# PostgreSQL-embedded Statistical Analysis with PL/R

**Joe Conway**
mail@joeconway.com

O'Reilly Open Source Convention
July 7 - 11, 2003

# Introduction

- 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. It is similar to the S language and environment, which was developed at Bell Laboratories by John Chambers and colleagues, and is sold commercially by Insightful Corp. as S-PLUS.
  - R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible.

# PL/R Prerequisites

- PostgreSQL 7.3 or greater
  - download from www.postgresql.org
- R 1.6.2 or greater
  - download from cran.r-project.org

# Installation

- Obtain and install PostgreSQL and R
  - configure R with **`--enable-R-shlib`**
  - be sure to set **`R_HOME`**
  - note on libR preloading
- Obtain PL/R
  - www.joeconway.com
  - gborg.postgresql.org
  - Debian unstable: "apt-get install postgresql-plr"
- How do I install it?
  - see documentation

# PL/R Advantages

- Leverage people's knowledge and skills
  - statistics is a specialty not common amongst database developers
- Leverage hardware
- Processing/bandwidth efficiency
- Consistency of analysis
- Abstraction of complexity

# PL/R Disadvantages

- PostgreSQL user
  - Slower than standard SQL aggregates (e.g. `AVG`) for simple cases
  - New language to learn
- R user
  - Debugging more challenging than working directly in R
  - Less flexible for ad hoc analysis

# Creating PL/R Functions

- Similar to other PostgreSQL PLs

```
CREATE OR REPLACE FUNCTION func_name(arg-types)
RETURNS return-type AS '
    function body
' LANGUAGE 'plr';
```

- But a little different from standard R functions

```
func_name <- function(argument-names) {
  function body
}
```

# Simple Example - Summary Statistics

- With standard SQL Aggregates

```
select avg(id_val),
       stddev(id_val),
       min(id_val),
       max(id_val),
       max(id_val) - min(id_val),
       count(id_val)
from sample_numeric_data
where ia_id='G121XA34';
```

# Simple Example - Summary Statistics

- Equivalent PL/R function

```
create or replace function statsum(text)
returns summarytup as '
 sql<-paste("select id_val from sample_numeric_data ",
           "where ia_id='''", arg1, "'''", sep="")
 rs <- pg.spi.exec(sql)
 rng <- range(rs[,1])
 return(data.frame(mean = mean(rs[,1]),
  stddev = sd(rs[,1]), min = rng[1], max = rng[2],
  range = rng[2] - rng[1], count = length(rs[,1])))
' language 'plr';
select * from statsum('G121XA34');
```

# Simple Example - Summary Statistics

- EXPLAIN ANALYZE: Standard SQL Aggregates

```
Aggregate  (cost=302.25..302.26 rows=1 width=8)
   (actual time=9.02..9.02 rows=1 loops=1)
   ->  Index Scan using sample_numeric_data_idx2 on
   sample_numeric_data  (cost=0.00..300.87 rows=78
   width=8) (actual time=0.31..7.56 rows=97 loops=1)
         Index Cond: (ia_id = 'G121XA34'::text)
 Total runtime: 9.31 msec
(4 rows)
```

# Simple Example - Summary Statistics

- EXPLAIN ANALYZE: PL/R Function

```
Function Scan on statsum  (cost=0.00..12.50
   rows=1000 width=44) (actual time=34.27..34.33
   rows=1 loops=1)
 Total runtime: 34.45 msec
(2 rows)
```

# Argument Type Conversions

- One-dimensional PostgreSQL arrays are converted to multi-element R vectors
- Two-dimensional PostgreSQL arrays are mapped to R matrixes
- Three-dimensional PostgreSQL arrays are converted to three-dimensional R arrays.
- Composite-types are transformed into R data.frames.

| SQL | R |
|---|---|
| NULL | NA |
| boolean | logical |
| int2, int4 | integer |
| int8, float4, float8, cash, numeric | numeric |
| everything else | character |

# Return Type Conversions

- Data type similar to arguments
- Result Form
  - 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
- See documentation for more detail

