Triggers

Introduction on Triggers

- A trigger is a set of actions that are run automatically when a specified change operation (SQL INSERT, UPDATE, or DELETE statement) is performed on a specified table.
- Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

- Triggers help the database designer ensure certain actions, such as maintaining an audit file, are completed regardless of which program or user makes changes to the data.
- The programs are called triggers since an event, such as adding a record to a table, fires their execution.

- **Events**
- The triggers can occur AFTER or INSTEAD OF a DML action. Triggers are associated with the database DML actions INSERT, UPDATE, and DELETE. Triggers are defined to run when these actions are executed on a specific table.

Uses for triggers

- Enforce business rules
- Validate input data
- Generate a unique value for a newly-inserted row in a different file.
- Write to other files for audit trail purposes
- Query from other files for cross-referencing purposes
- Access system functions
- Replicate data to different files to achieve data consistency

- **Complex Auditing**
- You can use triggers to track changes made to tables.
- Typically when creating audit trails, you'll use AFTER triggers.
- You may think this is redundant, as many changes are logged in the databases journals, but the logs are meant for database recovery and aren't easily accessible by user programs. The TransactionHistory table is easily referenced and can be incorporated into end user reports.

- **Enforce Business Rules**
- Triggers can be used to inspect all data before a DML action is performed. You can use INSTEAD OF triggers to “intercept” the pending DML operation, apply any business rules, and ultimately complete the transaction.
- An example business rule may be that a customer status is defined as:
- Gold – Purchases over $1,000,000 in the past 12 months.
- Silver – Purchase of $500,000 to $1,000,000 in the past 12 months.
- Bronze – All other purchase levels.
- An INSTEAD OF trigger could be defined to check the customer status each time a customer record is added or modified. The status check would involve creating a sum of all the customers’ purchases and ensuring the new status corresponds with the sum of the last 12 months of purchases.

- **Derive Column Values**
- Triggers can be used to calculate column values. For instance, for each customer you may wish to maintain a TotalSales column on the customer record. Of course, for this to remain accurate, it would have to be update every time a sales was made.
- This could be done using an AFTER trigger on INSERT, UPDATE, and DELETE statements for the Sales table.

Benefits of using triggers in business

- **Faster application development.** Because the database stores triggers, you do not have to code the trigger actions into each database application.
- **Global enforcement of business rules**. Define a trigger once and then reuse it for any application that uses the database.
- **Easier maintenance**. If a business policy changes, you need to change only the corresponding trigger program instead of each application program.
- **Improve performance in client/server environment**. All rules run on the server before the result returns.
- Implementation of SQL triggers is based on the SQL standard. It supports constructs that are common to most programming languages. It supports the declaration of local variables, statements to control the flow of the procedure, assignment of expression results to variables, and error handling.

Create MySQL triggers

- A trigger is a named database object that is associated with a table, and it activates when a particular event (e.g. an insert, update or delete) occurs for the table.
- The statement CREATE TRIGGER creates a new trigger in MySQL.
- Syntax:

```
CREATE
[DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
trigger_body
```

- trigger_time: { BEFORE | AFTER }
- trigger_event: { INSERT | UPDATE | DELETE }

- **DEFINER clause:** The DEFINER clause specifies the MySQL account to be used when checking access privileges at trigger activation time. If a user value is given, it should be a MySQL account specified as 'user_name'@'host_name' (the same format used in the GRANT statement), CURRENT_USER, or CURRENT_USER().
  The default DEFINER value is the user who executes the CREATE TRIGGER statement. This is the same as specifying DEFINER = CURRENT_USER explicitly.
  If you specify the DEFINER clause, these rules determine the valid DEFINER user values:
- If you do not have the SUPER privilege, the only permitted user value is your own account, either specified literally or by using CURRENT_USER. You cannot set the definer to some other account.
- If you have the SUPER privilege, you can specify any syntactically valid account name. If the account does not actually exist, a warning is generated.
- Although it is possible to create a trigger with a nonexistent DEFINER account, it is not a good idea for such triggers to be activated until the account actually does exist. Otherwise, the behavior with respect to privilege checking is undefined.

