PL/R – The Fast Path to Advanced Analytics

Joe Conway
joe.conway@credativ.com
mail@joeconway.com

credativ Group

September 28, 2011

Intro to PL/R

What is PL/R?

- R Procedural Language for PostgreSQL. Enables user-defined SQL functions to be written in the R language

What is R?

- R is an open source (GPL) language and environment for statistical computing and graphics. R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible.

`http://www.r-project.org`

`http://www.joeconway.com/plr`

Pros

- Leverage people's knowledge and skills
  - statistics/math is a specialty
- Leverage hardware
  - server better able to handle analysis of large datasets
- Processing/bandwidth efficiency
  - why send large datasets across the network?
- Consistency of analysis
  - ensure analysis done consistently once vetted
- Abstraction of complexity
  - keep system understandable and maintainable
- Leverage R
  - rich core functionality and huge ecosystem

Cons

- PostgreSQL user
  - Slower than standard SQL aggregates and PostgreSQL functions for simple cases
  - New language to learn
- R user
  - Debugging more challenging than working directly in R
  - Less flexible for ad hoc analysis

Installing PL/R

- Installation from source: general steps

```
tar -xzf plr-8.3.0.12.tar.gz
cd plr/
USE_PGXS=1 make
su -c "USE_PGXS=1 make install"
make installcheck
```

- References:

http://www.joeconway.com/plr/
http://www.joeconway.com/plr/doc/plr-install.html

Installing PL/R from Binary

- RPM: `http://yum.postgresql.org/news-packagelist.php`
- Win32, Win64: `http://www.joeconway.com/plr`
- Debian:

```
aptitude search '~n plr'
```

- CentOS:

```
yum list R-\*
No Matching Packages
```

- Fedora:

```
yum list R-\*
No Matching Packages
```

Installing PL/R

- Language Installation into Database
  - Using plr.sql

    ```
    psql mydatabase < plr.sql
    ```

  - Manually

    ```
    CREATE FUNCTION plr_call_handler()
    RETURNS LANGUAGE_HANDLER
    AS '$libdir/plr' LANGUAGE C;

    CREATE LANGUAGE plr HANDLER plr_call_handler;
    ```

  - As of PG 9.1: CREATE EXTENSION

    ```
    CREATE EXTENSION plr;
    -- when/if no longer wanted
    DROP EXTENSION plr;
    ```

Creating PL/R Functions

- Similar to other PostgreSQL PLs

```
CREATE OR REPLACE FUNCTION func_name(arg-type1 [, arg-type2 ...])
RETURNS return-type AS $$
    function body referencing arg1 [, arg2 ...]
$$ LANGUAGE 'plr';

CREATE OR REPLACE FUNCTION func_name(myarg1 arg-type1
                                     [, myarg2 arg-type2 ...])
RETURNS return-type AS $$
    function body referencing myarg1 [, myarg2 ...]
$$ LANGUAGE 'plr';
```

- But a little different from standard R functions

```
func_name <- function(myarg1 [,myarg2...]) {
  function body referencing myarg1 [, myarg2 ...]
}
```

Creating PL/R Functions

An alternative method may be used to create a function in PL/R, if certain criteria are met

- Must be a simple call to an existing R function
- Name used for the PL/R function must match that of the R function exactly
- Function may be defined with no body, and the arguments will be passed directly to the R function of the same name

```
CREATE OR REPLACE FUNCTION sd(float8[])
RETURNS float AS '' LANGUAGE 'plr';

SELECT round(sd(ARRAY[1.23,1.31,1.42,1.27])::numeric, 8);
   round
------------
 0.08180261
(1 row)
```

Argument Type Conversions

- Arguments may be explicitly named when creating a function
- Otherwise argument values are passed as variables arg1 ... argN to the R script
- Define function STRICT to avoid thinking about NULL input values
- In a non-strict function, if the actual value of an argument is NULL, the corresponding argN variable will be set to a NULL R object

Argument Type Conversions

PostgreSQL Type	R Type
int2, int4	integer
int8, float4, float8, cash, numeric	numeric
bytea	object
everything else	character

- One-dimensional PostgreSQL arrays: converted to multi-element R vectors
  - Pass-by-value integer and numeric types special-cased for performance
- Two-dimensional PostgreSQL arrays: mapped to R matrixes
- Three-dimensional PostgreSQL arrays: converted to three-dimensional R arrays.
- Composite-types are transformed into R data.frames

Return Type Conversions

- Return values are special-cased for performance if
  - R data type is Integer or Real
  - PostgreSQL type is 1D array of pass-by-value numeric
  - no NULL/NA elements
- Scalar bytea return values are also special-cased
  - R object being returned is serialized
  - Binary result directly mapped into PostgreSQL bytea
- Otherwise return values are first coerced to R character
  - If resulting string is acceptable for PostgreSQL return type, will produce a result

Return Type Conversions

- To return a NULL value from a PL/R function, return NULL

```
CREATE OR REPLACE FUNCTION r_max (integer, integer)
RETURNS integer AS $$
    if (is.null(arg1) && is.null(arg2))
        return(NULL)
    if (is.null(arg1))
        return(arg2)
    if (is.null(arg2))
        return(arg1)
    if (arg1 > arg2)
       return(arg1)
    arg2
$$ LANGUAGE 'plr';
```

Return Type Conversions

- Data type similar to arguments
- Mapping between the dimensionality of the declared PostgreSQL return type and the type of R object
- Depends on both R object dimensions as well declared PostgreSQL dimensions (i.e. scalar, array, composite type)
  - if return value in PL/R function is a data.frame, and Postgres return type is setof composite, the data frame is returned as rows and columns
  - if R = 1, 2, or 3D array, and Postgres = array, then return is array

Return Type Conversions

PgSQL return type	R type	Result
scalar	array, matrix, vector	first column of first row
setof scalar	1D array, greater than 2D array, vector	multi-row, 1 column set
scalar	data.frame	textual representation of the first column's vector
setof scalar	2D array, matrix, data.frame	#columns ¿ 1, error #columns == 1, multi-row, 1 column set
array	1D array, greater than 3D array, vector	1D array
array	2D array, matrix, data.frame	2D array
array	3D array	3D array
composite	1D array, greater than 2D array, vector	first row, 1 column
setof composite	1D array, greater than 2D array, vector	multi-row, 1 column set
composite	2D array, matrix, data.frame	first row, multi-column
setof composite	2D array, matrix, data.frame	multi-row, multi-column set

SQL Queries

```
pg.spi.exec(character query)
```

- Execute SQL query given as a string
- Error in the query causes an error to be raised
- Returns number of rows processed for INSERT, UPDATE, or DELETE statements
- Returns zero if the query is a utility statement
- SELECT statement: values of selected columns placed in data.frame with tgt column names as frame column names
- Non-numeric columns are not converted to R "factors" (but pg.spi.factor is provided)

SQL Queries

```
CREATE OR REPLACE FUNCTION test_spi_tup(text)
RETURNS SETOF RECORD AS $$
  pg.spi.exec(arg1)
$$ language 'plr';

SELECT * FROM test_spi_tup($$
  SELECT oid,
         NULL::text as nullcol,
         typname
  FROM pg_type
  WHERE typname = 'oid'
  OR typname = 'text'
$$)
AS t(typeid oid, nullcol text, typename name);
 typeid | nullcol | typename
--------+---------+----------
     25 |         | text
     26 |         | oid
(2 rows)
```

Prepared SQL

```
load_r_typenames()
pg.spi.prepare(character query, integer vector type_vector)
pg.spi.execp(external pointer saved_plan, variable listvalue_list)
```

- load_r_typenames() used to make predefined PostgreSQL data type global variables available
- pg.spi.prepare() prepares and saves a query plan for later execution
- pg.spi.execp() executes previously prepared query
- saved_plan is the external pointer returned by pg.spi.prepare
- If query references arguments, value_list must be supplied: this is an R list of actual values for the plan arguments
  - Must be the same length as the argument type_vector previously given to pg.spi.prepare
  - Pass NA for value_list if the query has no arguments

