TRIGGERS

In SQL Server, triggers are database objects, actually, a special kind of stored procedure, which “reacts” to certain actions we make in the database.

A PostgreSQL trigger is a function invoked automatically whenever an event such as insert, update, or delete occurs.

You can specify whether the trigger is invoked before or after an event. If the trigger is invoked before an event, it can skip the operation for the current row or even change the row being updated or inserted. In case the trigger is invoked after the event, all changes are available to the trigger.

A trigger function is similar to a regular user-defined function. However, a trigger function does not take any arguments and has a return value with the type trigger.

CREATE TRIGGER

The CREATE TRIGGER statement creates a new trigger. The following illustrates the basic syntax of the CREATE TRIGGER statement:

```
CREATE TRIGGER trigger_name
   {BEFORE | AFTER} { event }
   ON table_name
   [FOR [EACH] { ROW | STATEMENT }]
       EXECUTE PROCEDURE trigger_function
```

In this syntax:

First, specify the name of the trigger after the TRIGGER keywords.

Second, specify the timing that cause the trigger to fire. It can be BEFORE or AFTER an event occurs.

Third, specify the event that invokes the trigger. The event can be INSERT , DELETE, UPDATE or TRUNCATE.

Fourth, specify the name of the table associated with the trigger after the ON keyword.

Fifth, specify the type of triggers which can be:

- Row-level trigger that is specified by the FOR EACH ROW clause.
- Statement-level trigger that is specified by the FOR EACH STATEMENT clause.

A row-level trigger is fired for each row while a statement-level trigger is fired for each transaction.

Suppose a table has 100 rows and two triggers that will be fired when a DELETE event occurs.

If the DELETE statement deletes 100 rows, the row-level trigger will fire 100 times, once for each deleted row. On the other hand, a statement-level trigger will be fired for one time regardless of how many rows are deleted.

Finally, specify the name of the trigger function after the EXECUTE PROCEDURE keywords.

The following statement creates a new table called employees:

```
DROP TABLE IF EXISTS employees;

CREATE TABLE employees(
   id INT GENERATED ALWAYS AS IDENTITY,
   first_name VARCHAR(40) NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   PRIMARY KEY(id)
);
```

Suppose that when the name of an employee changes, you want to log the changes in a separate table called employee_audits :

```
CREATE TABLE employee_audits (
   id INT GENERATED ALWAYS AS IDENTITY,
   employee_id INT NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   changed_on TIMESTAMP(6) NOT NULL
);
```

First, create a new function called log_last_name_changes:

```
CREATE OR REPLACE FUNCTION log_last_name_changes()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
        IF NEW.last_name <> OLD.last_name THEN
                INSERT INTO employee_audits(employee_id,last_name,changed_on)
                VALUES(OLD.id,OLD.last_name,now());
        END IF;

        RETURN NEW;
END;
$$
```

The function inserts the old last name into the employee_audits table including employee id, last name, and the time of change if the last name of an employee changes.

The OLD represents the row before update while the NEW represents the new row that will be updated. The OLD.last_name returns the last name before the update and the NEW.last_name returns the new last name.

Second, bind the trigger function to the employees table. The trigger name is last_name_changes. Before the value of the last_name column is updated, the trigger function is automatically invoked to log the changes.

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Third, insert some rows into the employees table:

```
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe');

INSERT INTO employees (first_name, last_name)
VALUES ('Lily', 'Bush');
```

Fourth, examine the contents of the employees table:

```
SELECT * FROM employees;
```

	id integer	first_name character varying (40)	last_name character varying (40)
1	1	John	Doe
2	2	Lily	Bush

Suppose that Lily Bush changes her last name to Lily Brown.

Fifth, update Lily's last name to the new one:

```
UPDATE employees
SET last_name = 'Brown'
WHERE ID = 2;
```

Seventh, check if the last name of Lily has been updated:

```
SELECT * FROM employees;
```

	id integer	first_name character varying (40)	last_name character varying (40)
1	1	John	Doe
2	2	Lily	Brown

As you can see from the output, Lily's last name has been updated.

