

PostgreSQL Functions By Example

Joe Conway
joe.conway@credativ.com

credativ Group

January 20, 2012

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

What Forms Can They Take?

- PostgreSQL provides four kinds of functions:
 - SQL
 - Procedural Languages
 - Internal
 - C-language
- Arguments
 - Base, composite, or combinations
 - Scalar or array
 - Pseudo or polymorphic
 - VARIADIC
 - IN/OUT/INOUT
- Return
 - Singleton or set (SETOF)
 - Base or composite type
 - Pseudo or polymorphic

SQL Functions

- Behavior
 - Executes an arbitrary list of SQL statements separated by semicolons
 - Last statement may be INSERT, UPDATE, or DELETE with RETURNING clause
- Arguments
 - Referenced by function body using \$n: \$1 is first arg, etc. . .
 - If composite type, then dot notation \$1.name used to access
 - Only used as data values, not as identifiers
- Return
 - If singleton, first row of last query result returned, NULL on no result
 - If SETOF, all rows of last query result returned, empty set on no result

Procedural Languages

- User-defined functions
- Written in languages besides SQL and C
 - Task is passed to a special handler that knows the details of the language
 - Handler could be self-contained (e.g. PL/pgSQL)
 - Handler could be dynamically loaded (e.g. PL/Perl)

<http://www.postgresql.org/docs/9.1/interactive/xplang.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  
'dsqrt'  
LANGUAGE internal STRICT;
```

<http://www.postgresql.org/docs/9.1/interactive/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
 - Needs separate topic

<http://www.postgresql.org/docs/9.1/interactive/xfunc-c.html>



Language Availability

- PostgreSQL includes the following server-side procedural languages:

<http://www.postgresql.org/docs/9.1/interactive/xplang.html>

- PL/pgSQL
 - Perl
 - Python
 - Tcl
- Other languages available:

http://pgfoundry.org/softwaremap/trove_list.php?form_cat=311

- Java
- PHP
- Ruby
- R
- Shell
- others ...

Creating New Functions

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } defexpr ] [, ...
    [ RETURNS rettype
    | RETURNS TABLE ( colname coltype [, ... ] ) ]
{ LANGUAGE langname
  | WINDOW
  | IMMUTABLE | STABLE | VOLATILE
  | 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 [, ... ] ) ]
```

<http://www.postgresql.org/docs/9.1/interactive/sql-createfunction.html>

Dollar Quoting

- Works for all character strings
- Particularly useful for function bodies

```
CREATE OR REPLACE FUNCTION dummy () RETURNS text AS
$$
  DECLARE
    result text;
  BEGIN
    PERFORM 'SELECT 1+1';
    RETURN 'ok';
  END;
$$
LANGUAGE plpgsql;
```

<http://www.postgresql.org/docs/9.1/static/sql-syntax-lexical.html#SQL-SYNTAX-DOLLAR-QUOTING>



Function Overloading

- IN argument 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 $1
```

```
$$ LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION foo (int) RETURNS text AS $$
  SELECT ($1 + 1)::text
```

```
$$ LANGUAGE sql;
```

```
SELECT foo('42'), foo(41);
```

```
foo | foo
```

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

```
42  | 42
```

```
(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 argument types
 - change return type
 - change types of any `OUT` parameters

```
CREATE OR REPLACE FUNCTION ...;
```



Volatility

- VOLATILE (default)
 - Each call can return a different result
Example: `random()` or `timeofday()`
 - 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

```
select lower('ABC'), now(), timeofday() from generate_series(1,3);
```



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

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

```
CREATE TABLE foo (f1 int);
REVOKE ALL ON foo FROM public;
CREATE FUNCTION see_foo() RETURNS SETOF foo AS $$
    SELECT * FROM foo
$$ LANGUAGE SQL SECURITY DEFINER;
```

```
\c - guest
```

You are now connected to database "postgres" as user "guest".

```
SELECT * FROM foo;
ERROR: permission denied for relation foo
SELECT * FROM see_foo();
   f1
----
(0 rows)
```



Simple

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

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



Custom Operator

```
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)
```



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)
```



PL/pgSQL

- PL/pgSQL is SQL plus procedural elements
 - variables
 - if/then/else
 - loops
 - cursors
 - error checking
- Loading the language handler into a database:

```
createlang plpgsql dbname
```

<http://www.postgresql.org/docs/9.1/interactive/plpgsql.html>



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
```

```
1405006117752879898543142606244511569936384000000000
(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/9.1/interactive/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;
  
```



Thank You

- Questions?