PostgreSQL
Functions

Βάσεις Δεδομένων
Πολυτεχνείο Κρήτης

PostgreSQL Functions

- **PostgreSQL provides four kinds of functions:**
  - query language functions (functions written in SQL)
  - procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl)
  - internal functions
  - C-language functions

- Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

PostgreSQL Functions

- Advantages of using PostgreSQL stored functions
  - Reduce the number of round trips between application and database servers. All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.
  - Increase application performance because the user-defined functions are pre-compiled and stored in the PostgreSQL database server.
  - Be able to reuse in many applications. Once you develop a function, you can reuse it in any applications.
- Disadvantages of using PostgreSQL stored functions
  - Slow in software development because it requires specialized skills that many developers do not possess.
  - Make it difficult to manage versions and hard to debug.
  - May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server

Functions (Definition)

```
CREATE [OR REPLACE] FUNCTION func_name(arg1 arg1_datatype [, ..])
RETURNS some_type | SETOF sometype | TABLE (..) AS
$$
    BODY of function ..
    ..
$$
LANGUAGE language_of_function
IMMUTABLE | STABLE | VOLATILE

--LANGUAGE: sql, plpgsql (or others like perl, tcl, python)
```

Functions (SQL)

- Functions execute an arbitrary list of SQL statements
- The body of an SQL function must be a list of SQL statements separated by semicolons. A semicolon after the last statement is optional.
- Any collection of commands in the SQL language can be packaged together and defined as a function. Besides SELECT queries, the commands can include data modification queries (INSERT, UPDATE, and DELETE), as well as other SQL commands.
- The final command must be a SELECT or have a RETURNING clause that returns whatever is specified as the function's return type. Alternatively, if you want to define a SQL function that performs actions but has no useful value to return, you can define it as returning void

Functions (SQL)

- The function interface defines the args and the return type
- You can refer to variables by their name or ordinal position
  $1, $2, $3 etc .
- After the body, is noted the Language and a tag that denotes how it should be cached. In this case we have noted:
  - IMMUTABLE meaning that the output of the function can be expected to be the same if the inputs are the same.
  - Other options are STABLE - meaning it will not change within a query given same inputs and
  - VOLATILE such as functions involving random() and CURRENT_TIMESTAMP that can be expected to change output even in the same query call.
- PostgreSQL 8.3 introduced the ability to set costs and estimated rows returned for a function.

Functions (SQL)

No return value

```
CREATE FUNCTION clean_emp() RETURNS void AS
$$
   DELETE FROM emp WHERE salary < 0;
$$
LANGUAGE SQL;

SELECT clean_emp();
```

Return simple type

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS
$$
   UPDATE bank  SET balance = balance - debit
   WHERE accountno = tf1.accountno
   RETURNING balance;
   - - (or  SELECT balance FROM bank WHERE accountno = tf1.accountno;)
$$ LANGUAGE SQL;
```

Functions(SQL)

SQL Functions on Composite Types

Input argument of composite type

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

INSERT INTO emp VALUES ('Bill', 4200, 45);

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;

 name | dream
---------+-------
 Bill     |  8400
```

Construct a composite argument value

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

Function that returns a composite type

```
CREATE FUNCTION new_emp() RETURNS emp
AS $$
    SELECT text 'None' AS name,
                1000.0 AS salary,
                25 AS age;
$$ LANGUAGE SQL;
```

Return only one field of composite

```
SELECT (new_emp()).name;
– OR
CREATE FUNCTION getname(emp) RETURNS text
AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
```

Functions (SQL)

OUT parameters

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int) AS $$
        SELECT x + y, x * y
$$ LANGUAGE SQL;

SELECT sum_n_product(11,42);
```

Return custom type

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2' LANGUAGE SQL;
```

Functions with Variable Numbers of Arguments

```
CREATE FUNCTION choose(VARIADIC arr character varying[]) RETURNS character varying  AS $$
    SELECT $1[3];
$$ LANGUAGE SQL;

SELECT choose('hello','world','hello','chania');
```

Functions as Table Sources (use in FROM clause)

