MySQLTUTORIAL

MySQL AFTER UPDATE Trigger

Summary: in this tutorial, you will learn how to create a MySQL `AFTER UPDATE` trigger to log the changes made to a table.

Introduction to MySQL AFTER UPDATE triggers

MySQL `AFTER UPDATE` triggers (https://www.mysqltutorial.org/mysql-triggers.aspx) are invoked automatically after an update (https://www.mysqltutorial.org/mysql-update-data.aspx) event occurs on the table associated with the triggers.

The following shows the syntax of creating a MySQL `AFTER UPDATE` trigger:

```
CREATE TRIGGER trigger_name
AFTER UPDATE
ON table_name FOR EACH ROW
trigger_body
```

In this syntax:

First, specify the name of the trigger that you want to create in the `CREATE TRIGGER` (https://www.mysqltutorial.org/create-the-first-trigger-in-mysql.aspx) clause.

Second, use `AFTER UPDATE` clause to specify the time to invoke the trigger.

Third, specify the name of the table to which the trigger belongs after the `ON` keyword.

Finally, specify the trigger body which consists of one or more statements.

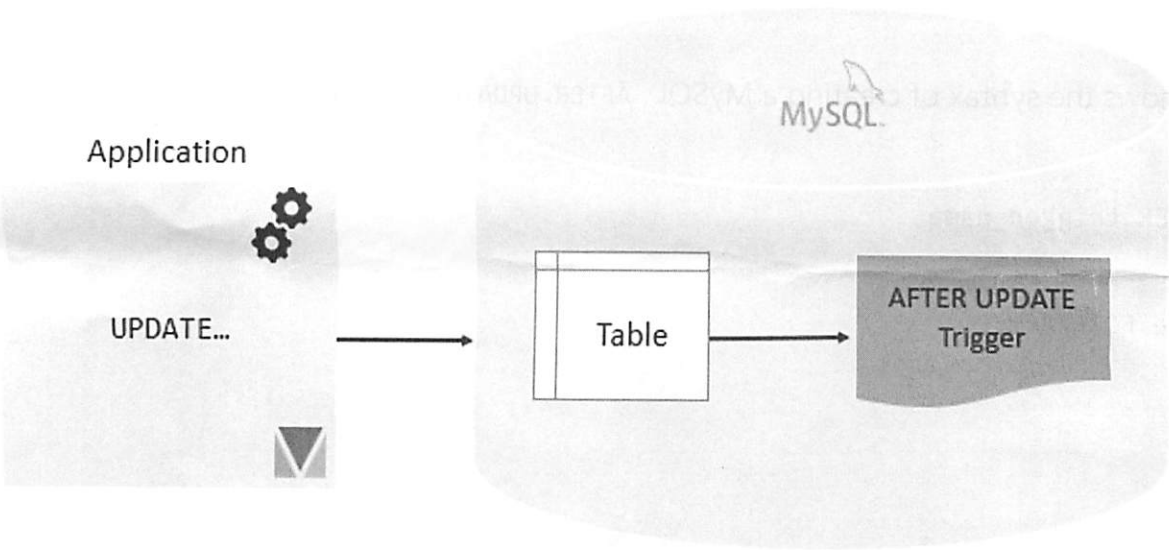
If the trigger body has more than one statement, you need to use the `BEGIN END` block. And, you also need to change the default delimiter (https://www.mysqltutorial.org/mysql-stored-procedure/mysql-delimiter/) as shown in the following code:

```
DELIMITER $$

CREATE TRIGGER trigger_name
    AFTER UPDATE
    ON table_name FOR EACH ROW
BEGIN
    -- statements
END$$

DELIMITER ;
```

In a `AFTER UPDATE` trigger, you can access `OLD` and `NEW` rows but cannot update them.



MySQL AFTER UPDATE trigger example

Let's look at an example of creating a `AFTER UPDATE` trigger.

Setting up a sample table

First, create a table (https://www.mysqltutorial.org/mysql-create-table/) called `Sales` :

```
DROP TABLE IF EXISTS Sales;

CREATE TABLE Sales (
```

```
    id INT AUTO_INCREMENT,
    product VARCHAR(100) NOT NULL,
    quantity INT NOT NULL DEFAULT 0,
    fiscalYear SMALLINT NOT NULL,
    fiscalMonth TINYINT NOT NULL,
    CHECK(fiscalMonth >= 1 AND fiscalMonth <= 12),
    CHECK(fiscalYear BETWEEN 2000 and 2050),
    CHECK (quantity >=0),
    UNIQUE(product, fiscalYear, fiscalMonth),
    PRIMARY KEY(id)
);
```

Second, insert sample data (https://www.mysqltutorial.org/mysql-insert-multiple-rows/) into the `Sales` table:

```
INSERT INTO Sales(product, quantity, fiscalYear, fiscalMonth)
VALUES
    ('2001 Ferrari Enzo',140, 2021,1),
    ('1998 Chrysler Plymouth Prowler', 110,2021,1),
    ('1913 Ford Model T Speedster', 120,2021,1);
```

Third, query data (https://www.mysqltutorial.org/mysql-select-statement-query-data.aspx) from the `Sales` table to display its contents:

```
SELECT * FROM Sales;
```

id	product	quantity	fiscalYear	fiscalMonth
1	2001 Ferrari Enzo	140	2021	1
2	1998 Chrysler Plymouth Prowler	110	2021	1
3	1913 Ford Model T Speedster	120	2021	1

