09. Programming Databases

Triggers

[abstracts from postgresql documentation]

What is a trigger?

- A trigger is a special type of procedure that the database should automatically execute on some event (database update events)
- Triggers can be defined to execute either before or after any INSERT, UPDATE, or DELETE operation.
- If a trigger event occurs, the trigger's function is called.

Types of Triggers

Trigger could be **row-level-trigger** or **statement-level-trigger**

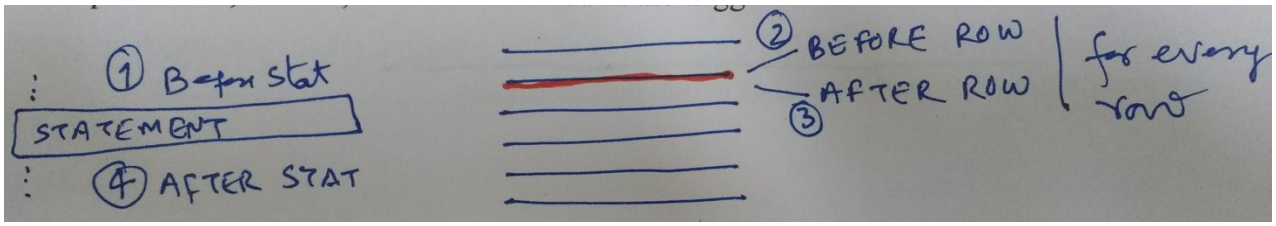
- **Row-level-trigger**. Note that an update statement on database can affect 0 or n rows. A row level triggered is invoked once for each row that is affected by the statement that fired the trigger.
- In contrast, **statement-level-trigger** is invoked only once when an appropriate statement is executed, regardless of the number of rows affected by that statement. A statement-level-trigger is executed even if it affects zero rows.

Triggers are also classified as ***before* triggers** and ***after* triggers**.

- **Statement-level before triggers** naturally fire before the statement starts to do anything, while **statement-level after triggers** fire at the very end of the statement.
- **Row-level before triggers** fire immediately before a particular row is operated on, while **row-level after triggers** fire after working on the row (but before any statement-level after triggers).

Trigger Execution Cycle

- Trigger is invoked on event of performing an update (insert/update/delete) operation on a relation (or on a view).
- Before triggers are fired before executing the operation. Also returns true/false, that is used to decide if the operation is to be carried out or not.
- If the trigger is row level it is executed for every affected row.
- After triggers are fired after executing the operation on affected row (if it is row level trigger), or after finishing the operation on all rows, if it is a statement level trigger.



Application examples of triggers

- **Row Level Before trigger** can typically be used to enforce complex data validation rules before adding/modifying/deleting a tuple (row). For example a student cannot take more than 22 credits in a semester, or selection of courses etc; it can be used to implement complex constraints like this.
- Another example is "save old values before making any change in a row", or for logging events that are happening on every row of database.
- **Row Level After**: Can be used to make additional updates, for example computing some derived attribute, or updating other attributes for example updating stock on sales
- **Statement Level Before**: Can be to ensure "assertions", do some initialization
- **Statement Level After**: Some updates may not be done at row level, instead done after statement for example posting into account ledger on sale or purchase; sending some alerts, etc.
- For some cleaning, for example deletion after copying or so.
- Note that many of these works can also be done in host programming languages, but rules that are found to be associated with the databases only, irrespective of application; in that case **triggers are better option**.

Creating Triggers

- A trigger description contains three parts: Event, Condition, and Action
- **Event**: A database event that activates the trigger; for example "UPDATing of relation EMPLOYEE"
- **When** (one of following):
    - Before Statement,
    - Before Row
    - After Statement
    - After Row
- **Action**: A code that is to be executed when *event* occurs
- Create a *trigger function* with CREATE FUNCTION command. The function is declared with no arguments and a return type of trigger.
- Trigger function must be declared with no arguments.
  [You can however pass parameters to a trigger function, and are accessed through **TG_ARGV** special variable, automatically creating and available in procedure body]
- Finally use CREATE TRIGGER command to define a trigger, where you attach an already created function (as said above) as action part of trigger
-

Special variable available in trigger functions

- 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.
- **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.
- **TG_NAME**: (Trigger Name). Data type text; variable that contains the name of the trigger actually fired.
- **TG_WHEN**: Data type text; a string of either BEFORE or AFTER 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**: (Trigger Operation). Data type text; a string of INSERT, UPDATE, or DELETE telling for which operation the trigger was fired.
- More are-
    - **TG_RELID**: (Relation ID) Data type oid; the object ID of the table that caused the trigger invocation.
    - **TG_TABLE_NAME**:
    - **TG_TABLE_SCHEMA**:
    - **TG_NARGS**: the number of arguments given to the trigger procedure
    - **TG_ARGV**: Data type array of text; the arguments from the CREATE TRIGGER statement, start from 0.

Example ##: Here is an example using these variables

Following action could be specified when qty of saved sales is modified.

- On Table: InvoiceDetails
- Event: UPDATE
- Type: AFTER UPDATE FOR EACH ROW
- Action: **UPDATE stock SET qty = qty + OLD.qty - NEW.qty**

Example ##: A trigger based logger (from postgresql documentation)

Consider a table EMP here

```
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
);
```

Define a trigger as

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

Here is trigger function

```
CREATE OR REPLACE FUNCTION process_emp_audit()
      RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation perf
        -- make use of the special variable TG_OP to work out the
        --
        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 t
    END;
$emp_audit$ LANGUAGE plpgsql;
```

INSTEAD OF trigger

- Instead of trigger is invoked "instead of performing the operation". That is requested operation is never performed; instead, the action defined in the trigger is executed.
- INSTEAD OF trigger specifies that *action* is to be executed instead of activating event.
- This trigger, normally defined to make the view updatable; when an attempt is made to INSERT/UPDATE/DELETE onto a view, alternate action is taken instead of actually performing the event on the view

Example ##: sourced from a PostgreSQL book[1]:

Consider following table EMPZ and a EMPV view defined.

EMPZ

```
 emp_id | emp_name | emp_city
--------+----------+----------
      1 | Adam     | Chicago
      2 | John     | Miami
      3 | Smith    | Dallas
```

EMPV

```
CREATE VIEW EMPV AS
        SELECT * FROM EMPZ;
```

To make the view updatable we define an INSTEAD of trigger as following-

```
CREATE FUNCTION triggerfunc_on_empv() RETURNS trigger AS $$
BEGIN
        IF (TG_OP = 'INSERT') THEN
                INSERT INTO EMPZ VALUES (NEW.emp_id, NEW.emp_name, NEW.emp_city);
                RETURN NEW;
        END IF;
        RETURN NULL;
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trigger_on_empv INSTEAD OF INSERT ON EMPV
        FOR EACH ROW EXECUTE PROCEDURE triggerfunc_on_empv();
```

Following INSERT statement into EMPV will INSTEAD insert a row into EMPZ!

```
INSERT INTO EMPV VALUES (4, 'Gary', 'Houston');
```

More reading from PostgreSQL documentation for PL/pgSQL
http://intranet.daiict.ac.in/~pm_jat/postgres/html/plpgsql.html

[1] Ahmed, Ibrar, Asif Fayyaz, and Amjad Shahzad. *PostgreSQL Developer's Guide*. Packt Publishing Ltd, 2015.