

# Fun with Functions, by Example

Joe Conway  
joe.conway@credativ.com

credativ International

March 26, 2015

# What are Functions?

- Full fledged SQL objects
- Many other database objects are implemented with them
- Fundamental part of PostgreSQL's system architecture
- Created with CREATE FUNCTION
- Executed through normal SQL
  - target-list:  
`SELECT myfunc(f1) FROM foo;`
  - FROM clause:  
`SELECT * FROM myfunc();`
  - WHERE clause:  
`SELECT * FROM foo WHERE myfunc(f1) = 42;`

# How are they Used?

- Functions
- Operators
- Data types
- Index methods
- Casts
- Triggers
- Aggregates
- Ordered-set Aggregates
- Window Functions

# What Forms Can They Take?

- PostgreSQL provides four kinds of functions:
  - SQL
  - Procedural Languages
  - Internal
  - C-language

<http://www.postgresql.org/docs/9.4/static/sql-createfunction.html>

# SQL Functions

- Behavior
  - Arbitrary list of SQL statements separated by semicolons
  - Unless declared to return void, last statement must be SELECT, or INSERT/UPDATE/DELETE with RETURNING
- Arguments
  - Referenced using name or \$n: \$1 is first arg, etc. . .
  - Composite type: dot notation \$1.name used to access
  - Only used as data values, not as identifiers
- Return
  - Singleton: first row of last query result returned  
⇒ NULL on no result
  - Set: all rows of last query result returned  
⇒ empty set on no result

<http://www.postgresql.org/docs/9.4/static/xfunc-sql.html>

# Procedural Languages

- Written in languages besides SQL and C
- Passed to special handler that knows details of the language
- Object library dynamically loaded
- Could be self-contained (e.g. PL/pgSQL)
- Might be externally linked (e.g. PL/Perl)

<http://www.postgresql.org/docs/9.4/static/xplang.html>

# PL/pgSQL

- Self-contained Procedural Language
- SQL plus procedural elements
  - variables
  - conditionals
  - loops
  - cursors
  - error checking
- Installing/uninstalling  
(PostgreSQL 9.0 and later, installed by default):  

```
CREATE EXTENSION plpgsql;  
DROP EXTENSION plpgsql;
```

<http://www.postgresql.org/docs/9.4/static/plpgsql.html>

# Internal Functions

- Statically linked C functions
  - Could use CREATE FUNCTION to create additional alias names for an internal function
  - Most internal functions expect to be declared STRICT

```
CREATE FUNCTION square_root(double precision)
RETURNS double precision AS
'dsqr'
LANGUAGE internal STRICT;
```

<http://www.postgresql.org/docs/9.4/static/xfunc-internal.html>



# C Language Functions

- User-defined functions written in C
  - Compiled into dynamically loadable objects (also called shared libraries)
  - Loaded by the server on demand
  - contrib is good source of examples
  - Same as internal function coding conventions
  - Require PG\_MODULE\_MAGIC call

<http://www.postgresql.org/docs/9.4/static/xfunc-c.html>

# Language Availability

- PostgreSQL includes the following server-side procedural languages:

<http://www.postgresql.org/docs/9.4/static/xplang.html>

- PL/pgSQL
- Perl
- Python
- Tcl

- Other languages available:

[http://pgfoundry.org/softwaremap/trove\\_list.php?form\\_cat=311](http://pgfoundry.org/softwaremap/trove_list.php?form_cat=311)

- Java
- V8 (Javascript)
- Ruby
- R
- Shell
- others ...

# Dollar Quoting

- `$<tag>$`
- `<tag>` is zero or more characters
- Start and End tag must match
- Particularly useful for function bodies
- Works for all character strings
- Nest by choosing different `<tag>` at each level

```
CREATE OR REPLACE FUNCTION dummy()  
RETURNS text AS $_$  
BEGIN  
    RETURN $$Say 'hello'$$;  
END;  
$_$ LANGUAGE plpgsql;
```

# Anonymous Functions

```
DO [ LANGUAGE lang_name ] code
```

- Keyword DO executes anonymous code block
- Transient
- Any procedural language with support, defaults to plpgsql
- No parameters, returns void
- Parsed and executed once
- LANGUAGE clause can be before or after code block