Prepared SQL

```
SELECT load_r_typenames();

CREATE OR REPLACE FUNCTION test_spi_prep(text)
RETURNS TEXT AS $$
  sp <<- pg.spi.prepare(arg1, c(NAMEOID, NAMEOID));
  print("OK");
$$ language 'plr';

SELECT test_spi_prep('SELECT oid, typname
                      FROM pg_type
                      WHERE typname = $1 OR typname = $2');

CREATE OR REPLACE FUNCTION test_spi_execp(text, text, text)
RETURNS SETOF RECORD AS $$
  pg.spi.execp(pg.reval(arg1), list(arg2,arg3))
$$ language 'plr';
```

Prepared SQL (cont.)

```
SELECT * FROM test_spi_execp('sp','oid','text')
AS t(typeid oid, typename name);
 typeid | typename
--------+----------
     25 | text
     26 | oid
(2 rows)
```

Cursors

```
pg.spi.cursor_open(character cursor_name,
                   external pointer saved_plan,
                   variable list value_list)
pg.spi.cursor_fetch(external pointer cursor,
                    boolean forward, integer rows)
pg.spi.cursor_close(external pointercursor)
```

- pg.spi.cursor_open() opens a cursor identified by cursor_name, used to scroll through the results of query plan previously prepared by pg.spi.prepare
- pg.spi.cursor_fetch() fetches rows from the cursor object
- pg.spi.cursor_close() closes previously opened cursor

Utility

```
pg.quoteliteral(character SQL_string)
pg.quoteident(character SQL_string)
pg.thrownotice(character message)
pg.throwerror(character message)
pg.spi.factor(data.frame data)
```

- pg.quoteliteral() safely quotes string literals
- pg.quoteident () quotes string to be used as an identifier
- pg.thrownotice() and pg.throwerror() emit PostgreSQL NOTICE or ERROR message
- pg.spi.factor() accepts an R data.frame as input, and converts all non-numeric columns to factors

RPostgreSQL Compatibility

```
dbDriver(character dvr_name)
dbConnect(DBIDriver drv, character user, character password,
          character host, character dbname, character port,
          character tty, character options)
dbSendQuery(DBIConnection conn, character sql)
fetch(DBIResult rs, integer num_rows)
dbClearResult (DBIResult rs)
dbGetQuery(DBIConnection conn, character sql)
dbReadTable(DBIConnection conn, character name)
dbDisconnect(DBIConnection conn)
dbUnloadDriver(DBIDriver drv)
```

- Allows prototyping using R, move to PL/R for production
- Queries performed in current database
- Driver/connection parameters ignored; dbDriver, dbConnect, dbDisconnect, and dbUnloadDriver are no-ops

RPostgreSQL Compatibility Example

- PostgreSQL access from R

```
require(TSP)
require(fields)
require(RPostgreSQL)

drv <- dbDriver("PostgreSQL")
conn <- dbConnect(drv, user="postgres", dbname="pgissc")
sql.str <- "select id, st_x(location) as x, st_y(location) as y, location fro
waypts <- dbGetQuery(conn, sql.str)
dist.matrix <-  rdist.earth(waypts[,2:3], R=3949.0)
rtsp <- TSP(dist.matrix)
soln <- solve_TSP(rtsp)
dbDisconnect(conn)
dbUnloadDriver(drv)

print(paste("tour.dist=", attributes(soln)$tour_length))
```

RPostgreSQL Compatibility Example

- Same function from PL/R

```
CREATE OR REPLACE FUNCTION tsp_tour_length() RETURNS float8 AS $$
  require(TSP)
  require(fields)
  require(RPostgreSQL)

  drv <- dbDriver("PostgreSQL")
  conn <- dbConnect(drv, user="postgres", dbname="pgissc")
  sql.str <- "select id, st_x(location) as x, st_y(location) as y, location fro
  waypts <- dbGetQuery(conn, sql.str)
  dist.matrix <-  rdist.earth(waypts[,2:3], R=3949.0)
  rtsp <- TSP(dist.matrix)
  soln <- solve_TSP(rtsp)
  dbDisconnect(conn)
  dbUnloadDriver(drv)

  return(attributes(soln)$tour_length)
$$ LANGUAGE 'plr' STRICT;
```

RPostgreSQL Compatibility Example (cont.)

- Output from R

```
[1] "tour.dist= 2804.58129355858"
```

- Same function from PL/R

```
SELECT tsp_tour_length();
 tsp_tour_length
------------------
 2804.58129355858
(1 row)
```

State Variable

- Global R variable called pg.state.firstpass
- TRUE first time PL/R function called for particular query
- On subsequent calls value is left unchanged
- Allows PL/R function to perform expensive initialization on the first call, reuse the results for the remaining rows

State Variable - Example

```
CREATE TABLE t (f1 int); INSERT INTO t VALUES (1),(2),(3);
CREATE OR REPLACE FUNCTION state(INT) RETURNS INT AS $$
  if (pg.state.firstpass == TRUE)
  {pg.state.firstpass <<- FALSE; Sys.sleep(10); return(arg1)}
  else {return(arg1)}
$$ LANGUAGE plr;

\timing
SELECT f1, state(f1) FROM t;
 f1 | state
----+-------
  1 |     1
  2 |     2
  3 |     3
(3 rows)
Time: 10003.472 ms
```

Preloading PL/R Shared Object

- postgresql.conf variable `shared_preload_libraries` specifies one or more shared libraries to be preloaded and initialized at server start
- If more than one library is to be loaded, separate their names with commas
- This parameter can only be set at server start
- Library startup time is avoided when the library is first used
- On Windows hosts, preloading a library at server start will not reduce startup time
- If specified library not found, the server will fail to start

```
shared_preload_libraries = '$libdir/plr'
```

Auto-loading R code

- Special table, `plr_modules`, presumed to contain R functions
- If table exists, functions fetched and loaded into R interpreter on initialization
- `plr_modules` defined as follows

```
CREATE TABLE plr_modules (modseq int4,
                          modsrc text);
```

- `modseq` used to control order of installation
- `modsrc` contains text of R code to be executed
- `plr_modules` must be readable by all, but it is wise to make it owned and writable only by the database administrator
- Use `reload_plr_modules()` to force re-loading `plr_modules`

Auto-loading R code - Example

EXAMPLE

- Create R function named pg.test.module.load on initialization
- PL/R function may now simply reference the function directly

```
INSERT INTO plr_modules
  VALUES (0, 'pg.test.module.load <-function(msg) {print(msg)}');
SELECT reload_plr_modules();

CREATE OR REPLACE FUNCTION pg_test_module_load(TEXT) RETURNS TEXT AS $$
  pg.test.module.load(arg1)
$$ language 'plr';

SELECT pg_test_module_load('hello world');
 pg_test_module_load
---------------------
 hello world
(1 row)
```

Interactively Loading R Code

```
install_rcmd(text R_code)
```

- `install_rcmd()` installs R code, given as a string, into the interpreter
- Global status data held between calls or shared between different PL/R functions
- Persists for the duration of the SQL client connection

Interactively Loading R Code - Example

```
SELECT install_rcmd('pg.test.install <-function(msg) {print(msg)}');

CREATE OR REPLACE FUNCTION pg_test_install(TEXT) RETURNS TEXT AS $$
  pg.test.install(arg1)
$$ language 'plr';

SELECT pg_test_install('hello world');
 pg_test_install
-----------------
 hello world
(1 row)
```

Array

```
plr_singleton_array(float8 first_element)
plr_array_push(float8[] array, float8 next_element)
plr_array_accum(float8[] state_value, float8 next_element)
```

- `plr_singleton_array()` creates a new PostgreSQL array using `first_element`
- `plr_array_push()` pushes a new element onto the end of an existing PostgreSQL array
- `plr_array_accum()` creates new array using `next_element` if `state_value` is NULL, otherwise, pushes `next_element` onto the end of `state_value`
- Redundant with built in functionality of recent PGSQL

Array Example

EXAMPLE

```
CREATE OR REPLACE FUNCTION array_accum (int[], int)
RETURNS int[]
AS '$libdir/plr','plr_array_accum'
LANGUAGE 'C';

SELECT array_accum(NULL, 42);
 array_accum
-------------
 {42}
(1 row)
SELECT array_accum(ARRAY[23,35], 42);
 array_accum
-----------------
 {23,35,42}
(1 row)
```

Utility

```
plr_version()
load_r_typenames()
r_typenames()
plr_set_display(text display)
plr_get_raw(bytea serialized_object)
```

