MySQL Triggers

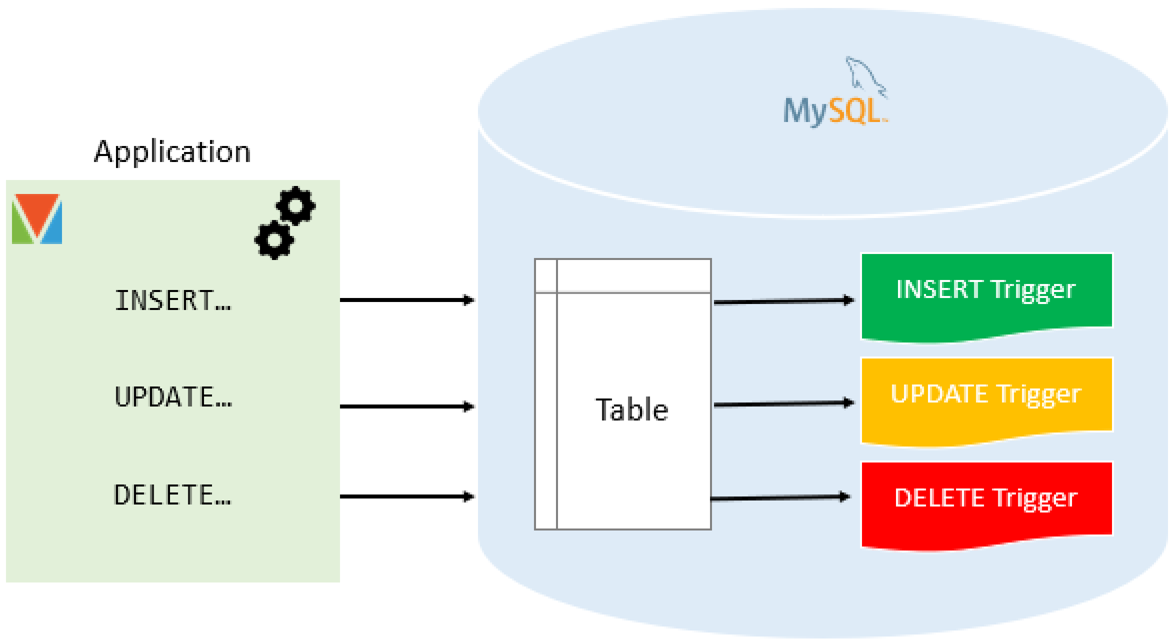
In MySQL, a trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table. For example, you can define a trigger that is invoked automatically before a new row is inserted into a table.

MySQL supports triggers that are invoked in response to the INSERT, UPDATE or DELETE event.

The SQL standard defines two types of triggers: **row-level triggers** and **statement-level triggers**.

- A row-level trigger is activated for each row that is inserted, updated, or deleted. For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected.
- A statement-level trigger is executed once for each transaction regardless of how many rows are inserted, updated, or deleted.

MySQL supports only row-level triggers. It doesn't support statement-level triggers.



Advantages of triggers

- Triggers provide another way to check the integrity of data.
- Triggers handle errors from the database layer.
- Triggers give an alternative way to run scheduled tasks. By using triggers, you don’t have to wait for the scheduled events to run because the triggers are invoked automatically *before* or after a change is made to the data in a table.
- Triggers can be useful for auditing the data changes in tables.

Disadvantages of triggers

- Triggers can only provide extended validations, not all validations. For simple validations, you can use the `NOT NULL`, `UNIQUE`, `CHECK` and `FOREIGN KEY` constraints.
- Triggers can be difficult to troubleshoot because they execute automatically in the database, which may not invisible to the client applications.
- Triggers may increase the overhead of the MySQL Server.

Create Trigger in MySQL

The `CREATE TRIGGER` statement creates a new trigger. Here is the basic syntax of the `CREATE TRIGGER` statement:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

In this syntax:

- First, specify the name of the trigger that you want to create after the `CREATE TRIGGER` keywords. Note that the trigger name must be unique within a database.
- Next, specify the trigger action time which can be either `BEFORE` or `AFTER` which indicates that the trigger is invoked before or after each row is modified.