<http://www.postgresql.org/docs/9.4/static/sql-do.html>

# Anonymous Functions

```
DO $_$  
DECLARE r record;  
BEGIN  
    FOR r IN SELECT u.rolname  
              FROM pg_authid u  
              JOIN pg_auth_members m on m.member = u.oid  
              JOIN pg_authid g on g.oid = m.roleid  
              WHERE g.rolname = 'admin'  
    LOOP  
        EXECUTE $$ ALTER ROLE $$ || r.rolname ||  
              $$ SET work_mem = '512MB' $$;  
    END LOOP;  
END$_$;
```

# Anonymous Functions

```
SELECT u.rolname, s.setconfig as setting
FROM pg_db_role_setting s
JOIN pg_authid u on u.oid = s.setrole
JOIN pg_auth_members m on m.member = u.oid
JOIN pg_authid g on g.oid = m.roleid
WHERE g.rolname = 'admin';
  rolname |      setting
-----+-----
 rockstar | {work_mem=512MB}
(1 row)
```

# Changing Existing Functions

- Once created, dependent objects may be created
- Must do `DROP FUNCTION ... CASCADE` to recreate
- Or use `OR REPLACE` to avoid dropping dependent objects
- Very useful for large dependency tree
- Can't be used in some circumstances  
(must drop/recreate instead)

You cannot:

- change function name or input arg types
- change return type
- change types of any OUT parameters

`CREATE OR REPLACE FUNCTION ...;`

# Creating New Functions

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype
           [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
  | WINDOW
  | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter {TO value | = value | FROM CURRENT}
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
} ... [ WITH ( attribute [, ...] ) ]
```



# Function Arguments - argmode

```
( [ [ argmode ] [ argname ] argtype
  [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- argmode (optional): IN, OUT, INOUT, or VARIADIC
  - IN is the default if argmode is omitted
  - Not required (but good style): IN, then INOUT, then OUT
  - Func name + IN/INOUT/VARIADIC arg sig identifies function

```
CREATE FUNCTION testfoo (IN int, INOUT int, OUT int)
RETURNS RECORD AS $$ VALUES ($2, $1 * $2) $$ language sql;
```

```
SELECT * FROM testfoo(14, 3);
```

```
column1 | column2
-----+-----
       3 |      42
(1 row)
```

# Function Arguments - argname

```
( [ [ argmode ] [ argname ] argtype  
  [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- **argname (optional):**

- Most, but not all, languages will use in function body
- Use named notation to improve readability and allow reordering
- Defines the OUT column name in the result row type

```
CREATE FUNCTION testfoo (IN a int, INOUT mult int = 2, OUT a int)  
RETURNS RECORD AS $$ VALUES (mult, a * mult) $$ language sql;
```

```
SELECT * FROM testfoo(mult := 3, a := 14);
```

```
mult | a  
-----+-----  
    3 | 42  
(1 row)
```

# Function Arguments - argtype

```
( [ [ argmode ] [ argname ] argtype  
  [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- argtype (required) (optionally schema-qualified):
  - base, array, composite, or domain types
  - can reference the type of a table column:  
table\_name.column\_name%TYPE
  - Polymorphic "pseudotypes":  
⇒ anyelement, anyarray, anynonarray, anyenum, anyrange

```
CREATE FUNCTION testfoo (INOUT a anyelement, INOUT mult anyelement)  
RETURNS RECORD AS $$ VALUES (a * mult, mult) $$ language sql;  
SELECT * FROM testfoo(mult := 3.14, a := 2.71828);
```

```
      a      | mult  
-----+-----  
  8.5353992 | 3.14  
(1 row)
```

# Function Arguments - default\_expr

```
( [ [ argmode ] [ argname ] argtype
  [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- default\_expr (optional):
  - Expression coercible to arg type
  - Used if arg not provided
  - Any input (IN/INOUT/VARIADIC) can have default
  - All following args must also have defaults

```
CREATE FUNCTION testfoo (IN a int, INOUT mult int = 2, OUT a int)
RETURNS RECORD AS $$ VALUES (mult, a * mult) $$ language sql;
SELECT * FROM testfoo(14);
 mult | a
-----+-----
      2 | 28
(1 row)
```

# Function Overloading

- Input argument (IN/INOUT/VARIADIC) signature used
- Avoid ambiguities:
  - Type (e.g. REAL vs. DOUBLE PRECISION)
  - Function name same as IN composite field name
  - VARIADIC vs same type scalar