- `plr_version()` displays PL/R version as a text string
- `load_r_typenames()` installs datatype Oid variables into the R interpreter as globals
- `r_typenames()` displays the datatype Oid variables
- `plr_set_display()` sets the DISPLAY environment vaiable under which the Postmaster is currently running
- `plr_get_raw()` unserializes R object and returns the pure raw bytes – e.g. JPEG or PNG graphic

Environment

`plr_environ()`

- `plr_environ()` displays environment under which the Postmaster is currently running
- Useful to debug issues related to R specific environment variables
- Installed with `EXECUTE` permission revoked from `PUBLIC`

```
SELECT * FROM plr_environ() WHERE name = 'PGDATA';
  name   |                  value
---------+-----------------------------------
 PGDATA | /usr/local/pgsql-REL8_4_STABLE/data
(1 row)
```

Aggregates

- Aggregates in PostgreSQL are extensible via SQL commands
- State transition function and possibly a final function are specified
- Initial condition for state function may also be specified

Aggregates Example

```
CREATE OR REPLACE FUNCTION r_median(ANYARRAY) RETURNS ANYELEMENT AS $$
  median(arg1)
$$ LANGUAGE 'plr';

CREATE AGGREGATE median (ANYELEMENT) (
  sfunc = array_append,
  stype = anyarray,
  finalfunc = r_median,
  initcond = '{}');
```

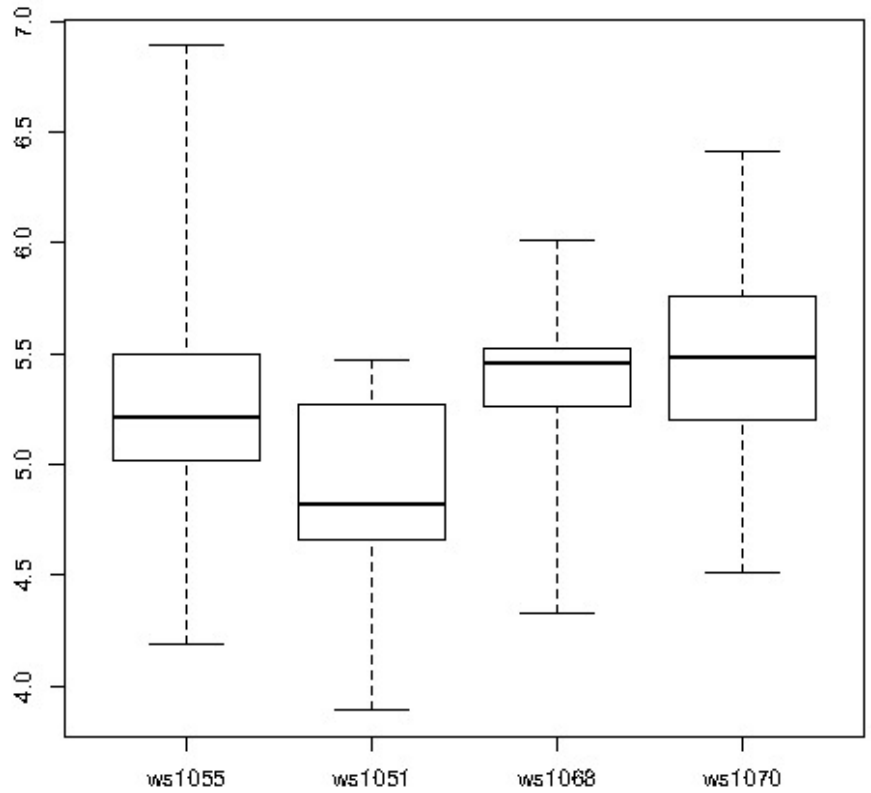
Aggregates Example (cont.)

```
CREATE TABLE FOO(f0 int, f1 text, f2 float8);
INSERT INTO foo VALUES (1,'cat1',1.21), (2,'cat1',1.24), (3,'cat1',1.18),
                       (4,'cat1',1.26), (5,'cat1',1.15), (6,'cat2',1.15),
                       (7,'cat2',1.26), (8,'cat2',1.32), (9,'cat2',1.30);

SELECT f1, median(f2) FROM foo GROUP BY f1 ORDER BY f1;
  f1   | median
------+--------
 cat1 |   1.21
 cat2 |   1.28
(2 rows)
```

Aggregates Example #2

```
CREATE OR REPLACE FUNCTION r_quartile(anyarray) RETURNS anyarray AS $$
  quantile(arg1, probs = seq(0, 1, 0.25), names = FALSE)
$$ LANGUAGE 'plr';

CREATE AGGREGATE quartile (ANYELEMENT) (
  sfunc = array_append,
  stype = anyarray,
  finalfunc = r_quantile,
  initcond = '{}');

SELECT workstation, quartile(id_val) FROM sample_numeric_data
WHERE ia_id = 'G121XB8A' GROUP BY workstation;
 workstation |            quantile
-------------+--------------------------------
 1055        | {4.19,5.02,5.21,5.5,6.89}
 1051        | {3.89,4.66,4.825,5.2675,5.47}
 1068        | {4.33,5.2625,5.455,5.5275,6.01}
 1070        | {4.51,5.1975,5.485,5.7575,6.41}
(4 rows)
```

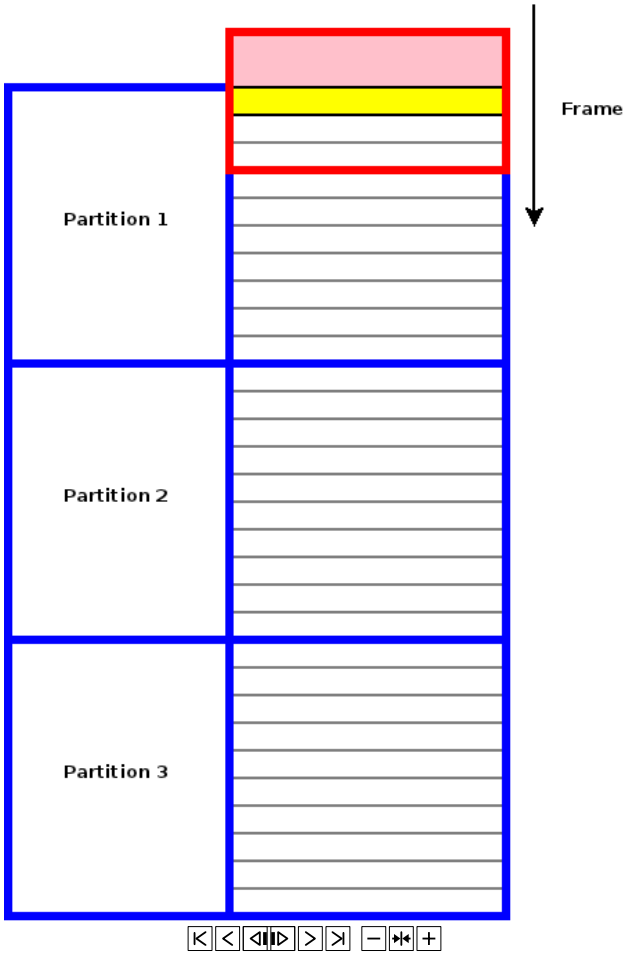
Aggregates Example - Quartile Boxplot Output



Window Functions

- Window Functions are available as of PostgreSQL 8.4
- Provide ability to calculate across sets of rows related to current row
- Similar to aggregate functions, but does not cause rows to become grouped
- Able to access more than just the current row of the query result

Window Functions



Window Function Auto-variables

- Variables automatically provided by PL/R to the R interpreter
- fargN: R vectors containing current row's data plus related rows in the frame
  - N corresponds to the function provided argument, e.g. first argument is 1, second is 2, etc.
  - Related rows are defined by frame clause, e.g. ROWS BETWEEN frame_start AND frame_end
- fnumrows: number of rows in current frame
- prownum: 1-based row offset of the current row in the current partition

Window Function Example

```
CREATE TABLE test_data
  (fyear integer,firm float8,eps float8);
INSERT INTO test_data
SELECT (b.f + 1) % 10 + 2000 AS fyear,
       floor((b.f+1)/10) + 50 AS firm,
       f::float8/100 + random()/10 AS eps
FROM generate_series(-500,499,1) b(f);

-- find slope of the linear model regression line
CREATE OR REPLACE FUNCTION r_regr_slope(float8, float8)
RETURNS float8 AS $BODY$
  slope <- NA
  y <- farg1
  x <- farg2
  if (fnumrows==9) try (slope <- lm(y ~ x)$coefficients[2])
  return(slope)
$BODY$ LANGUAGE plr WINDOW;
```