# User-defined R Functions

- Why talk about this?
  - PL/R functions are essentially anonymous within the embedded interpreter
  - named R functions can be called from the anonymous PL/R functions
- Creating and loading named R functions
  - dynamically
  - persistently

# Named R Functions - Dynamic

- Use install_rcmd() function

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

CREATE OR REPLACE FUNCTION pg_test_inst(text)
RETURNS text AS 'pg.test.inst(arg1)' LANGUAGE 'plr';

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

# Named R Functions - Persistent

- Use plr_modules table

```
CREATE TABLE plr_modules (modseq int4, modsrc text);
INSERT INTO plr_modules
VALUES (0, 'pg.test.module.load <-function(msg) {print
   (msg)}');
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)
```

# PL/R R Support Functions

- **`pg.spi.exec(sql)`** – execute arbitrary SQL and create an R data.frame
- **`pg.spi.prepare(sql, typeVector)`** – PREPARE a SQL statement for later (repeated) execution
- **`pg.spi.execp(savedPlan, valueList)`** – execute a PREPAREd statement
- **`pg.thrownotice(msg)`** – generate a PostgreSQL NOTICE
- **`pg.throwerror(msg)`** – generate a PostgreSQL ERROR, aborting current transaction
- **`pg.spi.factor(dataframe)`** – convert character columns of data.frame to R “factors”

# PL/R SQL Support Functions

- **`install_rcmd(text)`** – load a named R function into the embedded interpreter
- **`reload_plr_modules()`** – reload named R functions in the the plr_modules table
- **`plr_singleton_array(float8)`** – create single element array
- **`plr_array_push(float8[],float8)`** – push an element onto the end of an array
- **`plr_array_accum(float8[],float8)`** – same as plr_array_push(), but creates array from element if needed
- **`r_typenames()`**, **`plr_environ()`** – auxillary functions
- PostgreSQL 7.4 - related functionality

# Aggregate Example: quantile()

- State function – use plr_array_accum
- Final function – create PL/R function

```
CREATE OR REPLACE FUNCTION r_quantile(float8[])
RETURNS float8[] AS '
  quantile(arg1, probs = seq(0, 1, 0.25),
           names = FALSE)
' LANGUAGE 'plr';

CREATE AGGREGATE quantile (
    sfunc = plr_array_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = r_quantile
);
```

# Aggregate Example: quantile()

```
SELECT workstation, quantile(id_val)
FROM sample_numeric_data
WHERE ia_id = 'G121XB8A'
GROUP BY workstation;

 workstation |                quantile
-------------+-------------------------------
 1051        | {3.89,4.66,4.825,5.2675,5.47}
 1055        | {4.19,5.02,5.21,5.5,6.89}
 1068        | {4.33,5.2625,5.455,5.5275,6.01}
 1070        | {4.51,5.1975,5.485,5.7575,6.41}
(4 rows)
```

# More Complex - Histogram Example

- Histogram function in PL/R

```
create or replace function hist(text)
returns setof histtup as '
 sql<-paste("select id_val from sample_numeric_data ",
            "where ia_id=''", arg1, "''", sep="")
 rs <- pg.spi.exec(sql)
 h <- hist(rs[,1], plot = FALSE)
 return(
  data.frame(
   breaks = h$breaks[1:length(h$breaks)-1],
   count = h$counts))
' language 'plr';
```

# More Complex - Histogram Example

```
select * from hist('G121XA34');
 break | count
-------+-------
     0 |    17
  0.02 |    26
  0.04 |    20
  0.06 |    15
  0.08 |     9
   0.1 |     8
  0.12 |     1
  0.14 |     0
  0.16 |     0
  0.18 |     1
```