```
CREATE OR REPLACE FUNCTION foo (text) RETURNS text
AS $$ SELECT 'Hello ' || $1 $$ LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION foo (int) RETURNS text
AS $$ SELECT ($1 / 2)::text || ' was here' $$ LANGUAGE sql;
```

```
SELECT foo('42'), foo(84);
   foo   |      foo
-----+-----
Hello 42 | 42 was here
(1 row)
```

# Function Return Type

```
[ RETURNS rettype  
  | RETURNS TABLE ( column_name column_type [, ...] ) ]
```

- rettype (required) (optionally schema-qualified):
  - base, array, composite, or domain types
  - can reference the type of a table column:  
table\_name.column\_name%TYPE
  - Polymorphic "pseudotypes":  
⇒ anyelement, anyarray, anynonarray, anyenum, anyrange
  - Special "pseudotypes":
    - language\_handler: procedural language call handler
    - fdw\_handler: foreign-data wrapper handler
    - record: returning an unspecified row type
    - trigger: trigger function
    - void: function returns no value

# Function Return Type

```
[ RETURNS rettype  
  | RETURNS TABLE ( column_name column_type [, ...] ) ]
```

- rettype (required) (optionally schema-qualified):
  - INOUT/OUT args: RETURNS clause may be omitted  
⇒ Note: does not return a set
  - If RETURNS present, must agree with OUT
  - SETOF modifier - "set returning" or "table" function

# Function Return Type - OUT + No RETURNS

```
CREATE FUNCTION testbar1 (OUT f1 int, OUT f2 text) AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar1();
 f1 | f2
----+-----
 42 | hello
(1 row)
```



# Function Return Type - OUT + SETOF RECORD

```
CREATE FUNCTION testbar2 (OUT f1 int, OUT f2 text)
RETURNS SETOF RECORD AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;
```

```
SELECT * FROM testbar2();
```

```
 f1 | f2
----+-----
 42 | hello
 64 | world
(2 rows)
```

# Function Return Type - Custom Type

```
CREATE TYPE testbar3_type AS (f1 int, f2 text);
CREATE FUNCTION testbar3 ()
RETURNS SETOF testbar3_type AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar3();
 f1 |  f2
----+-----
 42 | hello
 64 | world
(2 rows)
```

# Function Return Type - RETURNS TABLE

```
CREATE FUNCTION testbar4 ()
RETURNS TABLE (f1 int, f2 text) AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar4();
```

| f1 | f2    |
|----|-------|
| 42 | hello |
| 64 | world |

(2 rows)

# Function Return Type - unspecified RECORD

```
CREATE FUNCTION testbar5 ()
RETURNS SETOF RECORD AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar5() as t(f1 int, f2 text);
 f1 |  f2
----+-----
 42 | hello
 64 | world
(2 rows)
```

## Function Return Type - RETURNS scalar

```
CREATE FUNCTION testbar6 ()  
RETURNS SETOF int AS $$  
  VALUES (42), (64);  
$$ language sql;  
SELECT * FROM testbar6();  
testbar6  
-----  
      42  
      64  
(2 rows)
```

## Function Return Type - RETURNS scalar with alias

```
CREATE FUNCTION testbar7 ()  
RETURNS SETOF int AS $$  
  VALUES (42), (64);  
$$ language sql;  
SELECT * FROM testbar7() AS t(f1);  
 f1  
----  
 42  
 64  
(2 rows)
```

# Function Return Type - Targetlist

```
SELECT testbar2();  
   testbar2  
-----  
(42,hello)  
(64,world)  
(2 rows)
```

## Function Return Type - Targetlist, expanded

```
SELECT (testbar2()).*;
```

```
 f1 |  f2
```

```
----+-----
```

```
42 | hello
```

```
64 | world
```

```
(2 rows)
```



# LANGUAGE

`LANGUAGE lang_name`

- Language of function body
  - Native: Internal, SQL
  - Interpreted, core: PL/pgSQL, PL/Perl, PL/Python, PL/Tcl
  - Interpreted, external: PL/Java, PL/J, PL/V8, PL/Ruby, PL/R, PL/Sh
  - Compiled, external: Custom C loadable libraries
  - Some (e.g. perl, tcl) have "trusted" and "untrusted" variants

