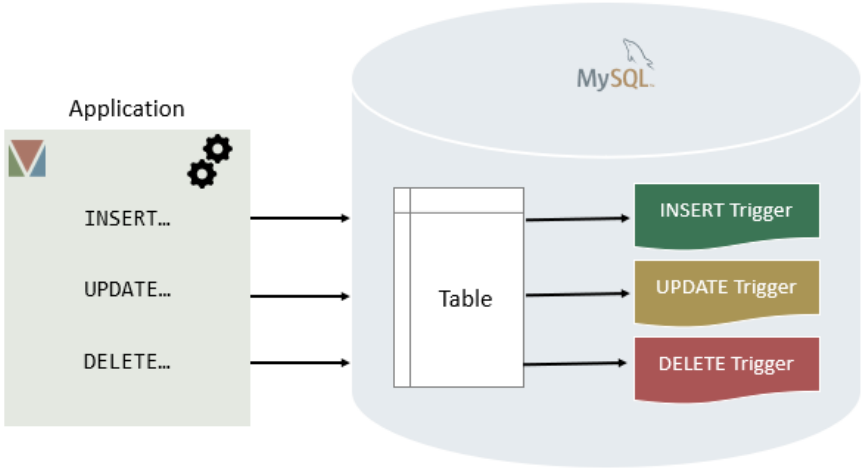
Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 8

****

All examples in this lab sheet are not based on the database used in previous labs to preserve modification in the schema of existing sample databases. You can create a new temporary database and test the examples given in this lab sheet.

1 Triggers

A trigger in MySQL is a set of SQL statements that reside in a system catalog. It is a special type of stored procedure that is invoked automatically in response to an event. Each trigger is associated with a table, which is activated on any DML statement such as INSERT, UPDATE, or DELETE.



1.1 Features of Triggers

- ☞ Triggers help us to enforce business rules.
- ☞ Triggers help us to validate data even before they are inserted or updated.
- ☞ Triggers help us to keep a log of records like maintaining audit trails in tables.
- ☞ SQL triggers provide an alternative way to check the integrity of data.
- ☞ Triggers provide an alternative way to run the scheduled task.
- ☞ Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed.
- ☞ Triggers reduce the client-side code that saves time and effort.
- ☞ Triggers help us to scale our application across different platforms.
- ☞ Triggers are easy to maintain.

1.2 Benefits of Triggers

- ☞ Audit data modifications
- ☞ Log events transparently
- ☞ Enforce complex business rules
- ☞ Derive column values automatically
- ☞ Implement complex security authorizations
- ☞ Maintain replicated tables

1.3 Limitations of Using Triggers in MySQL

- ☞ MySQL triggers do not allow to use of all validations; they only provide extended validations. For example, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- ☞ Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- ☞ Triggers may increase the overhead of the database server.

1.4 Types of Triggers in MySQL

We can define the maximum six types of actions or events in the form of triggers:

- ☞ Before Insert: It is activated before the insertion of data into the table.
- ☞ After Insert: It is activated after the insertion of data into the table.
- ☞ Before Update: It is activated before the update of data in the table.
- ☞ After Update: It is activated after the update of the data in the table.
- ☞ Before Delete: It is activated before the data is removed from the table.
- ☞ After Delete: It is activated after the deletion of data from the table.

1.5 Difference between Triggers and Subprograms

Triggers	Stored Procedures
A Trigger is implicitly invoked whenever any event such as INSERT, DELETE, UPDATE occurs in a TABLE.	A Procedure is explicitly called by user/application using statements or commands such as exec or CALL procedure_name
Only nesting of triggers can be achieved in a table. We cannot define/call a trigger inside another trigger.	We can define/call procedures inside another procedure.
Transaction statements such as COMMIT, ROLLBACK, SAVEPOINT are not allowed in triggers.	All transaction statements such as COMMIT, ROLLBACK is allowed in procedures.
Triggers are used to maintain referential integrity by keeping a record of activities performed on the table.	Procedures are used to perform tasks defined or specified by the users.

We cannot return values in a trigger. Also, as an input, we cannot pass values as a parameter.	We can return 0 to n values. However, we can pass values as parameters.