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe'); INSERT INTO foo VALUES (1, 2, 'Ed'); INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;    -- returns only one row
```

Functions (SQL – PL/pgSQL)

Functions returning sets (USING SQL)

Using table:

```
CREATE FUNCTION sel_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text, log_ts timestamptz) AS $$
        SELECT log_id, user_name, description, log_ts FROM logs WHERE user_name = $1;
$$ LANGUAGE SQL STABLE;
```

Using OUT parameters:

```
CREATE FUNCTION sel_logs_out(puname varchar, OUT log_id int, OUT uname varchar, OUT desc text, OUT log_ts timestamptz)
RETURNS SETOF record AS $$
        SELECT * FROM logs WHERE user_name = $1;
$$ LANGUAGE SQL STABLE;
```

Using composite type:

```
CREATE FUNCTION sel_logs_so (param_user_name varchar)
RETURNS SETOF logs AS $$
        SELECT * FROM logs WHERE user_name = $1;
$$ LANGUAGE SQL STABLE;
```

Functions returning sets USING PL/pgSQL

```
CREATE FUNCTION sel_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text, log_ts timestamptz) AS $$
BEGIN
RETURN QUERY
        SELECT log_id, user_name, description, log_ts
        FROM logs WHERE user_name = param_user_name;
END;
$$ LANGUAGE plpgsql STABLE;
```

Functions (PL)

- PostgreSQL allows user-defined functions to be written in other languages besides SQL and C.
- These other languages are generically called procedural languages (PLs).
- For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a special handler that knows the details of the language.
- The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as "glue" between PostgreSQL and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function.

- **There are currently four procedural languages available in the standard PostgreSQL distribution:**
  - **PL/pgSQL**
  - PL/Tcl
  - PL/Perl
  - PL/Python

PL/pgSQL

- PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function is organized into blocks.

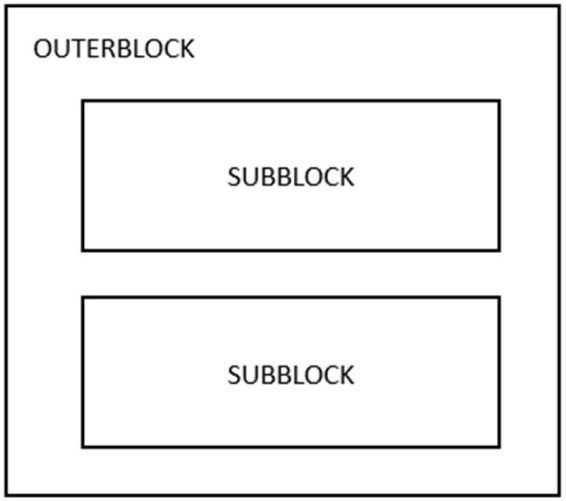
```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements;
END [ label ];
```

- Each block has two sections: declaration and body. The declaration section is optional while the body section is required. The block is ended with a semicolon (;) after the END keyword.
- A block may have an optional label located at the beginning and at the end. You use the block label in case you want to specify it in the EXIT statement of the block body or if you want to qualify the names of variables declared in the block.
- The declaration section is where you declare all variables used within the body section. Each statement in the declaration section is terminated with a semicolon (;).
- The body section is where you place the code. Each statement in the body section is also terminated with a semicolon (;).

PL/pgSQL

- Anonymoys Blocks
- Example:

```
DO $$
<<first_block>>
DECLARE
  counter integer := 0;
BEGIN
   counter := counter + 1;
   RAISE NOTICE 'The current value of counter is %', counter;
END first_block $$;
```



Functions (PL/pgSQL)

Example of Function using PL/pgSQL