```
CREATE FUNCTION ...
LANGUAGE sql;
LANGUAGE plpgsql;
LANGUAGE plperl;
LANGUAGE plr;
LANGUAGE C;
LANGUAGE internal;
```

# WINDOW

## WINDOW

- Window Functions
  - Can calculate across sets of rows, related to current row
  - Similar to aggregate functions
    - ⇒ but does not cause rows to become grouped
  - Window functions can be written in C, PL/R, PL/V8, others?

# Volatility

- **VOLATILE** (default)
  - Each call can return a different result  
Example: `random()` or `clock_timestamp()`
  - Functions modifying table contents must be declared volatile
- **STABLE**
  - Returns same result for same arguments within single query  
Example: `now()`
  - Consider configuration settings that affect output
- **IMMUTABLE**
  - Always returns the same result for the same arguments  
Example: `lower('ABC')`
  - Unaffected by configuration settings
  - Not dependent on table contents

# Volatility

```
select lower('ABC'), now()::time, clock_timestamp()::time
from generate_series(1,3);
```

| lower | now             | clock_timestamp |
|-------|-----------------|-----------------|
| abc   | 15:32:27.174246 | 15:32:27.174499 |
| abc   | 15:32:27.174246 | 15:32:27.174511 |
| abc   | 15:32:27.174246 | 15:32:27.174514 |

(3 rows)

```
select lower('ABC'), now()::time, clock_timestamp()::time
from generate_series(1,3);
```

| lower | now             | clock_timestamp |
|-------|-----------------|-----------------|
| abc   | 15:32:49.715606 | 15:32:49.715894 |
| abc   | 15:32:49.715606 | 15:32:49.715908 |
| abc   | 15:32:49.715606 | 15:32:49.715913 |

(3 rows)

## Behavior with Null Input Values

- CALLED ON NULL INPUT (default)
  - Function called normally with the null input values
- RETURNS NULL ON NULL INPUT
  - Function not called when null input values are present
  - Instead, null is returned automatically

```
CREATE FUNCTION sum1 (int, int) RETURNS int AS $$
    SELECT $1 + $2
```

```
$$ LANGUAGE SQL RETURNS NULL ON NULL INPUT;
```

```
CREATE FUNCTION sum2 (int, int) RETURNS int AS $$
    SELECT COALESCE($1, 0) + COALESCE($2, 0)
```

```
$$ LANGUAGE SQL CALLED ON NULL INPUT;
```

```
SELECT sum1(9, NULL) IS NULL AS "true", sum2(9, NULL);
```

```
  true | sum2
-----+-----
    t   |    9
(1 row)
```

# Security Attributes - LEAKPROOF

- Planner may push LEAKPROOF functions into views created with the security\_barrier option
- LEAKPROOF requirements
  - No side effects
  - Reveals no info about args other than by return value
  - Can only be set by the superuser
- Be sure function really is leak proof before making LEAKPROOF
- Why use LEAKPROOF at all?
  - Performance (predicate push down)

# Security Attributes - SECURITY INVOKER/DEFINER

- SECURITY INVOKER (default)
  - Function executed with the rights of the current user
- SECURITY DEFINER
  - Executed with rights of creator, like "setuid"

# Security Attributes - SECURITY INVOKER/DEFINER

```
\c - postgres
CREATE TABLE foo (f1 int);
INSERT INTO foo VALUES(42);
REVOKE ALL ON foo FROM public;
```

```
CREATE FUNCTION see_foo() RETURNS TABLE (luser name, f1 int) AS $$
    SELECT CURRENT_USER, f1 FROM foo
$$ LANGUAGE SQL SECURITY DEFINER;
```

```
\c - guest
SELECT * FROM foo;
ERROR: permission denied for relation foo
SELECT CURRENT_USER AS me, luser AS definer, f1 FROM see_foo();
   me   | definer | f1
-----+-----+---
 guest | postgres | 42
(1 row)
```



# Optimizer Hints

`COST execution_cost`

`ROWS result_rows`

- `execution_cost`
  - Estimated execution cost for the function
  - Positive floating point number
  - Units are `cpu_operator_cost`
  - Cost is per returned row
  - Default: 1 unit for C-language/internal, 100 units for all others
- `result_rows`
  - Estimated number rows returned
  - Positive floating point number
  - Only allowed when declared to return set
  - Default: 1000

# Optimizer Hints

```
CREATE FUNCTION testbar8 ()
RETURNS SETOF int AS $$
    VALUES (42), (64);
$$ LANGUAGE sql COST 0.1 ROWS 2;
```

