

Trigger

A trigger is a stored program (with queries) which is executed automatically to respond to a specific event such as insertion, updating or deletion occurring in a table.

Before Update Trigger

As the name implies, it is a trigger which enacts before an update is invoked. If we write an update statement, then the actions of the trigger will be performed before the update is implemented.

Considering tables:

```
create table customer (acc_no integer primary key, cust_name varchar(20),  
avail_balance decimal);
```

```
create table mini_statement (acc_no integer, avail_balance decimal, foreign key  
(acc_no) references customer(acc_no) on delete cascade);
```

Inserting values in them:

```
insert into customer values (1000, "Fanny", 7000);  
insert into customer values (1001, "Peter", 12000);
```

Trigger to insert (old) values into a mini_statement record (including account number and available balance as parameters) before updating any record in customer record/table:

```
delimiter //  
create trigger update_cus  
before update on customer  
  for each row  
  begin  
    insert into mini_statement values (old.acc_no, old.avail_balance);  
  end; //
```

Making updates to invoke trigger:

```
update customer set avail_balance = avail_balance + 3000 where acc_no = 1000;
```

update customer set avail_balance = avail_balance + 3000 where acc_no = 1001;

output:

```
select *from mini_statement;
```

```
+-----+-----+
| acc_no | avail_balance |
+-----+-----+
| 1000 |      7000 |
| 1001 |     12000 |
+-----+-----+
```

After Update Trigger:

As the name implies, this trigger is invoked after an updating occurs. (i.e., it gets implemented after an update statement is executed.).

Example:

We create another table:

```
create table micro_statement (acc_no integer, avail_balance decimal,
                             foreign key(acc_no) references customer(acc_no) on delete cascade);
```

Insert another value into customer:

```
insert into customer values (1002, "Janitor", 4500);
```

Trigger to insert (new) values of account number and available balance into micro_statement record after an update has occurred:

delimiter //

```
create trigger update_after
```

```
-> after update on customer
```

```
-> for each row
```

```
-> begin
```

```
-> insert into micro_statement values(new.acc_no, new.avail_balance);
```

```
-> end; //
```

Making an update to invoke trigger:

```
update customer set avail_balance = avail_balance + 1500 where acc_no = 1002;
```

Output:

```
select *from micro_statement;
```

```
+-----+-----+
| acc_no | avail_balance |
+-----+-----+
```

```
| 1002 |      6000 |  
+-----+-----+
```

Before Insert Trigger:

As the name implies, this trigger is invoked before an insert, or before an insert statement is executed.

Example:

Considering tables:

```
create table contacts (contact_id INT (11) NOT NULL AUTO_INCREMENT,  
                        last_name VARCHAR (30) NOT NULL, first_name VARCHAR (25),  
                        -> birthday DATE, created_date DATE, created_by VARCHAR(30),  
                        CONSTRAINT contacts_pk PRIMARY KEY (contact_id));
```

Trigger to insert contact information such as name, birthday and creation-date/user into a table contact before an insert occurs:

```
delimiter //  
create trigger contacts_before_insert  
    -> before insert  
    -> on contacts for each row  
    -> begin  
    -> DECLARE vUser varchar(50);  
    ->  
    -> -- Find username of person performing INSERT into table  
    -> select USER() into vUser;  
    ->  
    -> -- Update create_date field to current system date  
    -> SET NEW.created_date = SYSDATE();  
    ->  
    -> -- Update created_by field to the username of the person performing the  
INSERT  
    -> SET NEW.created_by = vUser;  
    -> end; //
```

Making an insert to invoke the trigger:

```
insert into contacts values (1, "Newton", "Enigma", "1999-08-19", "2018-03-17", "xyz");
```

Output:

```
select *from contacts;
```

```
+-----+-----+-----+-----+-----+-----+
| contact_id | last_name | first_name | birthday | created_date | created_by |
+-----+-----+-----+-----+-----+-----+
|      1 | Newton   | Enigma    | 1999-08-19 | 2019-05-11   | root@localhost |
+-----+-----+-----+-----+-----+-----+
```

After Insert Trigger:

As the name implies, this trigger gets invoked after an insert is implemented.

