Introduction to SQL Trigger

A SQL trigger is a set of SQL statements stored in the database catalog. A SQL trigger is executed or fired whenever an event associated with a table occurs e.g., insert, update or delete.

A SQL trigger is a special type of stored procedure. It is special because it is not called directly like a stored procedure. The main difference between a trigger and a stored procedure is that a trigger is called automatically when a data modification event is made against a table whereas a stored procedure must be called explicitly.

In MySQL, a trigger is a set of SQL statements that is invoked automatically when a change is made to the data on the associated table. You can to define maximum six triggers for each table.

- `BEFORE INSERT` – activated before data is inserted into the table.
- `AFTER INSERT` – activated after data is inserted into the table.
- `BEFORE UPDATE` – activated before data in the table is updated.
- `AFTER UPDATE` – activated after data in the table is updated.
- `BEFORE DELETE` – activated before data is removed from the table.
- `AFTER DELETE` – activated after data is removed from the table.

When you use a statement that does not use `INSERT`, `DELETE` or `UPDATE`statement to change data in a table, the triggers associated with the table are not invoked. For example, the TRUNCATE statement removes all data of a table but does not invoke the trigger associated with that table.

There are some statements that use the `INSERT` statement behind the scenes such as REPLACE statement or LOAD DATA statement. If you use these statements, the corresponding triggers associated with the table are invoked.

You must use a unique name for each trigger associated with a table. However, you can have the same trigger name defined for different tables though it is a good practice.

You should name the triggers using the following naming convention:

(BEFORE/ AFTER)_TABLENAME_(INSERT/UPDATE/DELETE)

For example, `before_order_update` is a trigger invoked before a row in the `order`table is updated.
The following naming convention is as good as the one above.

TABLENAME_(BEFORE/ AFTER)_ (INSERT/UPDATE/DELETE)

For example, `order_before_update` is the same as `before_update_update` trigger above.

MySQL trigger syntax

In order to create a new trigger, you use the `CREATE TRIGGER` statement. The following illustrates the syntax of the `CREATE TRIGGER` statement:

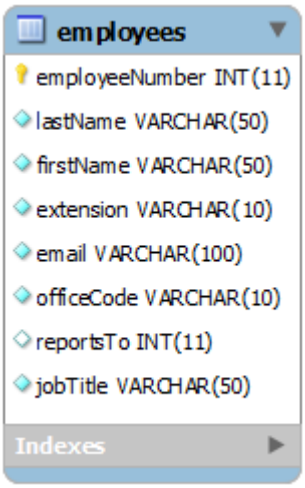
```
CREATE TRIGGER trigger_name trigger_time trigger_event
 ON table_name
 FOR EACH ROW
 BEGIN
 ...
 END;
```

Let's examine the syntax above in more detail.

- You put the trigger name after the `CREATE TRIGGER` statement. The trigger name should follow the naming convention `[trigger time]_[table name]_[trigger event]`, for example `before_employees_update`.
- Trigger activation time can be `BEFORE` or `AFTER`. You must specify the activation time when you define a trigger. You use the `BEFORE` keyword if you want to process action prior to the change is made on the table and `AFTER` if you need to process action after the change is made.
- The trigger event can be `INSERT`, `UPDATE` or `DELETE`. This event causes the trigger to be invoked. A trigger only can be invoked by one event. To define a trigger that is invoked by multiple events, you have to define multiple triggers, one for each event.
- A trigger must be associated with a specific table. Without a table trigger would not exist therefore you have to specify the table name after the `ON` keyword.
- You place the SQL statements between `BEGIN` and `END` block. This is where you define the logic for the trigger.

MySQL trigger example

Let's start creating a trigger in MySQL to log the changes of the `employees` table.



First, create a new table named `employees_audit` to keep the changes of the `employee` table. The following statement creates the `employee_audit` table.

```
CREATE TABLE employees_audit (
    id INT AUTO_INCREMENT PRIMARY KEY,
    employeeNumber INT NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
```

Next, create a `BEFORE UPDATE` trigger that is invoked before a change is made to the `employees` table

```
DELIMITER $$
CREATE TRIGGER before_employee_update
    BEFORE UPDATE ON employees
    FOR EACH ROW
BEGIN
    INSERT INTO employees_audit
    SET action = 'update',
     employeeNumber = OLD.employeeNumber,
        lastname = OLD.lastname,
        changedat = NOW();
END$$
DELIMITER ;
```

Inside the body of the trigger, we used the `OLD` keyword to access `employeeNumber` and `lastname`column of the row affected by the trigger. Notice that in a trigger defined for INSERT, you can use `NEW` keyword only. You cannot use the `OLD`keyword. However, in the trigger defined for DELETE, there is no new row so you can use the `OLD`keyword only. In the UPDATE trigger, `OLD` refers to the row before it is updated and `NEW` refers to the row after it is updated.

Then, to view all triggers in the current database, you use `SHOW TRIGGERS` statement as follows:

```
SHOW TRIGGERS;
```

Trigger	Event	Table	Statement	Timing	Created	sql_mode	Definer
before_employee_update	UPDATE	employees	BEGIN INSERT INTO employ...	BEFORE	2015-11-14 21:39:09.08	STRICT_TRANS_TABLES,NO_AUTO_CREATE_U...	root@localhost

After that, update the `employees` table to check whether the trigger is invoked.

```
UPDATE employees
SET
    lastName = 'Phan'
WHERE
    employeeNumber = 1056;
```

Finally, to check if the trigger was invoked by the `UPDATE` statement, you can query the `employees_audit` table using the following query:

```
SELECT
    *
FROM
    employees_audit;
```

The following is the output of the query:

	id	employeeNumber	lastname	changedat	action
▶	1	1056	Phan	2015-11-14 21:39:12	update

As you see, the trigger was really invoked and it inserted a new row into the `employees_audit` table.