1.6 Using Triggers

1.6.1 Creating Database Trigger

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

```
CREATE
    [DEFINER = user]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

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

In this syntax:

- ☞ This statement creates a new trigger. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table.
- ☞ The DEFINER clause determines the security context to be used when checking access privileges at trigger activation 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 indicates the kind of operation that activates the trigger. These trigger_event values are permitted:
    - INSERT: The trigger activates whenever a new row is inserted into the table (for example, through INSERT, LOAD DATA, and REPLACE statements).
    - UPDATE: The trigger activates whenever a row is modified (for example, through UPDATE statements).
    - DELETE: 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.

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

Points to remember:

- ☞ 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 names exist in the schema namespace, meaning that all triggers must have unique names within a schema.
- ☞ CREATE TRIGGER requires the TRIGGER privilege for the table associated with the trigger. If the DEFINER clause is present, the privileges required depend on the user value.
- ☞ The trigger_event does not represent a literal type of SQL statement that activates the trigger so much as it represents a type of table operation. For example, an INSERT trigger activates not only for INSERT statements but also LOAD DATA statements because both statements insert rows into a table.
- ☞ It is possible to define multiple triggers for a given table that have the same trigger event and action time.
  - By default, triggers that have the same trigger event and action time activate in the order they were created.
- ☞ To affect trigger order, specify a trigger_order clause that indicates FOLLOWS or PRECEDES and the name of an existing trigger that also has the same trigger event and action time.
  - With FOLLOWS, the new trigger activates after the existing trigger. With PRECEDES, the new trigger activates before the existing trigger.
- ☞ Triggers cannot use NEW.col_name or use OLD.col_name to refer to generated columns. For information about generated columns

Examples:

When inserting a row in orders table, we need add 18% GST before inserting the new row. The following trigger add 18% GST into the new price of order.

```
DELIMITER //
CREATE TRIGGER add_gst BEFORE INSERT ON orders
FOR EACH ROW
     BEGIN
          SET NEW.price = NEW.price*1.18;
     END //
DELIMITER ;
```

Example 2:

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY
KEY);
CREATE TABLE test4(
  a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  b4 INT DEFAULT 0
);
```

The following trigger is associated with table test1 and executed before insert statement. It insert same value in table test2, delete that value from test3, and update the value in table test4.

```
delimiter |
CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|
delimiter ;
```

Now insert into table test3 and test4.

```
INSERT INTO test3 (a3) VALUES
  (NULL), (NULL), (NULL), (NULL), (NULL),
  (NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES
  (0), (0), (0), (0), (0), (0), (0), (0), (0), (0);
```

Suppose that you insert the following values into table test1 as shown here:

```
INSERT INTO test1 VALUES
       (1), (3), (1), (7), (1), (8), (4), (4);
```

Now, after insert statement on table test1, retrieve data from all four tables and observe the result to identify the effect of above trigger.

Note: - 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

1.6.2 Execution of Triggers

- ☞ It is executed implicitly when DML, DDL, or system event occurs.
- ☞ The act of executing a trigger is also known as firing the trigger.
- ☞ When multiple triggers are to be executed, they have to be executed in a sequence.
- ☞ Database triggers execute with the privileges of the owner, not the current user.

1.6.3 Managing Triggers

Some DBS vendors provide modification in existing triggers such as ENABLE or DISABLE a trigger and ALTER trigger. MySQL does not provide ALTER TRIGGER and ENABLE or DISABLE facility.

1.6.4 Removing Triggers

To destroy the trigger, use a DROP TRIGGER statement. You must specify the schema name if the trigger is not in the default schema:

```
DROP TRIGGER testref;
```

1.6.5 Create Multiple Triggers

Here is the syntax for defining a trigger that will activate before or after an existing trigger in response to the same event and action time:

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE|AFTER}{INSERT|UPDATE|DELETE}
ON table_name FOR EACH ROW
{FOLLOWS|PRECEDES} existing_trigger_name
BEGIN
    -- statements
END$$