```
SELECT procost, prorows FROM pg_proc WHERE proname = 'testbar8';
   procost | prorows
-----+-----
       0.1 |        2
(1 row)
```

# Function Local Configs

```
SET configuration_parameter
{ TO value | = value | FROM CURRENT }
```

- SET clause
  - Specified config set to value for duration of function
  - SET FROM CURRENT uses session's current value

```
CREATE FUNCTION testbar9 ()
RETURNS SETOF int AS $$
    VALUES (42), (64);
$$ LANGUAGE sql SET work_mem = '512MB';
```

```
SELECT proconfig FROM pg_proc WHERE proname = 'testbar9';
    proconfig
-----
 {work_mem=512MB}
(1 row)
```

# Function Body

```
AS definition  
| AS obj_file, link_symbol
```

- definition
  - String literal
  - Parse by language parser
  - Can be internal function name
  - Can be path to object file if C language function name matches
  - Dollar quote, or escape single quotes and backslashes

# Function Body

AS definition

| AS obj\_file, link\_symbol

- obj\_file, link\_symbol
  - Used when C language function name does not match SQL function name
  - obj\_file is path to object file
    - ⇒ \$libdir: replaced by package lib dir name, determined at build time
  - link\_symbol is name of function in C source code
  - When more than one FUNCTION call refers to same object file, file only loaded once

```
# pg_config --pkglibdir  
/usr/local/pgsql-REL9_4_STABLE/lib
```

# Function Body

```
CREATE FUNCTION foobar ()  
RETURNS int AS $$  
    SELECT 42;  
$$ LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION plr_version ()  
RETURNS text  
AS '$libdir/plr','plr_version'  
LANGUAGE C;
```

# Custom Operator

```
CREATE FUNCTION sum (text, text)
RETURNS text AS $$
    SELECT $1 || ' ' || $2
$$ LANGUAGE SQL;
```

```
CREATE OPERATOR + (
    procedure = sum,
    leftarg = text,
    rightarg = text
);
```

```
SELECT 'hello' + 'world';
?column?
-----
hello world
(1 row)
```

# Custom Aggregate

```
CREATE OR REPLACE FUNCTION concat_ws_comma(text, ANYELEMENT)
RETURNS text AS $$
    SELECT concat_ws(',', $1, $2)
$$ LANGUAGE sql;
```

```
CREATE AGGREGATE str_agg (ANYELEMENT) (
    sfunc = concat_ws_comma,
    stype = text);
```

```
SELECT str_agg(f1) FROM foo;
   str_agg
-----
   41,42
(1 row)
```



# Thank You

- Questions?

# SETOF with OUT Arguments

```
CREATE OR REPLACE FUNCTION sql_with_rows(OUT a int, OUT b text)
RETURNS SETOF RECORD AS $$
    values (1,'a'),(2,'b')
$$ LANGUAGE SQL;
```

```
select * from sql_with_rows();
 a | b
----+---
 1 | a
 2 | b
(2 rows)
```

# INSERT RETURNING

```
CREATE TABLE foo (f0 serial, f1 int, f2 text);
```

```
CREATE OR REPLACE FUNCTION
```

```
sql_insert_returning(INOUT f1 int, INOUT f2 text, OUT id int) AS $$
```

```
    INSERT INTO foo(f1, f2) VALUES ($1,$2) RETURNING f1, f2, f0
    $$ LANGUAGE SQL;
```

```
SELECT * FROM sql_insert_returning(1,'a');
```

```
  f1 | f2 | id
-----+-----+-----
   1 | a  |  1
(1 row)
```

# Composite Argument

```
CREATE TABLE emp (name      text,
                   salary    numeric,
                   age        integer,
                   cubicle    point);
```

```
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
```

```
SELECT name, double_salary(emp.*) AS dream
FROM emp WHERE emp.cubicle ~= point '(2,1)';
```

```
SELECT name,
       double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
FROM emp;
```

# Polymorphic

```
CREATE FUNCTION myappend(anyarray, anyelement) RETURNS anyarray AS
$$
    SELECT $1 || $2;
$$ LANGUAGE SQL;