Finally, create a table (https://www.mysqltutorial.org/mysql-create-table/) that stores the changes in the `quantit y` column from the `sales` table:

```
DROP TABLE IF EXISTS SalesChanges;

CREATE TABLE SalesChanges (
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
    salesId INT,
    beforeQuantity INT,
    afterQuantity INT,
    changedAt TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Creating AFTER UPDATE trigger example

The following statement creates an `AFTER UPDATE` trigger on the `sales` table:

```
DELIMITER $$

CREATE TRIGGER after_sales_update
AFTER UPDATE
ON sales FOR EACH ROW
BEGIN
    IF OLD.quantity <> new.quantity THEN
        INSERT INTO SalesChanges(salesId,beforeQuantity, afterQuantity)
        VALUES(old.id, old.quantity, new.quantity);
    END IF;
END$$

DELIMITER ;
```

This `after_sales_update` trigger is automatically fired before an update event occurs for each row in the `sales` table.

If you update the value in the `quantity` column to a new value the trigger insert a new row to log the changes in the `SalesChanges` table.

Let's examine the trigger in detail:

First, the name of the trigger is `after_sales_update` specified in the `CREATE TRIGGER` (https://www.mysqltutorial.org/create-the-first-trigger-in-mysql.aspx) clause:

```
CREATE TRIGGER after_sales_update
```

Second, the triggering event is:

```
AFTER UPDATE
```

Third, the table that the trigger associated with is `sales` :

```
ON Sales FOR EACH ROW
```

Finally, use the `IF-THEN` (https://www.mysqltutorial.org/mysql-if-statement/) statement inside the trigger body to check if the new value is not the same as the old one, then insert the changes into the `SalesChanges` table:

```
IF OLD.quantity <> new.quantity THEN
    INSERT INTO SalesChanges(salesId,beforeQuantity, afterQuantity)
    VALUES(old.id, old.quantity, new.quantity);
END IF;
```

Testing the MySQL AFTER UPDATE trigger

First, update (https://www.mysqltutorial.org/mysql-update-data.aspx) the quantity of the row with id 1 to 350:

```
UPDATE Sales
SET quantity = 350
WHERE id = 1;
```

The `after_sales_update` was invoked automatically.

Second, query data from the `SalesChanges` table:

```
SELECT * FROM SalesChanges;
```

id	salesId	beforeQuantity	afterQuantity	changedAt
1	1	140	350	2019-09-07 13:24:58

Third, increase the sales quantity of all rows to 10%:

```
UPDATE Sales
SET quantity = CAST(quantity * 1.1 AS UNSIGNED);
```

Fourth, query data from the `SalesChanges` table:

```
SELECT * FROM SalesChanges;
```

id	salesId	beforeQuantity	afterQuantity	changedAt
1	1	140	350	2019-09-07 13:24:58
2	1	350	385	2019-09-07 13:26:02
3	2	110	121	2019-09-07 13:26:02
4	3	120	132	2019-09-07 13:26:02

The trigger fired three times because of the updates of the three rows.

In this tutorial, you have learned how to create a MySQL `AFTER UPDATE` trigger to validate data before it is updated to a table.

POSTGRESQL TUTORIAL

PostgreSQL CREATE TRIGGER

Summary: in this tutorial, you will learn how to use the PostgreSQL `CREATE TRIGGER` statement to create a trigger.

To create a new trigger in PostgreSQL, you follow these steps:

- First, create a trigger function using `CREATE FUNCTION` (https://www.postgresqltutorial.com/postgresql-create-function/) statement.
- Second, bind the trigger function to a table by using `CREATE TRIGGER` statement.

> If you are not familiar with creating a user-defined function, you can check out the PL/pgSQL section (https://www.postgresqltutorial.com/postgresql-stored-procedures/) .

Create trigger function syntax

A trigger function is similar to a regular user-defined function (https://www.postgresqltutorial.com/postgresql-create-function/) . However, a trigger function does not take any arguments and has a return value with the type `trigger` .

The following illustrates the syntax of creating trigger function:

```
CREATE FUNCTION trigger_function()
   RETURNS TRIGGER
   LANGUAGE PLPGSQL
AS $$
BEGIN
   -- trigger logic
END;
```

```
$$
```

Notice that you can create a trigger function using any languages supported by PostgreSQL. In this tutorial, we will use PL/pgSQL.

A trigger function receives data about its calling environment through a special structure called TriggerData which contains a set of local variables.

For example, `OLD` and `NEW` represent the states of the row in the table before or after the triggering event.

PostgreSQL also provides other local variables preceded by `TG_` such as `TG_WHEN` , and `TG_TABLE_NAME` .

Once you define a trigger function, you can bind it to one or more trigger events such as `INSERT` (https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-insert/) , UPDATE (https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-update/) , and `DELETE` (https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-delete/) .

Introduction to PostgreSQL CREATE TRIGGER statement

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.

PostgreSQL CREATE TRIGGER example

The following statement create 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,chang
                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 (https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-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	first_name	last_name
	integer	character varying (40)	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;
```

	id integer	employee_id integer	last_name character varying (40)	changed_on timestamp without time zone
1	1	2	Bush	2020-07-30 17:29:04.248925

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.