Window Function Example

```
SELECT *, r_regr_slope(eps, lag_eps) OVER w AS slope_R
FROM (SELECT firm AS f, fyear AS fyr, eps,
  lag(eps) OVER (PARTITION BY firm ORDER BY firm, fyear) AS lag_eps
FROM test_data) AS a WHERE eps IS NOT NULL
WINDOW w AS (PARTITION BY firm ORDER BY firm, fyear ROWS 8 PRECEDING);
 f |  fyr |       eps          |     lag_eps        |     slope_r
---+------+--------------------+--------------------+--------------------
 1 | 1991 | -4.99563754182309  |                    |
 1 | 1992 | -4.96425441872329  | -4.99563754182309  |
 1 | 1993 | -4.96906093481928  | -4.96425441872329  |
 1 | 1994 | -4.92376988714561  | -4.96906093481928  |
 1 | 1995 | -4.95884547665715  | -4.92376988714561  |
 1 | 1996 | -4.93236254784279  | -4.95884547665715  |
 1 | 1997 | -4.90775520844385  | -4.93236254784279  |
 1 | 1998 | -4.92082695348188  | -4.90775520844385  |
 1 | 1999 | -4.84991340579465  | -4.92082695348188  | 0.691850614092383
 1 | 2000 | -4.86000917562284  | -4.84991340579465  | 0.700526929134053
```

Window Function Example #2

```
-- The idea of Winsorizing is to return either the original value or,
-- if that value is outside certain bounds, a trimmed value.
CREATE OR REPLACE FUNCTION winsorize(float8, float8)
RETURNS float8 AS $BODY$
  library(psych)
  return(winsor(as.vector(farg1), arg2)[prownum])
$BODY$ LANGUAGE plr VOLATILE WINDOW;

SELECT fyear, eps, winsorize(eps, 0.1) OVER (PARTITION BY fyear) AS w_eps
FROM test_data ORDER BY fyear, eps;
 fyear |         eps          |        w_eps
-------+----------------------+----------------------
  1991 |    -4.99563754182309 |    -4.46270967368037
  1991 |    -4.81143716350198 |    -4.46270967368037
  1991 |    -4.73127805045806 |    -4.46270967368037
  1991 |    -4.60706958658993 |    -4.46270967368037
  1991 |    -4.50345986126922 |    -4.46270967368037
  1991 |    -4.45818187505938 |    -4.45818187505938
  1991 |    -4.37243791841902 |    -4.37243791841902
```

Triggers

- Triggers can be written in PL/R
- Function called as a trigger must have no arguments and return type `TRIGGER`
- NULL return silently suppresses the triggering operation for this row
- One row data.frame returned is inserted instead of the one given in pg.tg.new (BEFORE, FOR EACH ROW only)
- Info from trigger manager passed to PL/R function in variables

Trigger Variables

- pg.tg.name - name of the trigger
- pg.tg.relid - object ID of table invoking trigger
- pg.tg.relname - name of table invoking trigger
- pg.tg.when - BEFORE or AFTER (trigger type)
- pg.tg.level - ROW or STATEMENT (trigger type)
- pg.tg.op - INSERT, UPDATE, or DELETE
- pg.tg.new/pg.tg.old - NEW and OLD rows
- pg.tg.args - vector of arguments given in CREATE TRIGGER

Trigger Example

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$

    if (pg.tg.op == "INSERT")
    {
      retval <- pg.tg.new
      retval[pg.tg.args[1]] <- 0
    }
    if (pg.tg.op == "UPDATE")
    {
      retval <- pg.tg.new
      retval[pg.tg.args[1]] <- pg.tg.old[pg.tg.args[1]] + 1
    }
    if (pg.tg.op == "DELETE")
      retval <- pg.tg.old

    return(retval)
$$ LANGUAGE plr;
```

Trigger Example

```
CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount
BEFORE INSERT OR UPDATE ON mytab
FOR EACH ROW
EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Auditing Example

- Detecting Potential Fraud
  - Use Benford's law (also called first-digit law)
- Applies to data approximating geometric sequence
- Examples include, for example:
  - Sales figures
  - Census data
  - Medical claims
  - Expense reports
  - Energy savings

`http://en.wikipedia.org/wiki/Benford's_law`

Auditing Example

- California Energy Efficiency Program Data
- Create and populate table with investment cost data

```
CREATE TABLE open_emv_cost(value float8, district int);
COPY open_emv_cost
 FROM 'open-emv.cost.csv'
WITH delimiter ',';
```

`http://open-emv.com/data`

Auditing Example

- Create and Benford's Law function

```
CREATE TYPE benford_t AS (actual_mean float8,
                          n int,
                          expected_mean float8,
                          distortion float8,
                          z float8);

CREATE OR REPLACE FUNCTION benford(numarr float8[]) RETURNS benford_t AS $$
  xcoll <- function(x) {return ((10 * x) / (10 ^ (trunc(log10(x)))))}
  numarr <- numarr[numarr >= 10]
  numarr <- xcoll(numarr)
  actual_mean <- mean(numarr)
  n <- length(numarr)
  expected_mean <- (90 / (n * (10 ^ (1/n) - 1)))
  distortion<-((actual_mean - expected_mean) / expected_mean)
  z<-(distortion / sd(numarr))
  retval<-data.frame(actual_mean,n,expected_mean,distortion,z)
  return(retval)
$$ LANGUAGE plr;
```

Auditing Example

- Execute Benford's Law function

```
SELECT * FROM benford(array(SELECT value FROM open_emv_cost));
-[ RECORD 1 ]-+---------------------
actual_mean   | 38.1936561918275
n             | 240
expected_mean | 38.8993031865999
distortion    | -0.0181403505195804
z             | -0.00098403690808080443
```

- Data looks about right...

Geospatial Example

- Solve the famous Traveling Salesman Problem
  - Given list of location and distances, find a shortest possible tour that visits each location exactly once
- NP-hard problem in combinatorial optimization
- Applications include, for example:
  - Logistics
  - Land management
  - Semiconductor inspection
  - Geonome sequencing
  - Routing of SONET rings

`http://en.wikipedia.org/wiki/Travelling_salesman_problem`

`http://www.tsp.gatech.edu/apps/index.html`

Geospatial Example

- Create and populate table with locations

```
CREATE TABLE stands (id serial primary key,
                     strata integer not null,
                     initage integer);
SELECT AddGeometryColumn('','stands','boundary','4326','MULTIPOLYGON',2);
CREATE INDEX "stands_boundary_gist" ON "stands" USING gist ("boundary" gist_geometry_ops);
SELECT AddGeometryColumn('','stands','location','4326','POINT',2);
CREATE INDEX "stands_location_gist" ON "stands" USING gist ("location" gist_geometry_ops);

INSERT INTO stands (id,strata,initage,boundary,location) VALUES
 (1,1,1,GeometryFromText('MULTIPOLYGON(((59.250000 65.000000,55.000000 65.000000,55.000000 51.750000,
  60.735294 53.470588,  62.875000 57.750000, 59.250000 65.000000 )))', 4326),
  GeometryFromText('POINT( 61.000000 59.000000 )', 4326 ))
,(2,2,1,GeometryFromText('MULTIPOLYGON(((67.000000 65.000000,59.250000 65.000000,62.875000 57.750000,
  67.000000 60.500000, 67.000000 65.000000 )))', 4326),
  GeometryFromText('POINT( 63.000000 60.000000 )', 4326 ))
,(3,3,1,GeometryFromText('MULTIPOLYGON(((67.045455 52.681818,60.735294 53.470588,55.000000 51.750000,
  55.000000 45.000000,  65.125000 45.000000, 67.045455 52.681818 )))', 4326),
  GeometryFromText('POINT( 64.000000 49.000000 )', 4326 ))
;
```