DELIMITER ;
```

- ☞ The FOLLOWS allow the new trigger to activate after an existing trigger.
- ☞ The PRECEDES allow the new trigger to activate before an existing trigger.

2 Indexing

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows.

Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; MEMORY tables also support hash indexes; InnoDB uses inverted lists for FULLTEXT indexes.

MySQL uses indexes for these operations:

- ☞ To find the rows matching a WHERE clause quickly.
- ☞ To eliminate rows from consideration.
  - If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows.
- ☞ If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows.
- ☞ To retrieve rows from other tables when performing joins. MySQL can use indexes on columns more efficiently if they are declared as the same type and size.
- ☞ To sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index.

2.1 CREATE INDEX Statement

- ☞ Normally, you create all indexes on a table at the time the table itself is created with CREATE TABLE.
  - especially important for InnoDB tables, where the primary key determines the physical layout of rows in the data file.
- ☞ CREATE INDEX enables you to add indexes to existing tables.
- ☞ CREATE INDEX is mapped to an ALTER TABLE statement to create indexes.
- ☞ CREATE INDEX cannot be used to create a PRIMARY KEY; use ALTER TABLE instead.
- ☞ A key_part specification can end with ASC or DESC to specify whether index values are stored in ascending or descending order.
  - ASC and DESC are not permitted for HASH indexes.
  - ASC and DESC are also not supported for multi-valued indexes.
  - As of MySQL 8.0.12, ASC and DESC are not permitted for SPATIAL indexes.

CREATE INDEX Statement Syntax: -

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (key_part,...)
    [index_option]
    [algorithm_option | lock_option] ...

key_part: {col_name [(length)] | (expr)} [ASC | DESC]

index_option: {
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  | ENGINE_ATTRIBUTE [=] 'string'
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
}

index_type:
    USING {BTREE | HASH}

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:

    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

The following sections describe different aspects of the CREATE INDEX statement:

2.1.1 Column Prefix Key Parts

For string columns, indexes can be created that use only the leading part of column values, using col_name(length) syntax to specify an index prefix length:

- ☞ Prefixes can be specified for CHAR, VARCHAR, BINARY, and VARBINARY key parts.
- ☞ Prefixes must be specified for BLOB and TEXT key parts. Additionally, BLOB and TEXT columns can be indexed only for InnoDB, MyISAM, and BLACKHOLE tables.
- ☞ Prefix limits are measured in bytes. However, prefix lengths for index specifications in CREATE TABLE, ALTER TABLE, and CREATE INDEX statements are interpreted as number of characters for nonbinary string types (CHAR, VARCHAR, TEXT) and number of bytes for binary string types (BINARY, VARBINARY, BLOB).

If a specified index prefix exceeds the maximum column data type size, CREATE INDEX handles the index as follows:

- ☞ For a nonunique index, either an error occurs (if strict SQL mode is enabled), or the index length is reduced to lie within the maximum column data type size and a warning is produced (if strict SQL mode is not enabled).

- ☞ For a unique index, an error occurs regardless of SQL mode because reducing the index length might enable insertion of nonunique entries that do not meet the specified uniqueness requirement.

The statement shown here creates an index using the first 10 characters of the name column (assuming that name has a nonbinary string type):

```
CREATE INDEX part_of_name ON customer (name(10));
```

2.1.2 Functional Key Parts

A "normal" index indexes column values or prefixes of column values. For example, in the following table, the index entry for a given t1 row includes the full col1 value and a prefix of the col2 value consisting of its first 10 characters:

```
CREATE TABLE t1 (
  col1 VARCHAR(10),
  col2 VARCHAR(20),
  INDEX (col1, col2(10))

);
```

MySQL 8.0.13 and higher supports functional key parts that index expression values rather than column or column prefix values. Use of functional key parts enables indexing of values not stored directly in the table. Examples:

```
CREATE TABLE t1 (col1 INT, col2 INT, INDEX func_index ((ABS(col1))));
CREATE INDEX idx1 ON t1 ((col1 + col2));
CREATE INDEX idx2 ON t1 ((col1 + col2), (col1 - col2), col1);

ALTER TABLE t1 ADD INDEX ((col1 * 40) DESC);
```

- ☞ An index with multiple key parts can mix nonfunctional and functional key parts.
- ☞ ASC and DESC are supported for functional key parts.

Functional key parts must adhere to the following rules.

- ☞ In index definitions, enclose expressions within parentheses to distinguish them from columns or column prefixes.