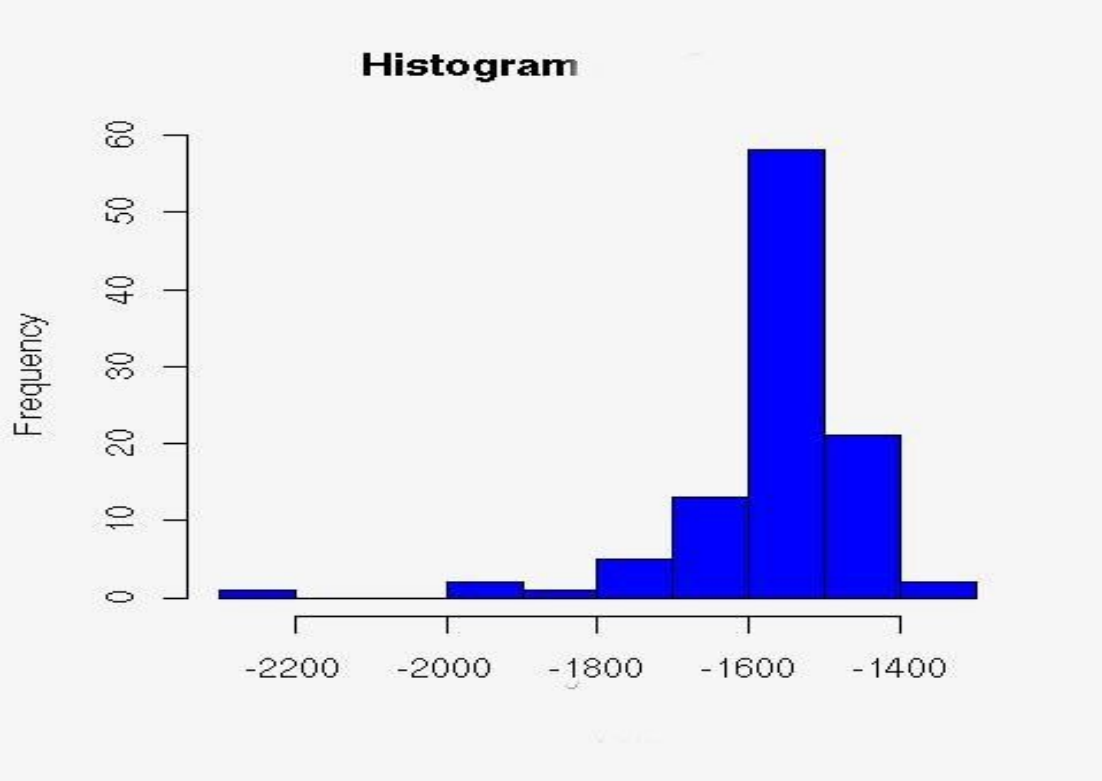
# More Complex - Histogram Example

- Calling it from PHP (w/ modified hist())

```
<snip>
$tmpfilename = 'charts/hist1.jpg';
$sql = "select * from hist('" . $_POST['userdata']
       . "','/tmp/" . $tmpfilename . "')";
$rs = pg_query($conn,$sql);
echo "<img src='$tmpfilename' border=0><br>";
</snip>
```