Geospatial Example

- Create and populate table with locations

```
INSERT INTO stands (id,strata,initage,boundary,location) VALUES
 (4,4,1,GeometryFromText('MULTIPOLYGON(((71.500000 53.500000,70.357143 53.785714,67.045455 52.681818,
  65.125000 45.000000,  71.500000 45.000000, 71.500000 53.500000 )))', 4326),
  GeometryFromText('POINT( 68.000000 48.000000 )', 4326 ))
,(5,5,1,GeometryFromText('MULTIPOLYGON(((69.750000 65.000000,67.000000 65.000000,67.000000 60.500000,
  70.357143 53.785714,  71.500000 53.500000,  74.928571 54.642857, 69.750000 65.000000 )))', 4326),
  GeometryFromText('POINT( 71.000000 60.000000 )', 4326 ))
,(6,6,1,GeometryFromText('MULTIPOLYGON(((80.000000 65.000000,69.750000 65.000000,74.928571 54.642857,
  80.000000 55.423077, 80.000000 65.000000 )))', 4326),
  GeometryFromText('POINT( 73.000000 61.000000 )', 4326 ))
,(7,7,1,GeometryFromText('MULTIPOLYGON(((80.000000 55.423077,74.928571 54.642857,71.500000 53.500000,
  71.500000 45.000000,  80.000000 45.000000, 80.000000 55.423077 )))', 4326),
  GeometryFromText('POINT( 75.000000 48.000000 )', 4326 ))
,(8,8,1,GeometryFromText('MULTIPOLYGON(((67.000000 60.500000,62.875000 57.750000,60.735294 53.470588,
  67.045455 52.681818,  70.357143 53.785714, 67.000000 60.500000 )))', 4326),
  GeometryFromText('POINT( 65.000000 57.000000 )', 4326 ))
;
```

Geospatial Example

- Create result data type and `plr_modules`

```
CREATE TABLE events
(
  seqid int not null primary key, -- visit sequence #
  plotid int, -- original plot id
  bearing real, -- bearing to next waypoint
  distance real,  -- distance to next waypoint
  velocity real, -- velocity of travel, in nm/hr
  traveltime real, -- travel time to next event
  loitertime real, -- how long to hang out
  totaltraveldist real, -- cummulative distance
  totaltraveltime real -- cummulaative time
);

SELECT AddGeometryColumn('','events','location','4326','POINT',2);
CREATE INDEX "events_location_gist" ON "events"
             USING gist ("location" gist_geometry_ops);
CREATE TABLE plr_modules (modseq int4 primary key,
                          modsrc text);
```

Geospatial Example

- Create main PL/R function

```
CREATE OR REPLACE FUNCTION solve_tsp(makemap bool, mapname text)
RETURNS SETOF events AS $$
  require(TSP)
  require(fields)

  sql.str <- "select id, st_x(location) as x,
              st_y(location) as y, location from stands"
  waypts <- pg.spi.exec(sql.str)
  dist.matrix <-  rdist.earth(waypts[,2:3], R=3949.0)
  rtsp <- TSP(dist.matrix)
  soln <- solve_TSP(rtsp)
  tour <- as.vector(soln)
  pg.thrownotice( paste("tour.dist=", attributes(soln)$tour_length))
  route <- make.route(tour, waypts, dist.matrix)
  if (makemap) {make.map(tour, waypts, mapname)}

  return(route)
$$ LANGUAGE 'plr' STRICT;
```

Geospatial Example

- Install make.route() function

```
INSERT INTO plr_modules VALUES (0,
 $$ make.route <-function(tour, waypts, dist.matrix) {
    velocity <- 500.0
    starts <- tour[1:(length(tour))-1]
    stops <- tour[2:(length(tour))]
    dist.vect <- diag( as.matrix( dist.matrix )[starts,stops] )
    last.leg <- as.matrix( dist.matrix )[tour[length(tour)],tour[1]]
    dist.vect <- c(dist.vect, last.leg )
    delta.x <- diff( waypts[tour,]$x )
    delta.y <- diff( waypts[tour,]$y )
    bearings <- atan( delta.x/delta.y ) * 180 / pi
    bearings <- c(bearings,0)
    for( i in 1:(length(tour)-1) ) {
      if( delta.x[i] > 0.0 && delta.y[i] > 0.0 ) bearings[i] <- bearings[i]
      if( delta.x[i] > 0.0 && delta.y[i] < 0.0 ) bearings[i] <- 180.0 + bearings[i]
      if( delta.x[i] < 0.0 && delta.y[i] > 0.0 ) bearings[i] <- 360.0 + bearings[i]
      if( delta.x[i] < 0.0 && delta.y[i] < 0.0 ) bearings[i] <- 180 + bearings[i]
    }
    route <- data.frame(seq=1:length(tour), ptid=tour, bearing=bearings, dist.vect=dist.vect,
                        velocity=velocity, travel.time=dist.vect/velocity, loiter.time=0.5)
    route$total.travel.dist <- cumsum(route$dist.vect)
    route$total.travel.time <- cumsum(route$travel.time+route$loiter.time)
    route$location <- waypts[tour,]$location
    return(route)}$$);
```

Geospatial Example

- Install make.map() function

```
INSERT INTO plr_modules
  VALUES (1, $$
  make.map <-function(tour, waypts, mapname) {
    require(maps)

    jpeg(file=mapname, width = 480, height = 480, pointsize = 10, quality = 75)
    map('world2', xlim = c(20, 120), ylim=c(20,80) )
    map.axes()
    grid()
    arrows(waypts[tour[1:(length(tour)-1)],]$x, waypts[tour[1:(length(tour)-1)],]$y,
           waypts[tour[2:(length(tour))],]$x, waypts[tour[2:(length(tour))],]$y,
           angle=10, lwd=1, length=.15, col="red")

    points( waypts$x, waypts$y, pch=3, cex=2 )
    points( waypts$x, waypts$y, pch=20, cex=0.8 )

    text(  waypts$x+2, waypts$y+2, as.character( waypts$id  ), cex=0.8 )
    title( "TSP soln using PL/R" )
    dev.off()
  }$$
);
```

Geospatial Example

- Run the TSP function

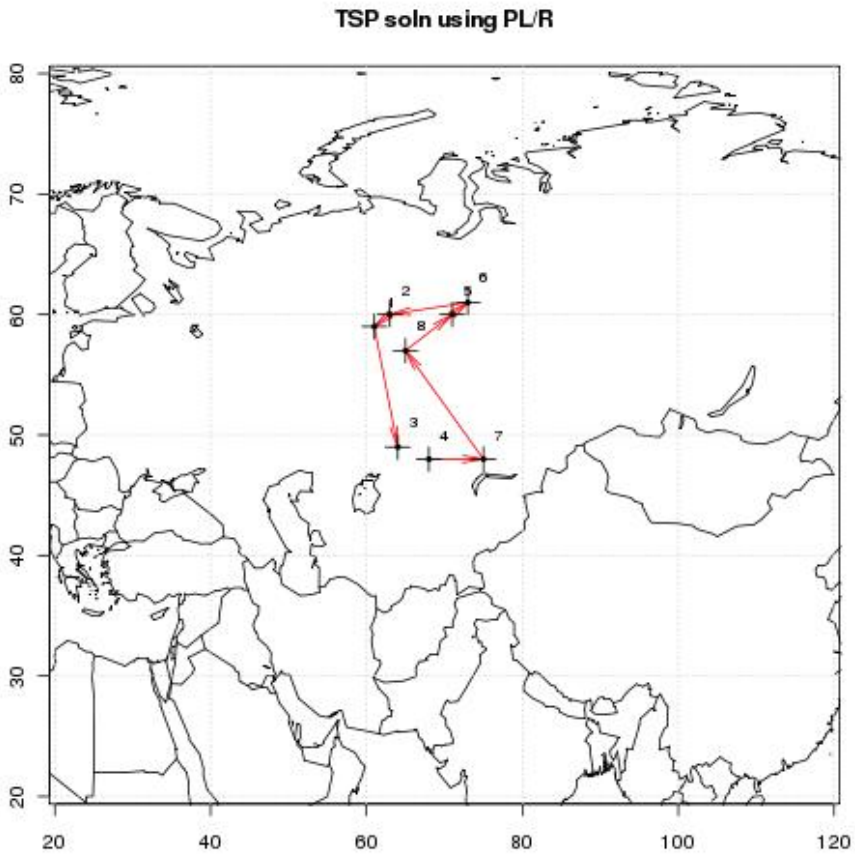
```
-- only needed if INSERT INTO plr_modules was in same session
SELECT reload_plr_modules();

SELECT seqid, plotid, bearing, distance, velocity, traveltime, loitertime, totaltraveldist
FROM solve_tsp(true, 'tsp.jpg');
NOTICE:  tour.dist= 2804.58129355858
 seqid | plotid | bearing | distance | velocity | traveltime | loitertime | totaltraveldist
-------+--------+---------+----------+----------+------------+------------+-----------------
     1 |      8 | 131.987 |  747.219 |      500 |    1.49444 |        0.5 |         747.219
     2 |      7 |     -90 |  322.719 |      500 |   0.645437 |        0.5 |         1069.94
     3 |      4 | 284.036 |  195.219 |      500 |   0.390438 |        0.5 |         1265.16
     4 |      3 | 343.301 |  699.683 |      500 |    1.39937 |        0.5 |         1964.84
     5 |      1 | 63.4349 |  98.2015 |      500 |   0.196403 |        0.5 |         2063.04
     6 |      2 | 84.2894 |  345.957 |      500 |   0.691915 |        0.5 |            2409
     7 |      6 | 243.435 |  96.7281 |      500 |   0.193456 |        0.5 |         2505.73
     8 |      5 |       0 |  298.855 |      500 |    0.59771 |        0.5 |         2804.58
(8 rows)
```

