PostgreSQL
Functions – Triggers

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 (2)

- Advantages of using PostgreSQL stored procedures
  - 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 procedures
  - 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 * FROM 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 varchar[]) RETURNS varchar 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

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 (plpgsql)

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);
         -- Since execution is not finished, we can check whether rows were returned -- and raise exception if not.
         IF NOT FOUND THEN
                  RAISE EXCEPTION 'No flight at %.', $1;
         END IF;
         RETURN;
END
$BODY$ LANGUAGE plpgsql;
-- Returns available flights or raises exception if there are no -- available flights.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Functions (plpgsql)

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;
```

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

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is
null
    result := 'NULL';
END IF;
```

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 FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE mviews RECORD;
BEGIN
RAISE NOTICE 'Refreshing materialized views...';
FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP
-- Now "mviews" has one record from cs_materialized_views
RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mviews.mv_name);
EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
EXECUTE 'INSERT INTO '
                   || quote_ident(mviews.mv_name) || ' '
                   || mviews.mv_query;
END LOOP;
RAISE NOTICE 'Done refreshing materialized views.';
RETURN 1; END; $$ LANGUAGE plpgsql;
```

Functions (plpgsql)

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 ];
```

```
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)

Handling errors

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

* postgreSQL error codes at postgresql
  Documentation:
 https://www.postgresql.org/docs/10/static/errcodes-appendix.html

```
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
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

Functions (plpgsql)

Declaring Cursor Variables

```
DECLARE
    curs1 refcursor; --unbound
    curs2 CURSOR FOR SELECT * FROM tenk1; --bound
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key; --bound
```

Opening Cursors

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string  [ USING expression [, ... ] ];
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey; --unbound

OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname) || ' WHERE col1 = $1' USING
keyvalue; --unbound

OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);

DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

Functions (plpgsql)

Using Cursors

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
MOVE [ direction { FROM | IN } ] cursor;
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
CLOSE cursor;
```

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

Returning Cursors

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
   OPEN $1 FOR SELECT col FROM test;
   RETURN $1;
END;
' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

Functions (plpgsql)

Looping Through a Cursor's Result

```
FOR my_rec IN curs2 LOOP
        Statements …
        Statements …
END LOOP
```

The following example shows one way to return multiple cursors from a single function

```
CREATE OR REPLACE FUNCTION myfunc(refcursor, refcursor)  RETURNS SETOF refcursor AS
$BODY$
BEGIN
        OPEN $1 FOR SELECT * FROM film;
        RETURN NEXT $1;
        OPEN $2 FOR SELECT * FROM category;
        RETURN NEXT $2;
END;
$BODY$  LANGUAGE plpgsql VOLATILE



-- need to be in a transaction to use cursors.
BEGIN;
  SELECT * FROM myfunc('a', 'b');

  FETCH ALL FROM a;
  FETCH ALL FROM b;
COMMIT;
```

Triggers

- Simply stated, a trigger is a piece of code that is executed in response to a data modification statement; that is, an insert, update, or delete.
- Triggers are event driven specialized procedures that are stored in, and managed by the RDBMS.
- Each trigger is attached to a single, specified table.
- Triggers can be thought of as an advanced form of “rule” or "constraint" written using SQL.
- A trigger can not be directly called or executed; it is automatically executed (or "fired") by the RDBMS as the result of an action (a data modification to the associated table)
- Once a trigger is created it is always executed when its "firing" event occurs

Why Use Triggers?

- Triggers are useful for implementing code that must be executed on a regular basis due to a pre-defined event.
- By utilizing triggers, scheduling and data integrity problems can be eliminated because the trigger will be fired whenever the triggering event occurs.
- Using triggers can be used for implementing business rules and constraints in the DBMS.
  - This is important because having the business rules in the database ensures that everyone uses the same logic to accomplish the same process.
- Triggers can be coded to access and/or modify other tables, log informational messages, and specify complex restrictions.

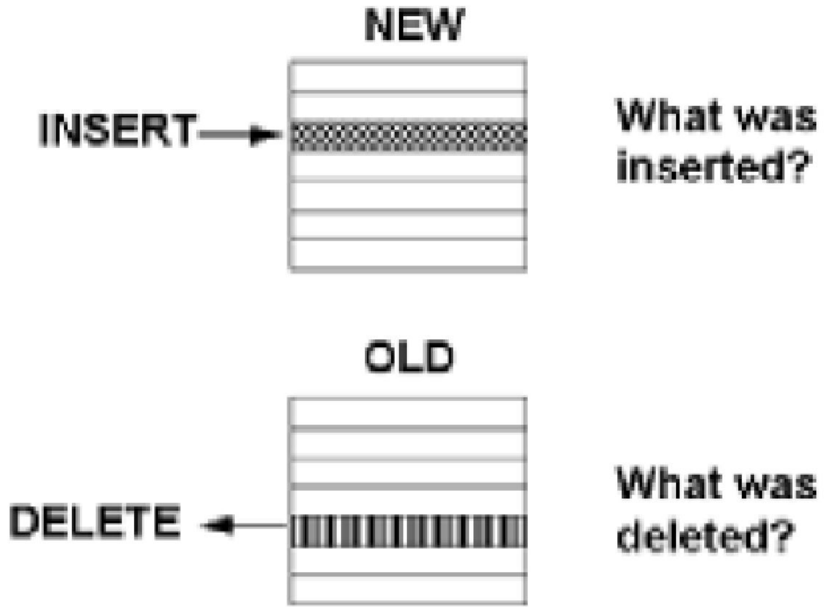
When does a Trigger fire?

- Three options exists for when a trigger can fire: before the firing activity occurs or after the firing activity occurs
- Only for Views (In PostgreSQL) a trigger can be defined to fire instead of an update/Insert/delete event on Views

Trigger Views

- Each trigger can have one NEW view of the table and one OLD view of the table available.
- These "views" are accessible only from triggers.

Trigger Views



Create Trigger

```
CREATE TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:
    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Create Trigger

- A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.
- Both the WHEN clause and the trigger actions may access elements of the row being inserted, deleted or updated using references of the form NEW.column-name and OLD.column-name, where column-name is the name of a column from the table that the trigger is associated with.
- If a WHEN clause is supplied, the PostgreSQL statements specified are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the PostgreSQL statements are executed for all rows.
- If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
- Triggers are automatically dropped when the table that they are associated with is dropped.
- The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just tablename, not database.tablename.
- Trigger functions can be written using SQL or PL/pgSQL

Triggers

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

NEW:Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers and for DELETE operations.
OLD: Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers and for INSERT operations.
TG_NAME: Data type name; variable that contains the name of the trigger actually fired.
TG_WHEN: Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.
TG_LEVEL: Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.
TG_OP: Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.
TG_RELID: Data type oid; the object ID of the table that caused the trigger invocation.
TG_RELNAME: Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG_TABLE_NAME instead.
TG_TABLE_NAME: Data type name; the name of the table that caused the trigger invocation.
TG_TABLE_SCHEMA: Data type name; the name of the schema of the table that caused the trigger invocation.
TG_NARGS: Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.
TG_ARGV[]: Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to tg_nargs) result in a null value.

Triggers

```
CREATE TABLE emp (
    empname          text NOT NULL,
    salary            integer
);
```

```
CREATE TABLE emp_audit(
    operation          char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer );
```

```
CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;
```