# More Complex - Histogram Example

- Demo with JPEG output




# Yet More Complex - Statistical Process Control Example

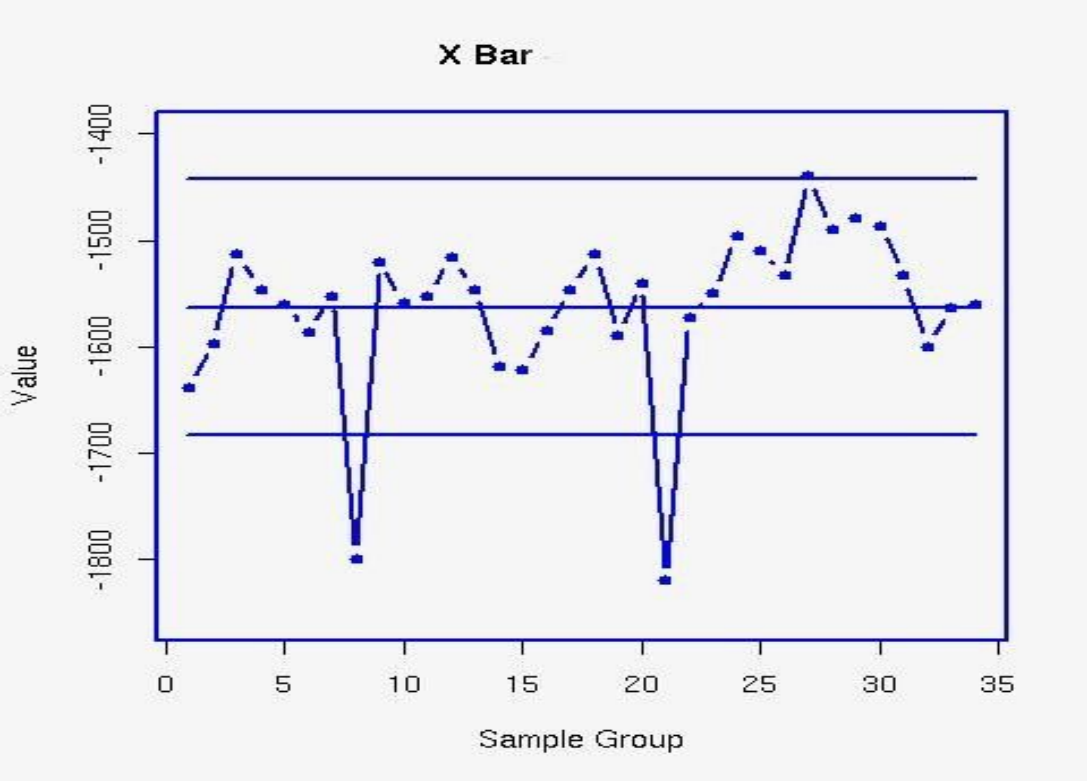
- Named controlChart R function loaded via plr_modules
  - About 120 lines of code
- controlchart() PL/R function
  - Another 130 lines of code

# Yet More Complex - Statistical Process Control Example

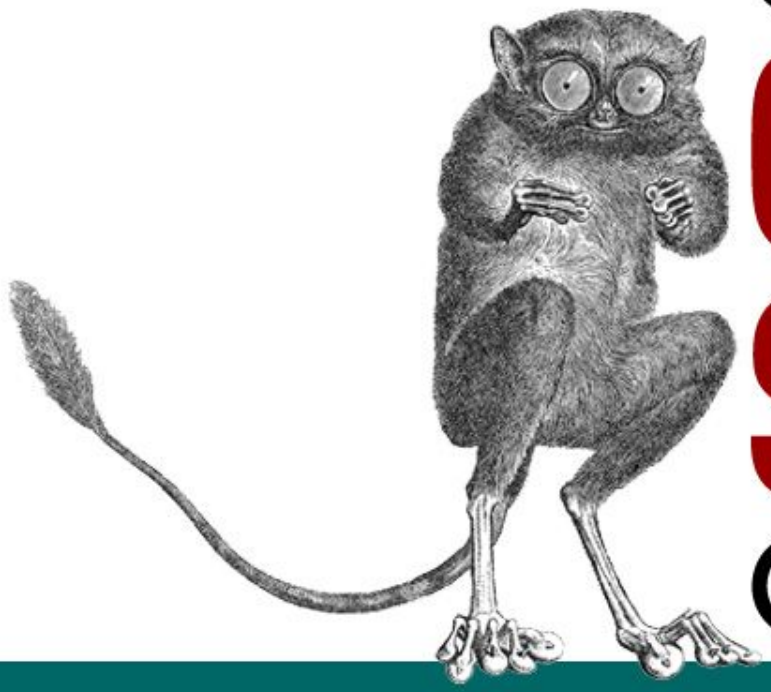
```
select * from controlchart('G121XA34', 3, 0, null)
   limit 1;
-[ RECORD 1 ]-----------------
group_num | 1
xb        | 0.0193605889310595
xbb       | 0.0512444187147061
xucl      | 0.0920736498010521
xlcl      | 0.0104151876283601
r         | 0.0344209665807481
rb        | 0.0559304535429398
rucl      | 0.127521434077903
rlcl      | 0
gma       | 0.0193605889310595
```

# Yet More Complex - Statistical Process Control Example

- Demo with JPEG output




# Questions?



# O'REILLY OPEN SOURCE CONVENTION™

July 7-11, 2003 • PORTLAND, OR