Geospatial Example

- Run the TSP function (first row expanded)

```
\x
SELECT * FROM solve_tsp(true, 'tsp.jpg');
NOTICE:  tour.dist= 2804.58129355858
-[ RECORD 1 ]---+-------------------------------------------------
seqid           | 1
plotid          | 3
bearing         | 104.036
distance        | 195.219
velocity        | 500
traveltime      | 0.390438
loitertime      | 0.5
totaltraveldist | 195.219
totaltraveltime | 0.890438
location        | 0101000020E6100000000000000005040000000000804840
-[ RECORD 2 ]---+-------------------------------------------------
[...]
```

Geospatial Example - Output



Stock Data Example

- get Hi-Low-Close data from Yahoo for any stock symbol
- plot with Bollinger Bands and volume
- requires extra R packages - from R:

```
install.packages(c('xts','Defaults','quantmod','cairoDevice','RGtk2'))
```

Stock Data Example

```
CREATE OR REPLACE FUNCTION plot_stock_data(sym text) RETURNS bytea AS $$
  library(quantmod)
  library(cairoDevice)
  library(RGtk2)

  pixmap <- gdkPixmapNew(w=500, h=500, depth=24)
  asCairoDevice(pixmap)

  getSymbols(c(sym))
  chartSeries(get(sym), name=sym, theme="white",
                TA="addVo();addBBands();addCCI()")

  plot_pixbuf <- gdkPixbufGetFromDrawable(NULL, pixmap,
                pixmap$getColormap(),0, 0, 0, 0, 500, 500)
  buffer <- gdkPixbufSaveToBufferv(plot_pixbuf, "jpeg",
                character(0),character(0))$buffer

  return(buffer)
$$ LANGUAGE plr;
```

Stock Data Example

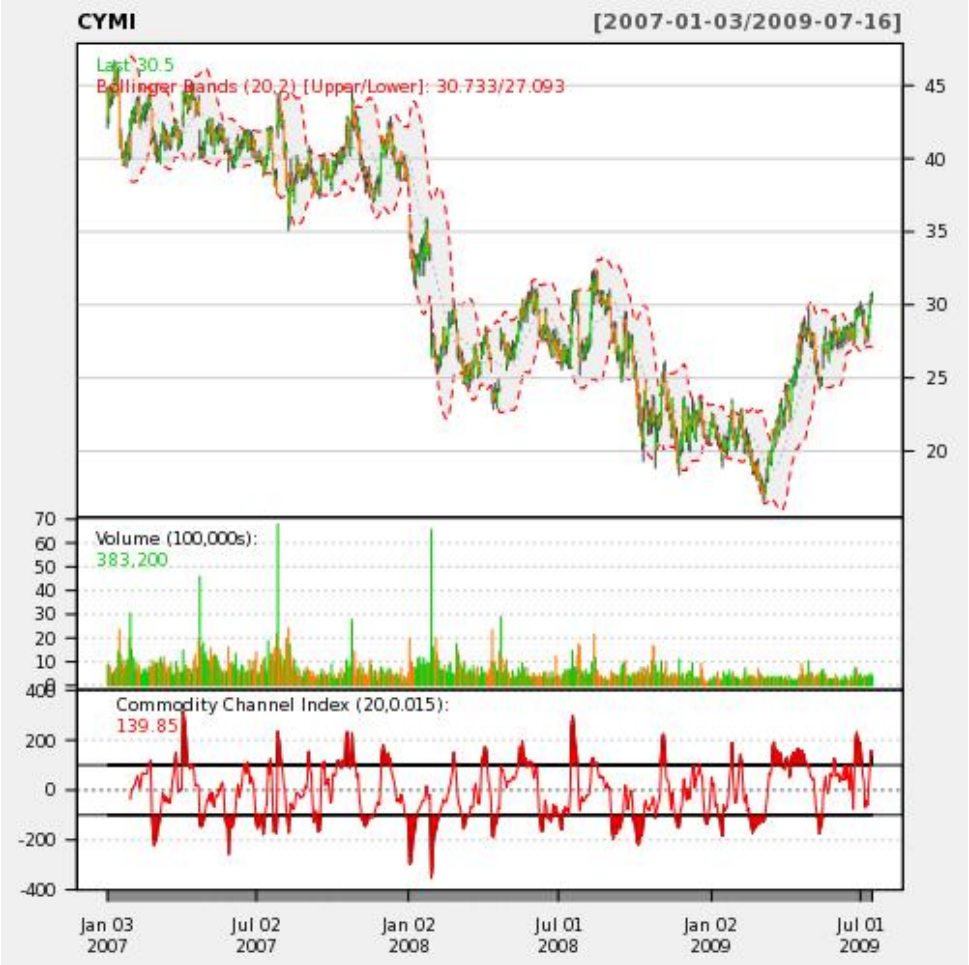
- Need screen buffer on typical server:

```
Xvfb :1 -screen 0 1024x768x24
export DISPLAY=:1.0
```

- Calling it from PHP for CYMI

```
<?php
$dbconn = pg_connect("...");
$rs = pg_query( $dbconn,
      "select plr_get_raw(plot_stock_data('CYMI'))");
$hexpic = pg_fetch_array($rs);
$cleandata = pg_unescape_bytea($hexpic[0]);

header("Content-Type: image/png");
header("Last-Modified: " .
       date("r", filectime($_SERVER['SCRIPT_FILENAME'])));
header("Content-Length: " . strlen($cleandata));
echo $cleandata;
?>
```

Stock Data Example - Output



Seismic Data Example

- Timeseries, waveform data
- Stored as array of floats recorded during seismic event at a constant sampling rate
- Available from online sources in individual file for each event
- Each file has about 16000 elements

Seismic Data Example

- Load 1000 seismic events (PL/pgSQL - 37 seconds)
- Store as arrays of float8

```
CREATE TABLE test_ts (dataid bigint NOT NULL PRIMARY KEY,
                      data double precision[]);
CREATE OR REPLACE FUNCTION load_test(int) RETURNS text AS $$
  DECLARE
    i    int;
    arr  text;
    sql  text;
  BEGIN
    arr := pg_read_file('array-data.csv', 0, 500000);
    FOR i IN 1..$1 LOOP
      sql := $i$INSERT INTO test_ts(dataid,data) VALUES ($i$ || i || $i$,'{$i$ || arr || $i$}')$i$;
      EXECUTE sql;
    END LOOP;
    RETURN 'OK';
  END;
$$ LANGUAGE plpgsql;

SELECT load_test(1000);
 load_test
-----------
 OK
(1 row)
Time: 37336.539 ms
```

