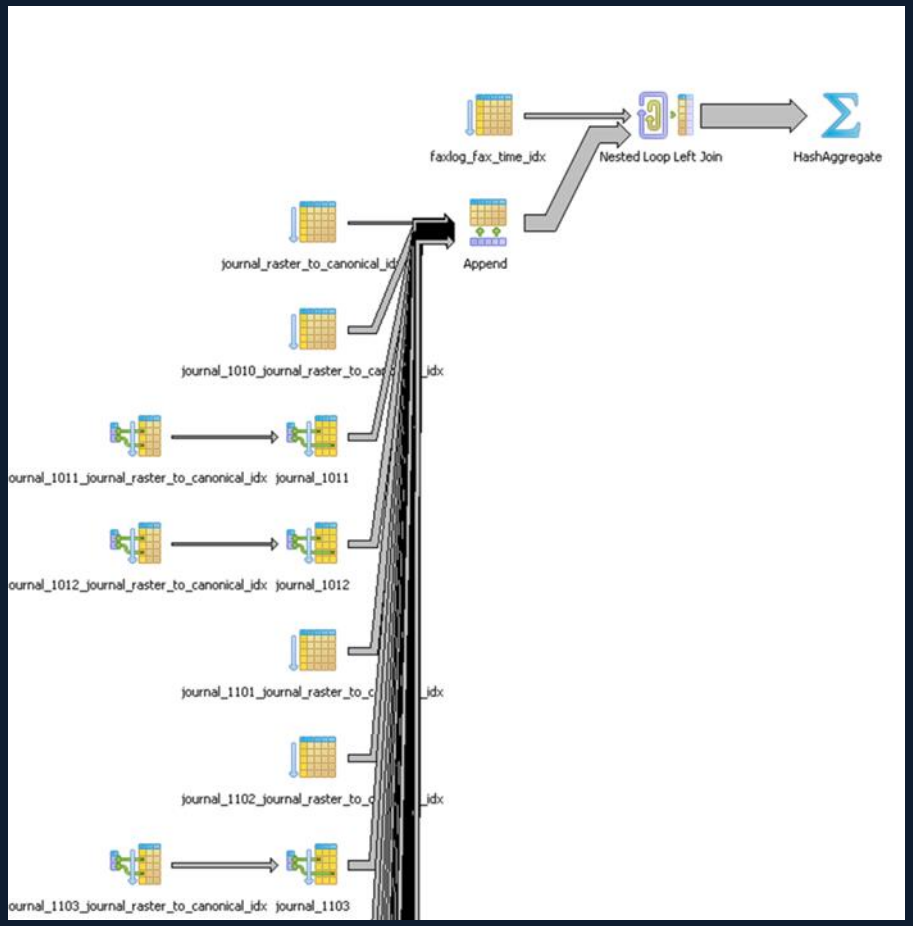
Total runtime: 22.368 ms
Before Indexes: 197813.222 ms
Before Indexes: 3.296887033 m

Journal – 16,384,930 Rows of Data
Faxlog – 1,244,754 Rows of Data

8,844 Times Faster

- You will notice that we used individual functions instead of a combined function. This allowed us to write the functions in pure sql and remove the logic from each function making the functions faster, which is needed with big data.

```
SELECT DISTINCT journal.username
FROM protimp.faxlog
LEFT JOIN protimp.journal
  ON ist.faxlog_raster_to_canonical(faxlog.fax_loc)
    = ist.journal_raster_to_canonical(journal.dfraster)
WHERE faxlog.fax_time >= '1/1/2012'
  AND faxlog.fax_time < '1/2/2012';
```



SCHARP at FRED HUTCH

Case Insensitive Indexes

Bulk Inserting from another Table/Query

Making your queries case insensitive

- What if you want to search for an employee like “deForest”. Many people will mistype this last name and a simple “=” will miss finding this employee. You may fix this by creating a lower case index and then using “= lower(‘deForest’)”. This will make finding this employee much faster since the conversions for the table are in the index and only the one conversion needs to be performed.

```
SELECT *
FROM public.name_test
WHERE lower(name) = lower('deForest');
```

SCHARP at FRED HUTCH

Making a unique case insensitive index

- By default indexes are case sensitive. You could run into problems if one person enters “deForest” and another person enters “DeForest”.
- A case insensitive unique index solves this problem.

```
CREATE TABLE public.name_test (
  name TEXT
) WITHOUT OIDS;

CREATE UNIQUE INDEX name_test_idx ON public.name_test
  USING (lower(name));

INSERT INTO public.name_test VALUES ('deForest');
Query OK, 1 rows affected (execution time: 0 ms; total time: 15 ms)

INSERT INTO public.name_test VALUES ('DeForest');
ERROR: duplicate key value violates unique constraint "name_test_idx"
DETAIL: Key (lower(name))=(deforest) already exists.
```