Example:

Consider tables:

```
create table contacts1(contact_id int (11) NOT NULL AUTO_INCREMENT,
    last_name VARCHAR(30) NOT NULL, first_name VARCHAR(25), birthday DATE,
    CONSTRAINT contacts_pk PRIMARY KEY (contact_id));
```

```
create table contacts1_audit (contact_id integer, created_date date,
created_by varchar (30));
```

Trigger to insert contact_id and contact creation-date/user information into contacts_audit record after an insert occurs:

```
delimiter //
```

```
create trigger contacts_after_insert
```

```
after insert
```

```
on contacts1 for each row
```

```
begin
```

```
    DECLARE vUser varchar(50);
```

```
    -- Find username of person performing the INSERT into table
```

```
    SELECT USER() into vUser;
```

```
    -- Insert record into audit table
```

```
    INSERT into contacts1_audit
```

```
    ( contact_id,
```

```
      created_date,
```

```
      created_by)
```

```
    VALUES
```

```
    ( NEW.contact_id,
```

```
      SYSDATE(),
```

```
      vUser );
```

```
END; //
```

Making an insert to invoke the trigger:

```
insert into contacts1 values (1, "Kumar", "Rupesh", "1999-06-20");
```

Output:

```
select *from contacts_audit;
```

```
+-----+-----+-----+
| contact_id | created_date | created_by |
+-----+-----+-----+
|      1 | 2019-05-11 | root@localhost |
+-----+-----+-----+
```

Before Delete Trigger:

As the name implies, this trigger is invoked before a delete occurs, or before deletion statement is implemented.

Example:

Consider tables:

```
create table contacts (contact_id int (11) NOT NULL AUTO_INCREMENT,
last_name VARCHAR (30) NOT NULL, first_name VARCHAR (25),
birthday DATE, created_date DATE, created_by VARCHAR(30),
CONSTRAINT contacts_pk PRIMARY KEY (contact_id));
```

```
create table contacts_audit (contact_id integer, deleted_date date, deleted_by
varchar(20));
```

Trigger to insert contact_id and contact deletion-date/user information into contacts_audit record before a delete occurs:

```
delimiter //
```

```
create trigger contacts_before_delete
before delete
on contacts for each row
begin
```

```
    DECLARE vUser varchar(50);
```

```
    -- Find username of person performing the DELETE into table
    SELECT USER() into vUser;
```

```
    -- Insert record into audit table
    INSERT into contacts_audit
    ( contact_id,
```

```

        deleted_date,
        deleted_by)
VALUES
( OLD.contact_id,
  SYSDATE(),
  vUser );
end; //

```

Making an insert and then deleting the same to invoke the trigger:
 insert into contacts values (1, "Bond", "Ruskin", str_to_date ("19-08-1995",
 "%d-%m-%Y"), str_to_date ("27-04-2018", "%d-%m-%Y"), "xyz");

delete from contacts where last_name="Bond";

Output:

```

select *from contacts_audit;
+-----+-----+-----+
| contact_id | deleted_date | deleted_by |
+-----+-----+-----+
|          1 | 2019-05-11   | root@localhost |
+-----+-----+-----+
1 row in set (0.0007 sec)

```

After Delete Trigger:

As the name implies, this trigger is invoked after a delete occurs, or after a delete operation is implemented.

Example:

Consider the tables:

```

create table contacts (contact_id int (11) NOT NULL AUTO_INCREMENT,
last_name VARCHAR (30) NOT NULL, first_name VARCHAR (25),
birthday DATE, created_date DATE, created_by VARCHAR (30),
CONSTRAINT contacts_pk PRIMARY KEY (contact_id));

```

```

create table contacts_audit (contact_id integer, deleted_date date, deleted_by
varchar(20));

```

Trigger to insert contact_id and contact deletion-date/user information into contacts_audit record after a delete occurs:

```

delimiter //
create trigger contacts_after_delete
after delete
on contacts for each row
begin

    DECLARE vUser varchar(50);

    -- Find username of person performing the DELETE into table
    SELECT USER() into vUser;

    -- Insert record into audit table
    INSERT into contacts_audit
    ( contact_id,
      deleted_date,
      deleted_by)
    VALUES
    ( OLD.contact_id,
      SYSDATE(),
      vUser );
end;

```

Making an insert and deleting the same to invoke the trigger:

```

insert into contacts values (1, "Newton", "Isaac", str_to_date ("19-08-1985",
"%d-%m-%Y"), str_to_date ("23-07-2018", "%d-%m-%Y"), "xyz");

```

```

delete from contacts where first_name="Isaac";

```

Output:

```

select *from contacts_audit;
+-----+-----+-----+
| contact_id | deleted_date | deleted_by |
+-----+-----+-----+
|      1 | 2019-05-11 | root@localhost |
+-----+-----+-----+
1 row in set (0.0009 sec)

```

Drop trigger:

```

Drop trigger contacts_after_insert

```

View

CREATE VIEW Statement:

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

```
CREATE VIEW Bangladeshi_Customers AS
SELECT CustomerName, ContactName
FROM CustomerInfo
WHERE Country = 'Bangladesh';
```

```
select *from Bangladeshi_Customers
```

```
CREATE VIEW Above_Average_Price AS
SELECT ProductName, Price
FROM ProductDetails
WHERE Price > (SELECT AVG(Price) FROM ProductDetails);
```

```
select *from Above_Average_Price
```

Updating a View:

A view can be updated with the CREATE OR REPLACE VIEW statement.