Seismic Data Example

- Load 1000 seismic events (PL/R - 12 seconds)
- Store as R objects

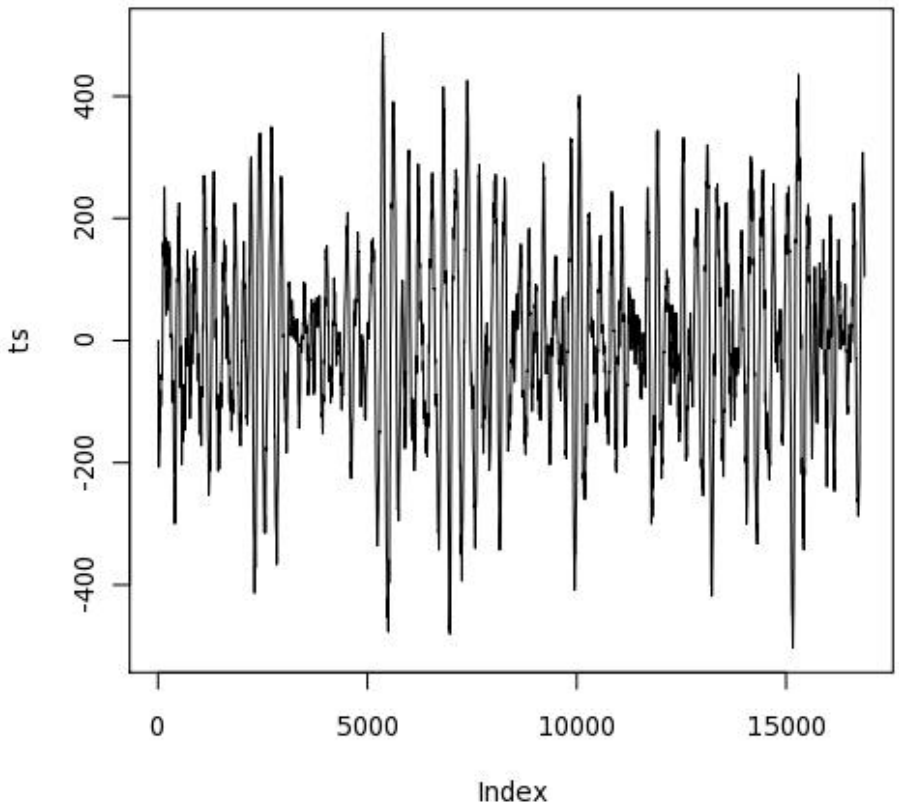
```
DROP TABLE IF EXISTS test_ts_obj;
CREATE TABLE test_ts_obj (
  dataid serial PRIMARY KEY,
  data bytea
);

CREATE OR REPLACE FUNCTION make_r_object(fname text) RETURNS bytea AS $$
  myvar<-scan(fname,sep=",")
  return(myvar);
$$ LANGUAGE 'plr' IMMUTABLE;

INSERT INTO test_ts_obj (data)
SELECT make_r_object('array-data.csv')
FROM generate_series(1,1000);

INSERT 0 1000
Time: 12166.137 ms
```

Seismic Data Example

- Plot the waveform

```
CREATE OR REPLACE FUNCTION plot_ts(ts double precision[]) RETURNS bytea AS $$
  library(cairoDevice)
  library(RGtk2)

  pixmap <- gdkPixmapNew(w=500, h=500, depth=24)
  asCairoDevice(pixmap)

  plot(ts,type="l")
  plot_pixbuf <- gdkPixbufGetFromDrawable(NULL, pixmap,
                                          pixmap$getColormap(),
                                          0, 0, 0, 0, 500, 500)
  buffer <- gdkPixbufSaveToBufferv(plot_pixbuf, "jpeg",
                                   character(0), character(0))$buffer
  return(buffer)
$$ LANGUAGE 'plr' IMMUTABLE;

SELECT plr_get_raw(plot_ts(data)) FROM test_ts WHERE dataid = 42;
```

Seismic Data Example - Waveform Output

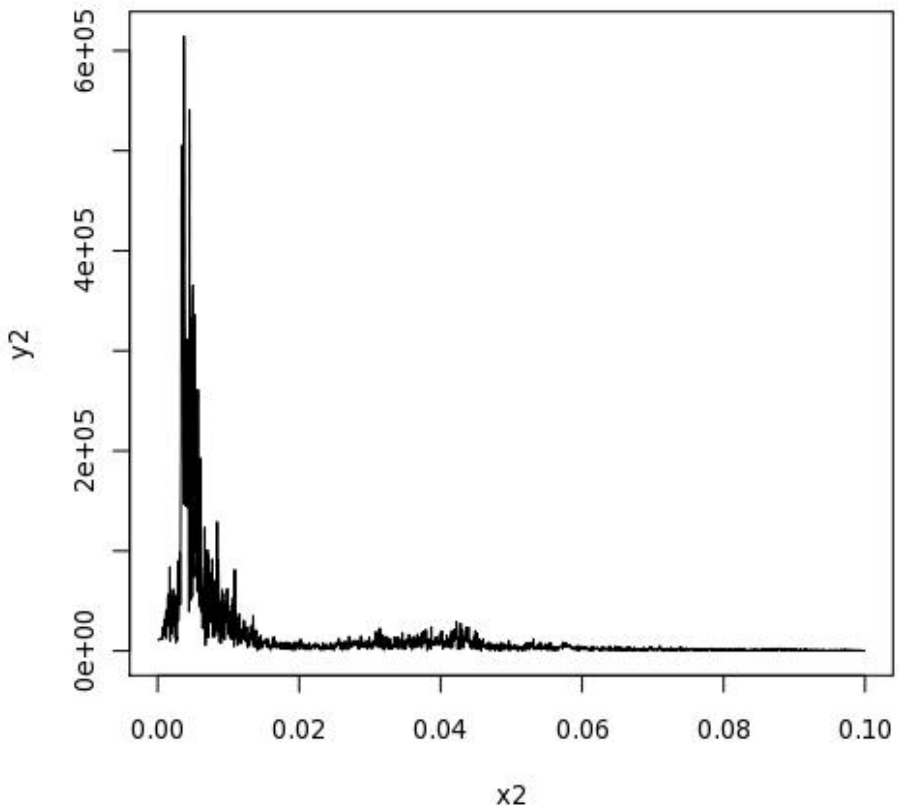


Seismic Data Example

- Analyze the waveform

```
CREATE OR REPLACE FUNCTION plot_fftps(ts bytea) RETURNS bytea AS $$
  library(cairoDevice)
  library(RGtk2)
  fourier<-fft(ts)
  magnitude<-Mod(fourier)
  y2 <- magnitude[1:(length(magnitude)/10)]
  x2 <- 1:length(y2)/length(magnitude)
  mydf <- data.frame(x2,y2)
  pixmap <- gdkPixmapNew(w=500, h=500, depth=24)
  asCairoDevice(pixmap)
  plot(mydf,type="l")
  plot_pixbuf <- gdkPixbufGetFromDrawable(NULL, pixmap,
                 pixmap$getColormap(), 0, 0, 0, 0, 500, 500)
  buffer <- gdkPixbufSaveToBufferv(plot_pixbuf, "jpeg",
                 character(0), character(0))$buffer
  return(buffer)
$$ LANGUAGE 'plr' IMMUTABLE;
SELECT plr_get_raw(plot_fftps(data)) FROM test_ts_obj WHERE dataid = 42;
```