SCHARP at FRED HUTCH

Working with View Definitions

Refactoring a database

View Definitions

- Here is a basic query that will return all queries, including system queries. You may want to exclude the system and temp queries from your results.

```
SELECT
    definition,
    schemaname,
    viewname
FROM pg_catalog.pg_views;
```

definition	schemaname	viewname
SELECT definition, schemaname, viewname FROM pg_catalog.pg_views;	public	test_view

SCHARP at FRED HUTCH

Basic Function

- The problem is what happens when the view fails to execute.
- EXECUTE will run the view and ignore the results (data).
- When updating a table, sometimes you will need to drop and re-create the views. If you do this as two loop, the first one to write the view defs into a text[] and then a second one to loop through the text[] and re-create the views.

```
DECLARE
    view_failures INTEGER;
    check_views RECORD;
BEGIN
    view_failures = 0;
    FOR check_views IN
        SELECT definition, schemaname, viewname
        FROM pg_catalog.pg_views
    LOOP
        EXECUTE check_views.definition;
    END LOOP;
    IF view_failures > 0 THEN
        RAISE NOTICE '% VIEWS FAILED', view_failures
        USING HINT = 'You must update these views before
continuing the transformation script.';
    END IF;
END;
```

SCHARP at FRED HUTCH

Handling Errors

- The problem is what happens when the view fails to execute.
- There are bugs in certain versions of PostgreSQL that the view definition will not execute even though the view works. I found these bugs and they are fixed in the latest versions of PostgreSQL.

```
…
LOOP
  BEGIN
    --SET ROLE finance;
    EXECUTE check_views.definition;
    --RAISE NOTICE 'View Passed: %.%',
check_views.schemaname, check_views.viewname;
  EXCEPTION
    WHEN invalid_schema_name THEN
      view_failures = view_failures + 1;
      RAISE NOTICE 'View Failed - Invalid Schema Name:
%.%', check_views.schemaname, check_views.viewname;
    WHEN OTHERS THEN
      view_failures = view_failures + 1;
      RAISE NOTICE E'View Failed: %.%\nFailure Reason: % -
%', check_views.schemaname, check_views.viewname,
SQLSTATE, SQLERRM;
  END;
  --RESET ROLE;
END LOOP;
…
```

SCHARP at FRED HUTCH

Counting all the rows for all tables

Usable with SELECT permissions only

Setting up the Inline Function

- Here we setup the basic inline function.
- We will need the following variables:
  - r for looping through a list of the tables.
  - insert_sql_code for the base INSERT statement and we will add our values to for each row of data.
  - sql_code as a reusable variable for our SQL statements that we are going to execute.
  - temp_table for the name of our temp table. This is so you can easily change the name of the table.

```
DO $$
DECLARE
  r record;
  insert_sql_code TEXT;
  sql_code TEXT;
  temp_table TEXT;
BEGIN
…
END LOOP;
END $$;
```

SCHARP at FRED HUTCH

The Body of the Function

- We setup some basic variables at the start of the function.
- Truncate the temp table in case you run this function more than once on the same connection.
- Loop through the tables, excluding foreign tables, views, temp tables, system tables, etc.
- Generate the INSERT INTO and count as a single statement and the execute the statement.

```
…
temp_table := 'xxxx';
insert_sql_code := 'INSERT INTO ' ||
  quote_ident(temp_table) || ' ';
sql_code := 'CREATE TEMP TABLE IF NOT EXISTS ' ||
  quote_ident(temp_table) || ' (
  table_catalog information_schema.sql_identifier,
  table_schema information_schema.sql_identifier,
  table_name information_schema.sql_identifier,
  "count" bigint)';

EXECUTE sql_code;
sql_code := 'TRUNCATE TABLE ' ||
  quote_ident(temp_table);
EXECUTE sql_code;

FOR r IN SELECT * FROM information_schema.tables
  WHERE table_schema NOT IN ('pg_catalog',
'information_schema')
  AND table_schema NOT LIKE 'pg_temp%'
  AND table_type = 'BASE TABLE'
LOOP
  sql_code := insert_sql_code || 'SELECT ' ||
quote_literal(r.table_catalog) || ', ' ||
quote_literal(r.table_schema) || ', ' ||
quote_literal(r.table_name) || ', count(*) FROM ' ||
quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name);
  EXECUTE sql_code;
END LOOP;
…
```

SCHARP at FRED HUTCH

Viewing the Results

- Now we can just query the temp table for a list of the tables and their row count.

```
SELECT * FROM xxxx ORDER BY table_schema, table_name;
```