- **trigger_name:** All triggers must have unique names within a schema. Triggers in different schemas can have the same name.
- **trigger_time:** trigger_time is the trigger action time. It can be BEFORE or AFTER to indicate that the trigger activates before or after each row to be modified.
- **trigger_event:** trigger_event indicates the kind of operation that activates the trigger. These trigger_event values are permitted:
- The trigger activates whenever a new row is inserted into the table; for example, through INSERT, LOAD DATA, and REPLACE statements.
- The trigger activates whenever a row is modified; for example, through UPDATE statements.
- The trigger activates whenever a row is deleted from the table; for example, through DELETE and REPLACE statements. DROP TABLE and TRUNCATE TABLE statements on the table do not activate this trigger, because they do not use DELETE. Dropping a partition does not activate DELETE triggers, either.

- **tbl_name :** The trigger becomes associated with the table named tbl_name, which must refer to a permanent table. You cannot associate a trigger with a TEMPORARY table or a view.
- **trigger_body:** trigger_body is the statement to execute when the trigger activates. To execute multiple statements, use the BEGIN ... END compound statement construct. This also enables you to use the same statements that are permissible within stored routines.

- Here is a simple example:

```
CREATE TRIGGER ins_sum
BEFORE INSERT ON account
FOR EACH ROW SET @sum = @sum + NEW.amount;
```

- In the above example, there is new keyword '**NEW**' which is a MySQL extension to triggers. There is two MySQL extension to triggers '**OLD**' and '**NEW**'. OLD and NEW are not case sensitive.
- Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger

- In an INSERT trigger, only NEW.col_name can be used.
- In a UPDATE trigger, you can use OLD.col_name to refer to the columns of a row before it is updated and NEW.col_name to refer to the columns of the row after it is updated.
- In a DELETE trigger, only OLD.col_name can be used; there is no new row.
- A column named with OLD is read only. You can refer to it (if you have the SELECT privilege), but not modify it. You can refer to a column named with NEW if you have the SELECT privilege for it. In a BEFORE trigger, you can also change its value with SET NEW.col_name = value if you have the UPDATE privilege for it. This means you can use a trigger to modify the values to be inserted into a new row or used to update a row. (Such a SET statement has no effect in an AFTER trigger because the row change will have already occurred.)

SHOW TRIGGERS

- SHOW TRIGGERS statement is used to list the triggers currently defined for tables in a database

- Syntax:

SHOW TRIGGERS [{FROM | IN} db_name] [LIKE 'pattern' | WHERE expr]

- **SHOW TRIGGERS FROM CSD2204W18;**

AFTER INSERT

- we have two tables: emp_details and log_emp_details.
- To insert some information into log_emp_details table (which have three fields employee id and salary and edttime) every time, when an INSERT happen into emp_details table we have used the following trigger :

Example

```
CREATE
        TRIGGER `csd2204w18`.`emp_details_AINS`
        AFTER INSERT ON `csd2204w18`.`emp_details`
        FOR EACH ROW
BEGIN
        INSERT INTO log_emp_details VALUES(NEW.employee_id,
        NEW.salary, NOW());
END$$
```

- SELECT * FROM log_emp_details;

- INSERT INTO emp_details VALUES(236, 'RABI', 'CHANDRA', 'RABI','123.34.45700', '2013-01-12', 'AD_VP', 15000, 0.5);

- SELECT * FROM log_emp_details;

BEFORE INSERT

- In the following example, before insert a new record in emp_details table, a trigger check the column value of FIRST_NAME, LAST_NAME, JOB_ID and
- If there are any space(s) before or after the FIRST_NAME, LAST_NAME, TRIM() function will remove those.
- The value of the JOB_ID will be converted to upper cases by UPPER() function.