The following SQL adds the "City" column to the "Bangladeshi_Customers" view:


```
CREATE OR REPLACE VIEW Bangladeshi_Customers AS  
SELECT CustomerName, ContactName, City  
FROM CustomerInfo  
WHERE Country = 'Bangladesh';
```

```
select *from Bangladeshi_Customers
```

Dropping a View:

```
Drop view Bangladeshi_Customers
```

Stored Procedures

Procedures in MySQL:

A procedure is a subroutine (like a subprogram) in a regular scripting language, stored in a database. In the case of MySQL, procedures are written in MySQL and stored in the MySQL database/server. A MySQL procedure has a name, a parameter list, and SQL statement(s).

There are four different types of MySQL procedures:

1. Procedure with no parameters:

A procedure without parameters does not take any input or casts an output indirectly. It is simply called with its procedure name followed by () (without any parameters). It is used for simple queries.

Example:

Consider two tables author and book:

```
create table author (author_id integer primary key, authorName varchar(30),  
email varchar (25), gender varchar (6));
```

```
create table book (BookId integer not null unique, ISBN integer primary key,  
book_name varchar (30) not null, author integer, ed_num integer, price integer, pages  
integer, foreign key (author) references author (author_id) on delete cascade);
```

Inserting values into them:

insert into author values

(1, "Kraig Muller", "Wordnewton@gmail.com", "Male");

insert into author values

(2, "Karrie Nicolette", "karrie23@gmail.com", "Female");

insert into book values

(1, 001, "Glimpses of the past", 1, 1, 650, 396);

insert into book values

(2, 002, "Beyond The Horizons of Venus", 1, 1, 650, 396);

insert into book values

(3, 003, "Ultrasonic Aquaculture", 2, 1, 799, 500);

insert into book values

(4, 004, "Cyrogenic Engines", 2, 1, 499, 330);

Procedure (with no parameters) to display all the books:

```
delimiter //
create procedure display_book()
begin
select *from book;
end //
call display_book()
```

Output:

BookId	ISBN	book_name	author	ed_num	price	pages
1	1	Glimpses of the past	1	1	650	396
2	2	Beyond The Horizons of	1	1	650	396
3	3	Ultrasonic Aquaculture	2	1	799	500
4	4	Cyrogenic Engines	2	1	499	330

2. Procedure with IN parameter:

An IN parameter is used to take a parameter as input.

Example:

Procedure to update price of a book taking ISBN of the book and its new price as input:
(considering the tables above)

```
delimiter //
create procedure update_price (IN temp_ISBN varchar(10), IN new_price integer)
begin
update book set price=new_price where ISBN=temp_ISBN;
end //
→ call update_price(001, 600)
```

We changed the price of book with ISBN '001'(Glimpses of the past) to 600 (from its default price 650).

Output:

```
select *from book;
```

```
+-----+-----+-----+-----+-----+-----+
```

BookId	ISBN	book_name	author	ed_num	price	pages
1	1	Glimpses of the past	1	1	600	396
2	2	Beyond The Horizons of	1	1	650	396
3	3	Ultrasonic Aquaculture	2	1	799	500
4	4	Cyrogenic Engines	2	1	499	330

3. Procedure with OUT parameter:

An OUT parameter is used to pass a parameter as output or display like the select operator.

Example:

Procedure to display the highest price among all the books with an output parameter:

```
delimiter //
create procedure disp_max(OUT highestprice integer)
begin
    select max(price) into highestprice from book;
end //
call disp_max(@Maximumprice) //
select @Maximumprice
```

Output:

The highest price from our book database is of the book with ISBN 003 (Ultrasonic Aquaculture) with a price of 799, which is displayed.

```
+-----+
| @M |
+-----+
| 799 |
+-----+
```

4. Procedure with IN-OUT parameter:

An INOUT parameter is a combination of IN and OUT parameters.

Example:

Procedure to take gender type input ('Male'/'Female' here) with an in-out parameter which reflects the number of authors falling in that gender category/type:

```
delimiter //
create procedure disp_gender(INOUT total_gender integer, IN take_gender varchar(6))
```

```

begin
select COUNT(gender) INTO total_gender FROM author where gender =
take_gender;
end; //
delimiter ;
call disp_gender(@TM, "Male");
select @TM;
call disp_gender(@TF, "Female");
select @TF;

```

Output:

We have two authors, one being male and one being female as per insertions in the table author. Hence, output is 1 for one male author and 1 for one female author respectively.

```

+----+
| @M |
+----+
| 1 |
+----+

```

```

+----+
| @F |
+----+
| 1 |
+----+

```

Drop procedure:

Drop procedure disp_gender