SELECT myappend(ARRAY[42,6], 21), myappend(ARRAY['abc','def'], 'xyz');
 myappend | myappend
-----+-----
 {42,6,21} | {abc,def,xyz}
(1 row)
```

# Target List versus FROM Clause

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
SELECT new_emp();
        new_emp
```

```
-----
(None,1000.0,25,"(2,2)")
```

```
SELECT * FROM new_emp();
 name | salary | age | cubicle
-----+-----+-----+-----
None  | 1000.0 | 25  | (2,2)
```

```
SELECT (new_emp()).name;
 name
-----
None
```

# VARIADIC

```
CREATE FUNCTION mleast(VARIADIC numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

```
SELECT mleast(10, -1, 5, 4.4);
 mleast
-----
      -1
(1 row)
```

```
SELECT mleast(42, 6, 42.42);
 mleast
-----
       6
(1 row)
```

# DEFAULT Arguments

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int LANGUAGE SQL AS $$SELECT $1 + $2 + $3$$;
```

```
SELECT foo(10, 20, 30);
```

```
foo
```

```
-----
```

```
60
```

```
(1 row)
```

```
SELECT foo(10, 20);
```

```
foo
```

```
-----
```

```
33
```

```
(1 row)
```



# Simple

```
CREATE OR REPLACE FUNCTION sum (text, text)
RETURNS text AS $$
BEGIN
    RETURN $1 || ' ' || $2;
END;
$$ LANGUAGE plpgsql;

SELECT sum('hello', 'world');
      sum
-----
hello world
(1 row)
```

# Parameter ALIAS

```
CREATE OR REPLACE FUNCTION sum (int, int)
RETURNS int AS $$
  DECLARE
    i ALIAS FOR $1;
    j ALIAS FOR $2;
    sum int;
  BEGIN
    sum := i + j;
    RETURN sum;
  END;
$$ LANGUAGE plpgsql;

SELECT sum(41, 1);
   sum
-----
   42
(1 row)
```

# Named Parameters

```
CREATE OR REPLACE FUNCTION sum (i int, j int)
RETURNS int AS $$
  DECLARE
    sum int;
  BEGIN
    sum := i + j;
    RETURN sum;
  END;
$$ LANGUAGE plpgsql;

SELECT sum(41, 1);
 sum
-----
  42
(1 row)
```

## Control Structures: IF ...

```
CREATE OR REPLACE FUNCTION even (i int)
RETURNS boolean AS $$
  DECLARE
    tmp int;
  BEGIN
    tmp := i % 2;
    IF tmp = 0 THEN RETURN true;
    ELSE RETURN false;
    END IF;
  END;
$$ LANGUAGE plpgsql;

SELECT even(3), even(42);
 even | even
-----+-----
 f    | t
(1 row)
```

# Control Structures: FOR ... LOOP

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
DECLARE
    tmp numeric; result numeric;
BEGIN
    result := 1;
    FOR tmp IN 1 .. i LOOP
        result := result * tmp;
    END LOOP;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
SELECT factorial(42::numeric);
           factorial
```

```
-----
14050061177528798985431426062445115699363840000000000
(1 row)
```

# Control Structures: WHILE ... LOOP

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
    DECLARE tmp numeric; result numeric;
    BEGIN
        result := 1; tmp := 1;
        WHILE tmp <= i LOOP
            result := result * tmp;
            tmp := tmp + 1;
        END LOOP;
        RETURN result;
    END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);
               factorial
-----
1405006117752879898543142606244511569936384000000000
(1 row)
```

# Recursive

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
BEGIN
    IF i = 0 THEN
        RETURN 1;
    ELSIF i = 1 THEN
        RETURN 1;
    ELSE
        RETURN i * factorial(i - 1);
    END IF;
END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);
               factorial
-----
1405006117752879898543142606244511569936384000000000
(1 row)
```

# Record types

```
CREATE OR REPLACE FUNCTION format ()
RETURNS text AS $$
    DECLARE
        tmp RECORD;
    BEGIN
        SELECT INTO tmp 1 + 1 AS a, 2 + 2 AS b;
        RETURN 'a = ' || tmp.a || ' ; b = ' || tmp.b;
    END;
$$ LANGUAGE plpgsql;

select format();
      format
-----
a = 2; b = 4
(1 row)
```



# PERFORM

```
CREATE OR REPLACE FUNCTION func_w_side_fx() RETURNS void AS
$$ INSERT INTO foo VALUES (41),(42) $$ LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION dummy ()
RETURNS text AS $$
```