Debugging your Functions

Text Here

Write code in sections

- Write code in sections and setting a variable to let you know what section the code was in when it fails.
- Turn on/off debugging that is helpful for your debugging.

```
DECLARE
  error_stage TEXT;
  debug_on BOOLEAN;
BEGIN
  debug_on := TRUE;
  error_stage := 'Counting Duplicate Variables';
  IF debug_on THEN
    RAISE NOTICE 'Counting Duplicate Variables';
  END IF;
```

SCHARP at FRED HUTCH

Catching Errors

- When you don’t know what error you need to catch, use the WHEN OTHERS.
- Don’t ignore any errors within the error handler, if possible.

```
EXCEPTION WHEN OTHERS THEN

-- End Time
end_time := clock_timestamp();

BEGIN
  UPDATE status.plate SET
    plate_data_load_start_time = now(),
    up_to_date = FALSE
    WHERE plate.study_id = study_number
      AND plate.plate_number = plate_no;
EXCEPTION WHEN OTHERS THEN
-- Skip erroring when already inside the 1st error handler
END;

-- Return status
RETURN QUERY VALUES (df_study_no, plate_no, NULL::BIGINT,
NULL::BIGINT, NULL::BIGINT, NULL::BIGINT, NULL::BIGINT,
NULL::BIGINT, NULL::BIGINT, NULL::BIGINT, FALSE, SQLSTATE,
SQLERRM, error_stage,  end_time, (end_time - start_time),
file_path_name);
```

SCHARP at FRED HUTCH

Write an error log

- Write all your errors to a table with in the function or via the return on the function.

```
INSERT INTO tools.load_status
SELECT * FROM tools.load_from_file('study202.plate511.20140123.1221.csv');
```

SCHARP at FRED HUTCH

Temp vers Unlogged

- Temp tables are great for speeding of large queries but bad for tuning queries inside of functions.
- Unlogged tables are great for debugging. They are left behind once your function ends and you can then run explain queries against them.
  - They are not in the WAL files and so are not streaming replicated.
  - If your server crashes and restarts, they are dropped.
  - A perfect solution in place of temp tables while debugging and then change to temp for production.

SCHARP at FRED HUTCH

Index's

- Don’t forget to add Index’s, Primary Keys, Unique Keys, etc to your Temp/Unlogged tables.
- You need these for optimizing the use of these tables in your queries, just like with normal tables.

Transactions

- A function is a single transaction, by default.
- You may create sub transactions within your function if you wish.
- If you wrap your function in an outer function. Everything is covered by the outer functions transaction. This means if your outer function calls the inner function 5 times and you cancel it on the 4th time, no data from any of the completed inner functions will show up.

SCHARP at FRED HUTCH

Dynamic Queries

- When you need to make dynamic queries, it is best to save them into a variable, so you may export them via RAISE NOTICE.

```
DECLARE
  sql TEXT;
  degug_on BOOLEAN;
BEGIN
  debug_on := TRUE;
  sql := 'CREATE TABLE x AS SELECT * FROM pg_stat_activity;';
  IF debug_on THEN
    RAISE NOTICE 'Create Query: %', sql;
  END IF;
  EXECUTE sql;
```

SCHARP at FRED HUTCH

Number of rows affected

- Sometimes you may want to log how may rows are being inserted, updated and/or deleted within your function.

```
DECLARE
  sql TEXT;
  num_of_rows bigint;
BEGIN
  sql := 'CREATE TABLE x AS SELECT * FROM pg_stat_activity;';
  EXECUTE sql;
  GET DIAGNOSTICS num_of_rows = ROW_COUNT;
```

SCHARP at FRED HUTCH

Hiding Auto Notices

- For some actions, there are notices raised. You may hide these by changing the client_min_messages around that specific line of code.
- DROP TABLE IF EXISTS is one that I like to wrap in this style.

```
insert_plate_record_sql := 'ALTER TABLE
  temp_current_variable_change_' || df_study_no || '_' ||
plate_no || '
  ADD CONSTRAINT temp_current_variable_change_' ||
df_study_no || '_' || plate_no || '_idx
  PRIMARY KEY (df_study_number, plate_number,
variable_number, ptid, visit_number);';

SET client_min_messages = WARNING;
EXECUTE insert_plate_record_sql;
RESET client_min_messages;
```

SCHARP at FRED HUTCH

Code time?

- How long does each section of code take?
- Add a clock_timestamp().

```
IF debug_on THEN
  RAISE NOTICE 'Generating Temp Data: %',
clock_timestamp();
END IF;

IF debug_on THEN
  RAISE NOTICE 'Finished Generating Temp Data: %',
clock_timestamp();
END IF;
```

SCHARP at FRED HUTCH

Functions calling Functions

- While you might be tempted to split large functions into a lot of small functions for debugging purposes, don’t.
- Every time you call a function there is a start up overhead. This is especially bad for plperl and other non-sql languages.
- Each sub-function becomes it’s own sub-transaction.
- Compounds logging issues.

THANK YOU

SCHARP
at FRED HUTCH