UPDATE Triggers

This lesson shows how to create triggers associated with the UPDATE statement.

UPDATE Triggers

Update triggers for a table are automatically executed when an **UPDATE** is made to the table. These triggers can run before or after the table is updated. Both the **NEW** and **OLD** keywords can be used as both values of a column are accessible when using update triggers. When the trigger runs **BEFORE** the update is performed, the **NEW** value can be updated while the **OLD** value cannot be updated. When the trigger runs **AFTER** the update has been performed, we can only access the **NEW** and **OLD** values but cannot update them.

Update triggers cannot be made for views. **BEFORE UPDATE** triggers can be used to validate data and make necessary corrections or notify user before an update is made to the table. They can also be used to store the new and old values of a column to maintain an update log. **AFTER UPDATE** triggers can also be used to maintain a change log or to update summary table in the event of an update.

Syntax

```
CREATE TRIGGER trigger_name [BEFORE | AFTER] UPDATE
ON table_name
FOR EACH ROW
trigger_body
```

> Connect to the terminal below by clicking in the widget. Once connected, the command line prompt will show up. Enter or copy-paste the command **./DataJek/Lessons/48lesson.sh** and wait for the mysql prompt to start-up.

```
-- The lesson queries are reproduced below for convenient copy/paste into the terminal.

-- Query 1
CREATE TABLE DigitalActivity (
RowID INT AUTO_INCREMENT PRIMARY KEY,
ActorID INT NOT NULL,
Detail VARCHAR(100) NOT NULL,
UpdatedOn TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- Query 2
DELIMITER **

CREATE TRIGGER BeforeDigitalAssetUpdate
BEFORE UPDATE
ON DigitalAssets
FOR EACH ROW
BEGIN
 DECLARE errorMessage VARCHAR(255);

 IF NEW.LastUpdatedOn < OLD.LastUpdatedOn THEN
   SET errorMessage = CONCAT('The new value of LastUpatedOn column: ',
     NEW.LastUpdatedOn,' cannot be less than the current value: ',
     OLD.LastUpdatedOn);

   SIGNAL SQLSTATE '45000'
   SET MESSAGE_TEXT = errorMessage;
 END IF;

 IF NEW.LastUpdatedOn != OLD.LastUpdatedOn THEN
   INSERT into DigitalActivity (ActorId, Detail)
   VALUES (New.ActorId, CONCAT('LastUpdate value for ',NEW.AssetType,
          ' is modified from ',OLD.LastUpdatedOn, ' to ',
          NEW.LastUpdatedOn));
 END IF;
END **
DELIMITER ;

-- Query 3
UPDATE DigitalAssets
SET LastUpdatedOn = '2020-02-15 22:10:45'
WHERE ActorID = 2 AND Assettype = 'Website';

UPDATE DigitalAssets
SET LastUpdatedOn = '2018-01-15 22:10:45'
WHERE ActorID = 5 AND AssetType = 'Pinterest';

SELECT * FROM DigitalActivity;
```

```
-- Query 4
DELIMITER **

CREATE TRIGGER AfterActorUpdate
AFTER UPDATE ON Actors
FOR EACH ROW
BEGIN
   DECLARE TotalWorth, RowsCount INT;
   INSERT INTO ActorsLog
   SET ActorId = NEW.Id, FirstName = New.FirstName, LastName =  NEW.SecondName, DateTime = NO

  IF NEW.NetWorthInMillions != OLD.NetWorthInMillions THEN
    SELECT SUM(NetWorthInMillions) INTO TotalWorth
    FROM Actors;
    SELECT COUNT(*) INTO RowsCount
    FROM Actors;

    UPDATE NetWorthStats
    SET AverageNetWorth = ((Totalworth) / (RowsCount));
END IF;
END **
DELIMITER ;

-- Query 5
SELECT * FROM NetWorthStats;

UPDATE Actors
SET NetWorthInMillions = '100'
WHERE Id = 5;

SELECT * FROM NetWorthStats;
SELECT * FROM ActorsLog;

-- Query 6
UPDATE Actors
SET MaritalStatus = 'Single'
WHERE Id = 7;

SELECT * FROM NetWorthStats;
SELECT * FROM ActorsLog;
```

Terminal

1. Let's create an example of **BEFORE UPDATE** triggers. We will create a trigger on the **DigitalAssets** table that keeps track of changes to the **LastUpdatedOn** column. Whenever a row is updated, the trigger will check the query and if the change is made to the **LastUpdatedOn** column, then a new row is inserted in the **DigitalActivity** table. Before we define the trigger, let's create the **DigitalActivity** table as follows:

```
CREATE TABLE DigitalActivity (
RowID INT AUTO_INCREMENT PRIMARY KEY,

ActorID INT NOT NULL,
Detail VARCHAR(100) NOT NULL,
UpdatedOn TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

We intend to store the **ActorID** and information about the previous and new values of **LastUpdatedOn** column as well as the time when the change was made.

2. Now, we will create the trigger **BeforeDigitalAssetUpdate** as follows:

```
DELIMITER **

CREATE TRIGGER BeforeDigitalAssetUpdate
BEFORE UPDATE
ON DigitalAssets
FOR EACH ROW
BEGIN
 DECLARE errorMessage VARCHAR(255);

 IF NEW.LastUpdatedOn < OLD.LastUpdatedOn THEN
   SET errorMessage = CONCAT('The new value of LastUpatedOn colum
n: ',
     NEW.LastUpdatedOn,' cannot be less than the current value: '
,
     OLD.LastUpdatedOn);

   SIGNAL SQLSTATE '45000'
   SET MESSAGE_TEXT = errorMessage;
 END IF;

 IF NEW.LastUpdatedOn != OLD.LastUpdatedOn THEN
   INSERT into DigitalActivity (ActorId, Detail)
   VALUES (New.ActorId, CONCAT('LastUpdate value for ',NEW.AssetT
ype,
          ' is modified from ',OLD.LastUpdatedOn, ' to ',
          NEW.LastUpdatedOn));
 END IF;

END **
```

```
DELIMITER ;
```

We are performing two actions in this trigger. First the value of the timestamp provided in the **UPDATE** query is compared with the value that already exists for that record. If the timestamp provided is less than the old one, then an error message is displayed. This is based on the assumption that as the actors update their digital assets the timestamp of the new update will always be greater than the previous update. There is a flaw in this logic; assume that timestamp entered was incorrect and is now being updated to a smaller value. We will ignore that case here and assume that all values in the **LastUpdatedOn** column are correct. The purpose of this trigger is to issue an error message to the user and prevent an incorrect update.

The trigger also performs an **INSERT** to the **DigitalActivity** table. We first check if the NEW timestamp value is different from the OLD one. This is done to ensure that the **UPDATE** is made to the **LastUpdatedOn** column and not any other column in the **DigitalAssets** table.

3. To test the **BeforeDigitalActivityUpdate** trigger, we will perform two **UPDATE** operations as follows:

```
UPDATE DigitalAssets
SET LastUpdatedOn = '2020-02-15 22:10:45'
WHERE ActorID = 2 AND Assettype = 'Website';

UPDATE DigitalAssets
SET LastUpdatedOn = '2018-01-15 22:10:45'
WHERE ActorID = 5 AND AssetType = 'Pinterest';

SELECT * FROM DigitalActivity;
```

The first update query was successful and resulted in an entry in the **DigitalActivity** table. The second update query resulted in an error message and the record was not updated.

4. We will discuss **AFTER UPDATE** triggers now. In the last lesson, we created **ActorsLog** table and a summary table **NetWorthStats** to store the **AverageNetWorth** of all actors in the **Actors** table. Whenever a row is updated, we will log this activity in the **ActorsLog** table. If the **NetWorthInMillions** column was changed, then the summary table will be updated. The trigger **AfterActorsUpdate** is defined as follows:

```
DELIMITER **

CREATE TRIGGER AfterActorUpdate
AFTER UPDATE ON Actors
FOR EACH ROW
BEGIN
   DECLARE TotalWorth, RowsCount INT;

   INSERT INTO ActorsLog
   SET ActorId = NEW.Id, FirstName = New.FirstName, LastName =  N
EW.SecondName, DateTime = NOW(), Event = 'UPDATE';

  IF NEW.NetWorthInMillions != OLD.NetWorthInMillions THEN

   SELECT SUM(NetWorthInMillions) INTO TotalWorth
    FROM Actors;

    SELECT COUNT(*) INTO RowsCount
    FROM Actors;

    UPDATE NetWorthStats
    SET AverageNetWorth = ((Totalworth) / (RowsCount));
END IF;
END **

DELIMITER ;
```

This trigger will perform an **INSERT** operation and update **NetWorthStats** only when the column **NetWorthInMillions** is changed. We can access both the previous and new values of the column using **OLD** and **NEW** keywords but we cannot modify them.

5. To test this trigger, we will update a row of **Actors** table.

```
SELECT * FROM NetWorthStats;

UPDATE Actors
SET NetWorthInMillions = '100'
WHERE Id = 5;

SELECT * FROM NetWorthStats;
SELECT * FROM ActorsLog;
```

We checked the value of **AverageNetWorth** before executing the **UPDATE** query. **AfterUpdateTrigger** was fired after the update operation was successful and changed the **AverageNetWorth** as well as inserting a row in the **ActorsLog** table:

Now let's update one more record:

```
UPDATE Actors
SET MaritalStatus = 'Single'
WHERE Id = 7;

SELECT * FROM NetWorthStats;
SELECT * FROM ActorsLog;
```

This time there is no change in the **NetWorthStats** table but a row is inserted in the **ActorsLog** table.