Example

```
CREATE or REPLACE TRIGGER `emp_details_BINS`
        BEFORE INSERT   ON emp_details
        FOR EACH ROW
BEGIN
        SET NEW.FIRST_NAME = TRIM(NEW.FIRST_NAME);
        SET NEW.LAST_NAME = TRIM(NEW.LAST_NAME);
        SET NEW.JOB_ID = UPPER(NEW.JOB_ID);
END$$
```

- INSERT INTO emp_details VALUES (334, ' Ana ', ' King', 'ANA', '212.212.21201', '2013-02-05', 'it_prog', 17000, .50);
- SELECT * FROM EMP_DETAILS;

AFTER UPDATE

- In the EMP_DETAILS table, we want to increase salary of every employee by 1000 and log the update information into LOG_EMP_DETAILS_UPDATE table with the user & old and new salary.

```
CREATE or REPLACE TRIGGER `emp_details_AUPD`
        AFTER UPDATE ON emp_details FOR EACH ROW
BEGIN
        INSERT INTO LOG_EMP_DETAILS_UPDATE VALUES (user(),
        CONCAT('Update employee Record ',
        OLD.FIRST_NAME,' Old Salary :',OLD.Salary,' New Salary : ',
        NEW.Salary),NOW());
END$$
```

- UPDATE EMP_DETAILS SET SALARY = SALARY + 1000;

- SELECT * FROM EMP_DETAILS;

- SELECT * FROM LOG_EMP_DETAILS_UPDATE;

BEFORE UPDATE

- We have a table student_marks with 10 columns and 4 rows. There are data only in STUDENT_ID and NAME columns.
- Now the exam is over and we have received all subject marks, now we will update the table, total marks of all subject, the percentage of total marks and grade will be automatically calculated. For this sample calculation, the following conditions are assumed :
- Total Marks (will be stored in TOTAL column) : TOTAL = SUB1 + SUB2 + SUB3 + SUB4 + SUB5

  Percentage of Marks (will be stored in PER_MARKS column) :
  PER_MARKS = (TOTAL)/5

  Grade (will be stored GRADE column) :
  - If PER_MARKS>=90 -> 'EXCELLENT'
  - If PER_MARKS>=75 AND PER_MARKS<90 -> 'VERY GOOD'
  - If PER_MARKS>=60 AND PER_MARKS<75 -> 'GOOD'
  - If PER_MARKS>=40 AND PER_MARKS<60 -> 'AVERAGE'
  - If PER_MARKS<40-> 'NOT PROMOTED'

Example

```
CREATE OR REPLACE TRIGGER `stud_marks_BUPD`
BEFORE UPDATE ON stud_marks FOR EACH ROW
BEGIN
        SET NEW.TOTAL = NEW.SUB1 + NEW.SUB2 + NEW.SUB3 +
        NEW.SUB4 + NEW.SUB5;
        SET NEW.PERCENTAGE = NEW.TOTAL/5;
        IF NEW.PERCENTAGE >=90
        THEN            SET NEW.GRADE = 'EXCELLENT';
        ELSEIF NEW.PERCENTAGE>=75 AND NEW.PERCENTAGE<90
        THEN            SET NEW.GRADE = 'VERY GOOD';
        ELSEIF NEW.PERCENTAGE>=60 AND NEW.PERCENTAGE<75
        THEN            SET NEW.GRADE = 'GOOD';
        ELSEIF NEW.PERCENTAGE>=40 AND NEW.PERCENTAGE<60
        THEN            SET NEW.GRADE = 'AVERAGE';
        ELSE            SET NEW.GRADE = 'NOT PROMOTED';
END IF;
END$$
```

- UPDATE STUD_MARKS SET SUB1 = 54, SUB2 = 69, SUB3 = 89, SUB4 = 87, SUB5 = 59 WHERE STUDENT_ID = 1;

- SELECT * FROM STUD_MARKS;

AFTER DELETE

- We want to log the information into LOG_EMP_DETAILA_UPDATE table when the employee record gets deleted from EMP_DETAILS table.

```
CREATE or REPLACE TRIGGER `emp_details_ADEL`
AFTER DELETE ON emp_details FOR EACH ROW
BEGIN
        INSERT INTO LOG_EMP_DETAILS_UPDATE VALUES (user(), CONCAT('Delete Employee Record', OLD.FIRST_NAME, '-> Deleted on '),NOW());
END$$
```