Eighth, verify the contents of the employee_audits table:

```
SELECT * FROM employee_audits;
```

The change was logged in the employee_audits table by the trigger.

In this tutorial, you have learned how to use the PostgreSQL CREATE TRIGGER to create a new trigger.

In the above trigger function there is new keyword '**NEW**' which is a PostgreSQL extension to triggers. There are two PostgreSQL extensions to trigger '**OLD**' and '**NEW**'. OLD and NEW are not case sensitive.

- Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger
- In an INSERT trigger, only NEW.col_name can be used.
- In a UPDATE trigger, you can use OLD.col_name to refer to the columns of a row before it is updated and NEW.col_name to refer to the columns of the row after it is updated.
- In a DELETE trigger, only OLD.col_name can be used; there is no new row.

A column named with OLD is read only. You can refer to it (if you have the SELECT privilege), but not modify it. You can refer to a column named with NEW if you have the SELECT privilege for it. In a BEFORE trigger, you can also change its value with SET NEW.col_name = value if you have the UPDATE privilege for it. This means you can

use a trigger to modify the values to be inserted into a new row or used to update a row. (Such a SET statement has no effect in an AFTER trigger because the row change will have already occurred.)

Sample database, table, table structure, table records for various examples

Records of the table (on some fields): **emp_details**

```
postgres=#  SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY, COMMISSION_PCT FROM emp_d
 employee_id | first_name  | last_name |   job_id   |  salary  | commission_pct
-------------+-------------+-----------+------------+----------+----------------
         100 | Steven      | King      | AD_PRES    | 24000.00 |           0.00
         101 | Neena       | Kochhar   | AD_VP      | 17000.00 |           0.00
         102 | Lex         | De Haan   | AD_VP      | 17000.00 |           0.00
         103 | Alexander   | Hunold    | IT_PROG    |  9000.00 |           0.00
         104 | Bruce       | Ernst     | IT_PROG    |  6000.00 |           0.00
         105 | David       | Austin    | IT_PROG    |  4800.00 |           0.00
         106 | Valli       | Pataballa | IT_PROG    |  4800.00 |           0.00
         107 | Diana       | Lorentz   | IT_PROG    |  4200.00 |           0.00
         108 | Nancy       | Greenberg | FI_MGR     | 12000.00 |           0.00
         109 | Daniel      | Faviet    | FI_ACCOUNT |  9000.00 |           0.00
         110 | John        | Chen      | FI_ACCOUNT |  8200.00 |           0.00
         111 | Ismael      | Sciarra   | FI_ACCOUNT |  7700.00 |           0.00
         112 | Jose Manuel | Urman     | FI_ACCOUNT |  7800.00 |           0.00
(13 rows)
```

PostgreSQL Trigger: Example AFTER INSERT

In the following example we have two tables : emp_details and emp_log. To insert some information into the emp_logs table (which have three fields emp_id and salary and edttime). Every time, when an INSERT happen into emp_details table we have used the following trigger :

First a trigger function has to be created. Here is the trigger function rec_insert()

```
CREATE OR REPLACE FUNCTION rec_insert()
  RETURNS trigger AS
$$
BEGIN
         INSERT INTO emp_log(emp_id,salary,edittime)
         VALUES(NEW.employee_id,NEW.salary,current_date);

    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
```

Here is the trigger ins_same_rec:

```
CREATE TRIGGER ins_same_rec
  AFTER INSERT
  ON emp_details
  FOR EACH ROW
  EXECUTE PROCEDURE rec_insert();

```

We already have some details in emp_details and emp_log table, but let us insert a new record now

```
INSERT INTO emp_details VALUES(236, 'RABI', 'CHANDRA', 'RABI',
'590.423.45700', '2013-01-12', 'AD_VP', 15000, .5);
```