```
    BEGIN
```

```
        PERFORM func_w_side_fx();
```

```
        RETURN 'OK';
```

```
    END;
```

```
$$ LANGUAGE plpgsql;
```

```
SELECT dummy();
```

```
SELECT * FROM foo;
```

```
    f1
```

```
----
```

```
    41
```

```
    42
```

```
(2 rows)
```

# Dynamic SQL

```
CREATE OR REPLACE FUNCTION get_foo(i int)
RETURNS foo AS $$
  DECLARE
    rec RECORD;
  BEGIN
    EXECUTE 'SELECT * FROM foo WHERE f1 = ' || i INTO rec;
    RETURN rec;
  END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_foo(42);
 f1
----
 42
(1 row)
```

# Cursors

```
CREATE OR REPLACE FUNCTION totalbalance()
RETURNS numeric AS $$
DECLARE
    tmp RECORD; result numeric;
BEGIN
    result := 0.00;
    FOR tmp IN SELECT * FROM foo LOOP
        result := result + tmp.f1;
    END LOOP;
    RETURN result;
END;
$$ LANGUAGE plpgsql;

SELECT totalbalance();
      totalbalance
-----
          83.00
(1 row)
```

# Error Handling

```
CREATE OR REPLACE FUNCTION safe_add(a integer, b integer)
RETURNS integer AS $$
BEGIN
    RETURN a + b;
EXCEPTION
    WHEN numeric_value_out_of_range THEN
        -- do some important stuff
        RETURN -1;
    WHEN OTHERS THEN
        -- do some other important stuff
        RETURN -1;
END;
$$ LANGUAGE plpgsql;
```

<http://www.postgresql.org/docs/9.4/static/errcodes-appendix.html>

# Nested Exception Blocks

```
CREATE FUNCTION merge_db(key integer, data text)
RETURNS void AS $$
BEGIN
    LOOP
        UPDATE db SET b = data WHERE a = key;
        IF found THEN RETURN;
        END IF;
        BEGIN
            INSERT INTO db (a, b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- do nothing
        END;
    END LOOP;
EXCEPTION WHEN OTHERS THEN
    -- do something else
END;
$$ LANGUAGE plpgsql;
```

# Security Attributes - LEAKPROOF

```
\c - postgres
DROP TABLE IF EXISTS all_books CASCADE;
CREATE TABLE all_books(id serial primary key,
                        luser text,
                        bookname text,
                        price int);

INSERT INTO all_books
SELECT g.f,
       CASE WHEN g.f % 2 = 0 THEN 'joe' ELSE 'tom' END,
       'book-' || g.f::text,
       40 + g.f % 20
FROM generate_series(1,8) as g(f);

DROP VIEW IF EXISTS user_books;
CREATE VIEW user_books AS
SELECT id, luser, bookname, price FROM all_books
WHERE luser = CURRENT_USER;
GRANT ALL ON user_books TO public;
```

# Security Attributes - LEAKPROOF

- Note the "COST 1" below ...

```
CREATE OR REPLACE FUNCTION leak_info(text, text) returns int AS $$  
BEGIN  
  IF $1 != CURRENT_USER THEN  
    RAISE NOTICE '%:%', $1, $2;  
  END IF;  
  RETURN 0;  
END;  
$$ COST 1 LANGUAGE plpgsql;
```

# Security Attributes - LEAKPROOF

```
\c - joe
EXPLAIN ANALYZE SELECT * FROM user_books
  WHERE leak_info(luser, bookname) = 0;
NOTICE:  tom:book-1
NOTICE:  tom:book-3
NOTICE:  tom:book-5
NOTICE:  tom:book-7
```

## QUERY PLAN

```
-----
Seq Scan on all_books (cost=0.00..1.18 rows=1 width=72) (actual ...
  Filter: ((leak_info(luser, bookname) = 0) AND
           (luser = ("current_user"())::text))
  Rows Removed by Filter: 4
Planning time: 0.674 ms
Execution time: 2.044 ms
(5 rows)
```



# Security Attributes - LEAKPROOF

- Note the "WITH (security\_barrier)" below ...