- SELECT * FROM EMP_DETAILS;
- DELETE FROM EMP_DETAILS WHERE EMPLOYEE_ID = 100;
- SELECT * FROM EMP_DETAILS;
- SELECT * FROM LOG_EMP_DETAILS_UPDATE;

Error handling during trigger execution

- If a BEFORE trigger fails, the operation on the corresponding row is not performed.
- A BEFORE trigger is activated by the attempt to insert or modify the row, regardless of whether the attempt subsequently succeeds.
- An AFTER trigger is executed only if any BEFORE triggers and the row operation execute successfully.
- An error during either a BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation.
- For transactional tables, failure of a statement should cause a rollback of all changes performed by the statement.

Delete trigger

- To delete or destroy a trigger, use a DROP TRIGGER statement.
- *DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name*
- **DROP TRIGGER IF EXISTS csd2204w18.emp_details_ADEL;**
- If you drop a table, any triggers for the table are also dropped.

Multiple triggers

- MySQL allows you to create multiple triggers for the same event and action time in a table.
- The triggers will activate sequentially when the event occurs.
- The syntax for creating the first trigger remains the same.
- In case you have multiple triggers for the same event in a table, MySQL will invoke the triggers in the order that they were created.
- To change the order of triggers, you need to specify FOLLOWS or PRECEDES after the FOR EACH ROW clause.
- The FOLLOWS option allows the new trigger to activate after the existing trigger.
- The PRECEDES option allows the new trigger to activate before the existing trigger.

- The following is the syntax of creating a new additional trigger with explicit order:

```
DELIMITER $$
CREATE TRIGGER  trigger_name
[BEFORE|AFTER] [INSERT|UPDATE|DELETE] ON table_name
FOR EACH ROW [FOLLOWS|PRECEDES] existing_trigger_name
BEGIN
…
END$$
DELIMITER ;
```

-

- We will use the products table.
- Suppose, whenever we change the price of a product (column MSRP), we want to log the old price in a separate table named price_logs .

```
CREATE or REPLACE TRIGGER before_products_update
BEFORE UPDATE on Products FOR EACH ROW
BEGIN
        INSERT INTO price_logs (product_code,price) VALUES(old.c
        ode,old.price);
END$$
```

- UPDATE products SET price = 100 WHERE Code = '1';

- SELECT * FROM price_logs;

Example 2

- Suppose we want to see not only the old price and when it was changed but also who changed it.

- We will use user_change_logs table to store the data of users who made the changes.

- Now, we create a second trigger that activates on the BEFORE UPDATE event of the products table.
- This trigger will update the user_change_logs table with the data of the user who made the changes.
- It is activated after the before_products_update trigger.

```
CREATE TRIGGER before_products_update_2
   BEFORE UPDATE ON products
   FOR EACH ROW FOLLOWS before_products_update
BEGIN
   INSERT INTO user_change_logs(product_code,updated_by)
   VALUES(old.productCode,user());
END$$
```

- UPDATE products SET price = 100 WHERE Code = '1';
- SELECT * FROM price_logs;
- SELECT * FROM user_change_logs;

Class Activity

- Consider the table
  - CUSTOM(id, Name, Email, PurchaseAmount, MembershipType)

- Write a trigger to change the MembershipType of a customer if the PurchaseAmount is updated. Use the following criteria to find the MembershipType.

  - Gold – Purchases over $1,000,000.
  - Silver – Purchase of $500,000 to $1,000,000.
  - Bronze – All other purchase levels.

References

- https://www.w3resource.com/mysql/mysql-triggers.php
- https://www.essentialsql.com/what-is-a-database-trigger/
- http://www.mysqltutorial.org/mysql-triggers/create-multiple-triggers-for-the-same-trigger-event-and-action-time/