Record is shown in the last column and at the same time triggers the ins_same_rec

```
postgres=#  SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY, COMMISSION_PCT FROM emp_details;
 employee_id | first_name  | last_name |   job_id   |  salary  | commission_pct
-------------+-------------+-----------+------------+----------+----------------
         100 | Steven      | King      | AD_PRES    | 24000.00 |           0.00
         101 | Neena       | Kochhar   | AD_VP      | 17000.00 |           0.00
         102 | Lex         | De Haan   | AD_VP      | 17000.00 |           0.00
         103 | Alexander   | Hunold    | IT_PROG    |  9000.00 |           0.00
         104 | Bruce       | Ernst     | IT_PROG    |  6000.00 |           0.00
         105 | David       | Austin    | IT_PROG    |  4800.00 |           0.00
         106 | Valli       | Pataballa | IT_PROG    |  4800.00 |           0.00
         107 | Diana       | Lorentz   | IT_PROG    |  4200.00 |           0.00
         108 | Nancy       | Greenberg | FI_MGR     | 12000.00 |           0.00
         109 | Daniel      | Faviet    | FI_ACCOUNT |  9000.00 |           0.00
         110 | John        | Chen      | FI_ACCOUNT |  8200.00 |           0.00
         111 | Ismael      | Sciarra   | FI_ACCOUNT |  7700.00 |           0.00
         112 | Jose Manuel | Urman     | FI_ACCOUNT |  7800.00 |           0.00
         236 | RABI        | CHANDRA   | AD_VP      | 15000.00 |           0.50
(14 rows)

postgres=#  SELECT * FROM emp_log;
 emp_id | salary |  edittime
--------+--------+------------
    100 |  24000 | 2011-01-15
    101 |  17000 | 2010-01-12
    102 |  17000 | 2010-09-22
    103 |   9000 | 2011-06-21
    104 |   6000 | 2012-07-05
    105 |   4800 | 2011-06-02
    236 |  15000 | 2014-09-15
(7 rows)
```

DROP TRIGGER

To delete a trigger from a table you use the drop trigger statement with the following syntax

```
DROP TRIGGER [IF EXISTS] trigger_name
ON table_name [ CASCADE | RESTRICT ];
```

Use the CASCADE option if you want to drop objects that depend on the trigger automatically. Note that CASCADE option will also delete objects that depend on objects that depend on the trigger.

Use the RESTRICT option to refuse to drop the trigger if any objects depend on it. By default, the DROP TRIGGER statement uses RESTRICT.

Note that in SQL standard, trigger names are not local to tables so the statement is simply:

```
DROP TRIGGER trigger_name;
```

Example use the DROP TRIGGER statement to delete the username_check trigger:

```
DROP TRIGGER username_check
ON staff;
```

ALTER TRIGGER

The ALTER TRIGGER statement allows you to rename a trigger. The following shows the syntax of the ALTER TRIGGER statement:

```
ALTER TRIGGER trigger_name
ON table_name
RENAME TO new_trigger_name;
```

Example use the ALTER TRIGGER statement to rename the before_update_salary trigger to salary_before_update:

```
ALTER TRIGGER before_update_salary
ON employees
RENAME TO salary_before_update;
```

If you use psql tool, you can view all triggers associated with a table using the \dS command:

```
\dS employees
```

DISABLE TRIGGER

To disable a trigger, you use the ALTER TABLE DISABLE TRIGGER statement:

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL
```

Suppose you want to disable the trigger associated with the employeestable, you can use the following statement:

```
ALTER TABLE employees
DISABLE TRIGGER log_last_name_changes;
```

To disable all triggers associated with the employees table, you use the following statement:

```
ALTER TABLE employees
DISABLE TRIGGER ALL;
```

ENABLING TRIGGER

To enable a trigger or all triggers associated with a table, you use the ALTER TABLE ENABLE TRIGGER statement:

```
ALTER TABLE table_name
ENABLE TRIGGER trigger_name |  ALL;
```

The following statement enables the salary_before_update trigger on the employees table:

```
ALTER TABLE employees
ENABLE TRIGGER salary_before_update;
```

The following example enables all triggers that belong to the employees table:

```
ALTER TABLE employees
ENABLE TRIGGER ALL;
```