Seismic Data Example - Waveform Analysis Output



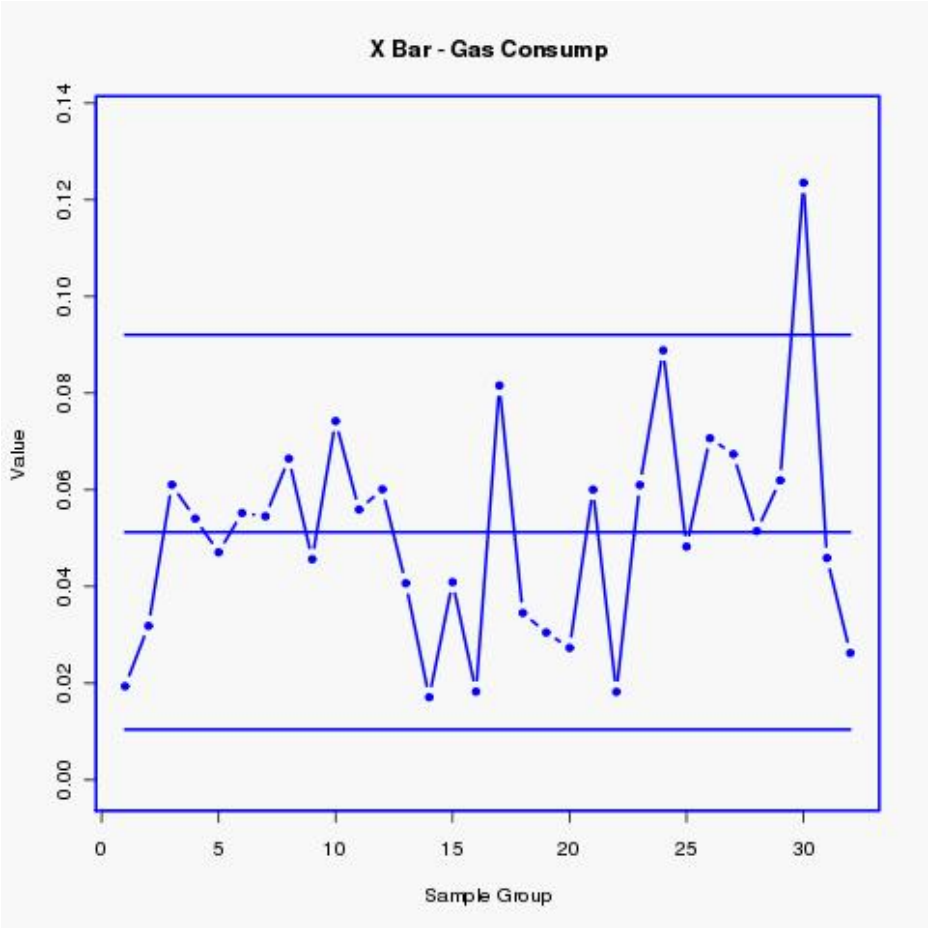
Statistical Process Control Example

- Named controlChart R function loaded via `plr_modules`; about 120 lines of code
- controlchart() PL/R function; another 130 lines of code

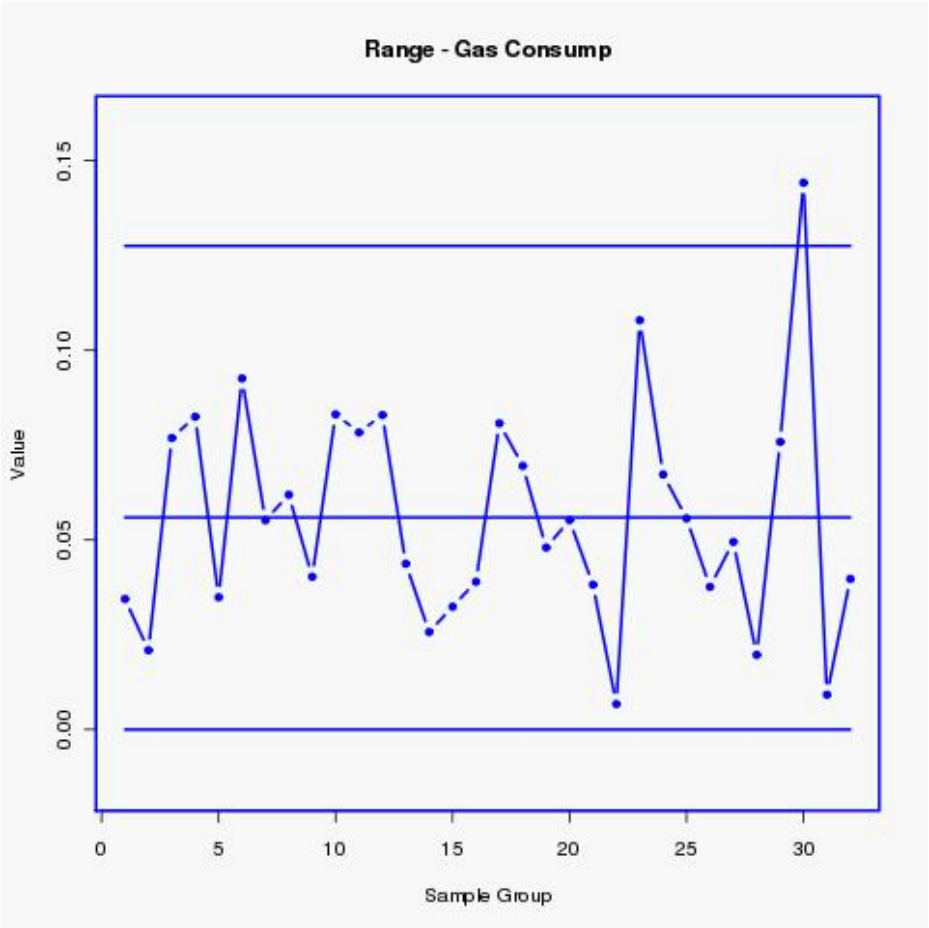
http://www.joeconway.com/source_code/controlchart.sql

```
SELECT * FROM controlchart('G121XA34', 3, 0, array['/tmp/xbar.jpg','/tmp/r.jpg'
SELECT * FROM controlchart('G121XA34', 3, 0, null) LIMIT 1;
-[ RECORD 1 ]-----------------
group_num | 1
xb        | 0.0193605889310595
xbb       | 0.0512444187147061
xucl      | 0.0920736498010521
xlcl      | 0.0104151876283601
r         | 0.0344209665807481
rb        | 0.0559304535429398
rucl      | 0.127521434077903
rlcl      | 0
gma       | 0.0193605889310595
Time: 21.986 ms
```

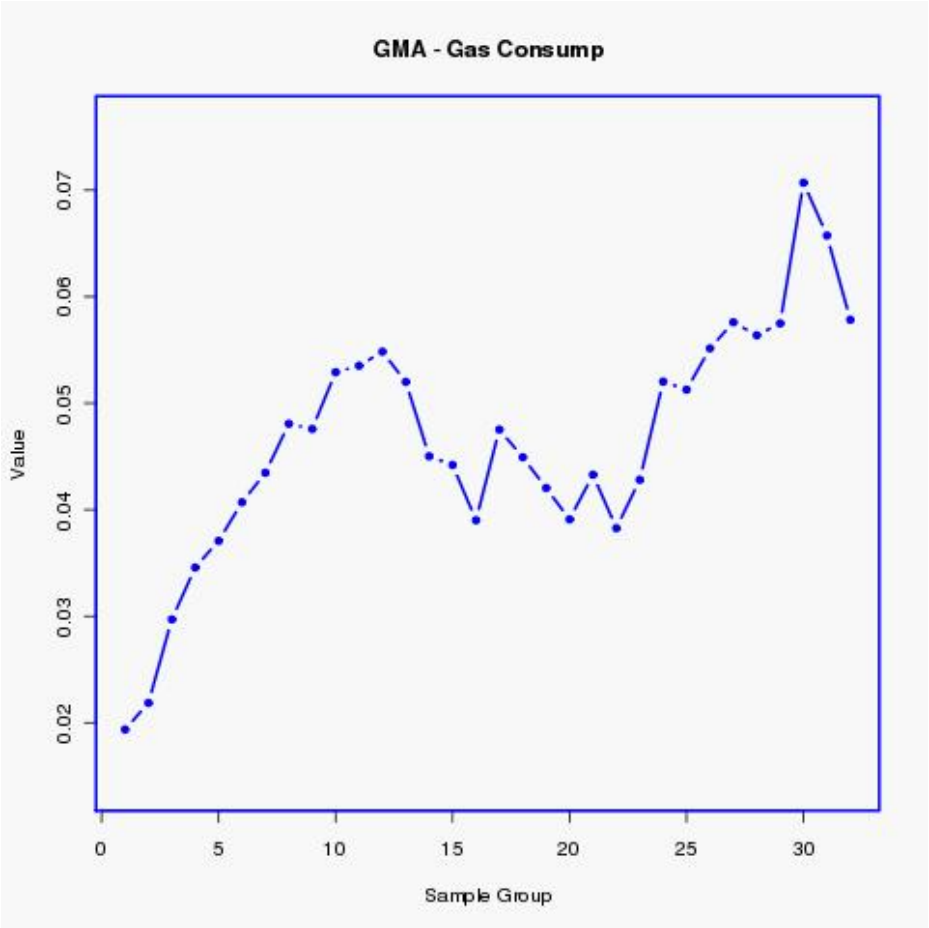
Statistical Process Control Example - X-Bar Chart Output



Statistical Process Control Example - R Chart Output



Statistical Process Control Example - GMA Chart Output



Questions?

Thank You!