- Then, specify the operation that activates the trigger, which can be INSERT, UPDATE, or DELETE.
- After that, specify the name of the table to which the trigger belongs after the ON keyword.
- Finally, specify the statement to execute when the trigger activates. If you want to execute multiple statements, you use the `BEGIN END` compound statement.

The trigger body can access the values of the column being affected by the DML statement.

To distinguish between the value of the columns BEFORE and AFTER the DML has fired, you use the NEW and OLD modifiers.

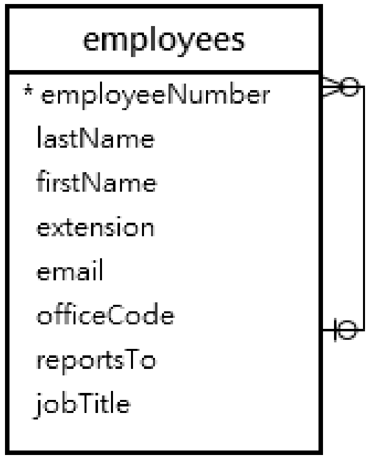
For example, if you update the column description, in the trigger body, you can access the value of the description before the update `OLD.description` and the new value `NEW.description`.

The following table illustrates the availability of the OLD and NEW modifiers:

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

MySQL trigger examples

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 to the employees 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.

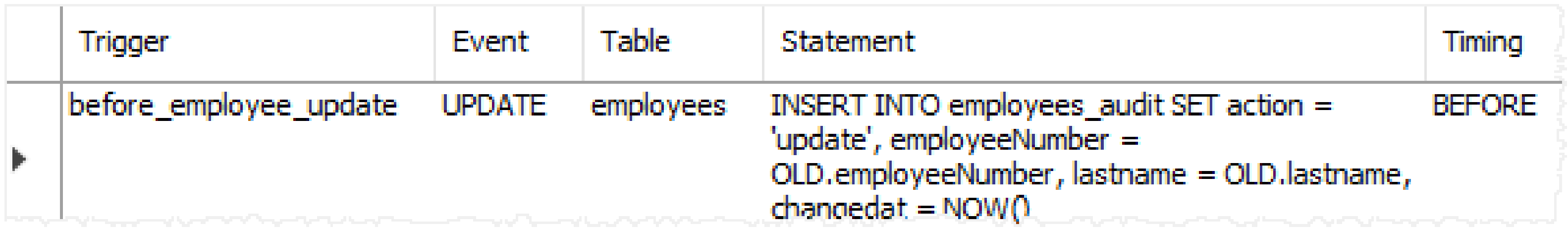
Next, create a `BEFORE UPDATE` trigger that is invoked before a change is made to the employees table.

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

Inside the body of the trigger, we used the OLD keyword to access values of the columns `employeeNumber` and `lastname` of the row affected by the trigger.

Then, show all triggers in the current database by using the `SHOW TRIGGERS` statement:

```
SHOW TRIGGERS;
```



Trigger	Event	Table	Statement	Timing
before_employee_update	UPDATE	employees	INSERT INTO employees_audit SET action = 'update', employeeNumber = OLD.employeeNumber, lastname = OLD.lastname, changedat = NOW()	BEFORE

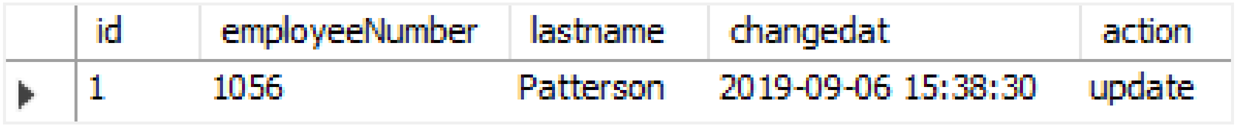
After that, update a row in the `employees` table:

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

Finally, query the `employees_audit` table to check if the trigger was fired by the `UPDATE` statement:

```
SELECT * FROM employees_audit;
```

The following shows the output of the query:



	id	employeeNumber	lastname	changedat	action
▶	1	1056	Patterson	2019-09-06 15:38:30	update

As you see clearly from the output, the trigger was automatically invoked and inserted a new row into the `employees_audit` table.