```
\c - postgres
DROP VIEW user_books;
CREATE VIEW user_books WITH (security_barrier) AS
  SELECT id, luser, bookname, price FROM all_books
  WHERE luser = CURRENT_USER;
GRANT ALL ON user_books TO public;
```

# Security Attributes - LEAKPROOF

```
\c - joe
EXPLAIN ANALYZE SELECT * FROM user_books
WHERE leak_info(luser, bookname) = 0;
QUERY PLAN
```

```
-----
Subquery Scan on user_books (cost=0.00..1.16 rows=1 width=72) (actual ...
  Filter: (leak_info(user_books.luser, user_books.bookname) = 0)
  -> Seq Scan on all_books (cost=0.00..1.14 rows=1 width=72) (actual ...
    Filter: (luser = ("current_user"())::text)
    Rows Removed by Filter: 4
Planning time: 0.648 ms
Execution time: 1.903 ms
(7 rows)
```

# Security Attributes - LEAKPROOF

```
\c - postgres
ALTER FUNCTION leak_info(text, text) LEAKPROOF;
```

```
\c - joe
EXPLAIN ANALYZE SELECT * FROM user_books
  WHERE leak_info(luser, bookname) = 0;
NOTICE:  tom:book-1
NOTICE:  tom:book-3
NOTICE:  tom:book-5
NOTICE:  tom:book-7
```

## QUERY PLAN

```
-----
Seq Scan on all_books  (cost=0.00..1.18 rows=1 width=72) (actual ...
  Filter: ((leak_info(luser, bookname) = 0) AND
           (luser = ("current_user"())::text))
  Rows Removed by Filter: 4
Planning time: 0.646 ms
Execution time: 2.145 ms
(5 rows)
```

# Window Function

```
CREATE TABLE mydata (
    pk int primary key,
    mydate date NOT NULL,
    gender text NOT NULL CHECK(gender IN ('M','F')),
    mygroup text NOT NULL,
    id int NOT NULL
);

INSERT INTO mydata VALUES
(1, '2012-03-25', 'F', 'A', 1), (2, '2005-05-23', 'F', 'B', 2),
(3, '2005-09-08', 'F', 'B', 2), (4, '2005-12-07', 'F', 'B', 2),
(5, '2006-02-26', 'F', 'C', 2), (6, '2006-05-13', 'F', 'C', 2),
(7, '2006-09-01', 'F', 'C', 2), (8, '2006-12-12', 'F', 'D', 2),
(9, '2006-02-19', 'F', 'D', 2), (10, '2006-05-03', 'F', 'D', 2),
(11, '2006-04-23', 'F', 'D', 2), (12, '2007-12-08', 'F', 'D', 2),
(13, '2011-03-19', 'F', 'D', 2), (14, '2007-12-20', 'M', 'A', 3),
(15, '2008-06-15', 'M', 'A', 3), (16, '2008-12-16', 'M', 'A', 3),
(17, '2009-06-07', 'M', 'B', 3), (18, '2009-10-09', 'M', 'B', 3),
(19, '2010-01-28', 'M', 'B', 3), (20, '2007-06-05', 'M', 'A', 4);
```

# Window Function

```
SELECT id, gender, obs_days, sum(chgd) as num_changes FROM
(SELECT id, gender,
    CASE WHEN row_number() OVER w > 1
        AND mygroup <> lag(mygroup) OVER w THEN 1
        ELSE 0 END AS chgd,
    last_value(mydate) OVER w - first_value(mydate) OVER w AS obs_days
FROM mydata
WINDOW w AS
(PARTITION BY id, gender ORDER BY id, gender, mydate
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
) AS ss GROUP BY id, gender, obs_days ORDER BY id, gender;
```

| id | gender | obs_days | num_changes |
|----|--------|----------|-------------|
| 1  | F      | 0        | 0           |
| 2  | F      | 2126     | 5           |
| 3  | M      | 770      | 1           |
| 4  | M      | 0        | 0           |

(4 rows)

# Lateral

```
SELECT d.datname, u.rolname, c.config
FROM pg_db_role_setting s
LEFT JOIN pg_authid u ON u.oid = s.setrole
LEFT JOIN pg_database d ON d.oid = s.setdatabase,
LATERAL unnest(s.setconfig) c(config);
```

| datname           | rolname  | config                           |
|-------------------|----------|----------------------------------|
| -----+-----+----- |          |                                  |
|                   | rockstar | work_mem=512MB                   |
| test              |          | search_path="public, testschema" |
| test              |          | work_mem=128MB                   |
| test              |          | statement_timeout=10s            |
|                   | joe      | statement_timeout=60s            |
|                   | joe      | log_min_duration_statement=10s   |
|                   | joe      | maintenance_work_mem=4GB         |