```
CREATE OR REPLACE FUNCTION fnsomefunc (numtimes integer, msg text) RETURNS text AS $$
        DECLARE
           strresult text;
        BEGIN
           strresult := '';
           IF numtimes = 42 THEN
              strresult := 'Right you are!';
           ELSIF numtimes > 0 AND numtimes < 100 THEN
              FOR i IN 1 .. numtimes LOOP
                 strresult := strresult || msg || '\r\n';
              END LOOP;
           ELSE
              strresult := 'You can not do that. Please don''t abuse our generosity.';
              IF numtimes <= 0 THEN
                 strresult := strresult || ' You are a bozo.';
              ELSIF numtimes > 1000 THEN
                 strresult := strresult || ' I do not know who you think you are. You are way out of control.';
              END IF;
           END IF;
           RETURN strresult;
        END;
$$
LANGUAGE plpgsql IMMUTABLE;
```

Functions (PL/pgSQL)

Declarations

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Examples

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Example A

```
CREATE FUNCTION sum_n_product(x int, y int,
OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Example B

```
CREATE FUNCTION
extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY
        SELECT quantity, quantity * price
        FROM sales
            WHERE itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

Functions (PL/pgSQL)

Returning From a Function

- RETURN expression;
- RETURN NEXT expression;
- RETURN QUERY query;
- RETURN QUERY EXECUTE command-string [USING expression [, ...]];

Return expression

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
   t2_row table2%ROWTYPE;
BEGIN
   SELECT * INTO t2_row FROM table2 WHERE ... ;
   RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

Return Query

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
  RETURN QUERY
        SELECT flightid
        FROM flight
        WHERE flightdate >= $1 AND flightdate < ($1 + 1);
END
$BODY$ LANGUAGE plpgsql;
-- Returns available flights
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Functions (PL/pgSQL)

Return using next

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
   r foo%rowtype;
BEGIN
   FOR r IN SELECT * FROM foo   WHERE fooid > 0
   LOOP
      -- can do some processing here
      – ...
      RETURN NEXT r; -- return current row of SELECT
   END LOOP;
   RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM getallfoo();
```

Functions (plpgsql)

Conditionals

```
IF boolean-expression THEN
    statements
END IF;

IF boolean-expression THEN
    statements
ELSE
    statements
END IF;

IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

Examples

```
IF vcount = 1 THEN
    INSERT INTO users_count (count) VALUES (vcount);
    RETURN 'insert done';
ELSE
    UPDATE users_count set count = vcount;
    RETURN 'update done';
END IF;
```

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
```

Functions (plpgsql)

Conditionals

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Simple Case

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

Searched Case

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and
      twenty';
END CASE;
```

Functions (PL/pgSQL)

Loops

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];

EXIT [ label ] [ WHEN boolean-expression ];
CONTINUE [ label ] [ WHEN boolean-expression ];

[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];

[ <<label>> ]
FOR name IN [ REVERSE ] expression ..
expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

```
WHILE amount_owed > 0 AND balance > 0 LOOP
    -- some computations here
END LOOP;
```

```
<<loop1>>
LOOP
   x := x+1;
       <<loop2>>
       LOOP
              y := y+1;
              EXIT loop1 WHEN y > 10;
       END LOOP;
   EXIT loop1 WHEN x > 100;
END LOOP;
```

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10
    – within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

Functions (plpgsql)

Looping through query results

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

Looping through arrays

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY
expression LOOP
    statements
END LOOP [ label ];
```

```
CREATE FUNCTION sum(int[]) RETURNS int AS $$
DECLARE
  s int := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION for_loop_through_query(n  INTEGER )
RETURNS VOID AS $$
DECLARE
   rec RECORD;
BEGIN
   FOR rec IN SELECT title FROM film ORDER BY title LIMIT n
   LOOP
                RAISE NOTICE '%', rec.title;
   END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Functions (PL/pgSQL)

- **Handling Errors:** By default, any error occurring in a PL/pgSQL function aborts execution of the function. Errors can be trapped and recover from them by using a BEGIN block with an EXCEPTION clause.

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

```
INSERT INTO mytab(firstname, lastname)
VALUES('Tom', 'Jones');

BEGIN
    UPDATE mytab SET firstname = 'Joe'
    WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
    EXCEPTION
    WHEN division_by_zero THEN
     -- or WHEN SQLSTATE '22012' THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

- postgreSQL error codes at postgresql. Documentation at:
  https://www.postgresql.org/docs/13/errcodes-appendix.html