  ```
  INDEX ((col1 + col2), (col3 - col4))
  ```

  This produces an error; the expressions are not enclosed within parentheses:

  ```
  INDEX (col1 + col2, col3 - col4)
  ```
- ☞ A functional key part cannot consist solely of a column name. For example, this is not permitted:

  ```
  INDEX ((col1), (col2))
  ```

  Instead, write the key parts as nonfunctional key parts, without parentheses:

  ```
  INDEX (col1, col2)
  ```
- ☞ A functional key part expression cannot refer to column prefixes.
- ☞ Functional key parts are not permitted in foreign key specifications.

- ☞ For CREATE TABLE ... LIKE, the destination table preserves functional key parts from the original table

Functional indexes are implemented as hidden virtual generated columns, which has these implications:

- ☞ Each functional key part counts against the limit on total number of table columns.
- ☞ Functional key parts inherit all restrictions that apply to generated columns. Examples:
- ☞ Only functions permitted for generated columns are permitted for functional key parts.
- ☞ Subqueries, parameters, variables, stored functions, and user-defined functions are not permitted.

2.1.3 Unique Indexes

A UNIQUE index creates a constraint such that all values in the index must be distinct. An error occurs if you try to add a new row with a key value that matches an existing row. If you specify a prefix value for a column in a UNIQUE index, the column values must be unique within the prefix length. A UNIQUE index permits multiple NULL values for columns that can contain NULL.

2.1.4 Multi-Valued Indexes

You can create a multi-valued index in a CREATE TABLE, ALTER TABLE, or CREATE INDEX statement. This requires using CAST(... AS ... ARRAY) in the index definition, which casts same-typed scalar values in a JSON array to an SQL data type array. A virtual column is then generated transparently with the values in the SQL data type array; finally, a functional index (also referred to as a virtual index) is created on the virtual column. It is the functional index defined on the virtual column of values from the SQL data type array that forms the multi-valued index.

The following example a multi-valued index zips can be created on an array $.zipcode on a JSON column custinfo in a table named customers. In each case, the JSON array is cast to an SQL data type array of UNSIGNED integer values.

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    custinfo JSON,
    INDEX zips( (CAST(custinfo->'$.zip' AS UNSIGNED
ARRAY)) )
    );
```

A multi-valued index can also be defined as part of a composite index. This example shows a composite index that includes two single-valued parts (for the id and modified columns), and one multi-valued part (for the custinfo column):

ALTER TABLE customers ADD INDEX comp(id, modified,

(CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)));

Now inset values in customer table:

```
INSERT INTO customers VALUES
    (NULL, NOW(),
'{"user":"Jack","user_id":37,"zipcode":[94582,94536]}'),
    (NULL, NOW(),
'{"user":"Jill","user_id":22,"zipcode":[94568,94507,94582]
}'),
    (NULL, NOW(),
'{"user":"Bob","user_id":31,"zipcode":[94477,94507]}'),
    (NULL, NOW(),
'{"user":"Mary","user_id":72,"zipcode":[94536]}'),
    (NULL, NOW(),
'{"user":"Ted","user_id":56,"zipcode":[94507,94582]}');
```

The optimizer uses a multi-valued index to fetch records when the following functions are specified in a WHERE clause:

Using MEMBER OF()

```
SELECT * FROM customers
    WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
```

Using JSON_CONTAINS()

```
SELECT * FROM customers
    WHERE JSON_CONTAINS(custinfo->'$.zipcode',
CAST('[94507,94582]' AS JSON));
```

Using JSON_OVERLAPS()

```
SELECT * FROM customers
    WHERE JSON_CONTAINS(custinfo->'$.zipcode',
CAST('[94507,94582]' AS JSON));
```

2.1.5 Spatial Indexes

The MyISAM, InnoDB, NDB, and ARCHIVE storage engines support spatial columns such as POINT and GEOMETRY. However, support for spatial column indexing varies among engines. Spatial and nonspatial indexes on spatial columns are available according to the following rules.

Spatial indexes on spatial columns have these characteristics:

- ☞ Available only for InnoDB and MyISAM tables. Specifying SPATIAL INDEX for other storage engines results in an error.
- ☞ An index on a spatial column must be a SPATIAL index. The SPATIAL keyword is thus optional but implicit for creating an index on a spatial column.
- ☞ Available for single spatial columns only. A spatial index cannot be created over multiple spatial columns.
- ☞ Indexed columns must be NOT NULL.
- ☞ Column prefix lengths are prohibited. The full width of each column is indexed.
- ☞ Not permitted for a primary key or unique index.

Nonspatial indexes on spatial columns (created with INDEX, UNIQUE, or PRIMARY KEY) have these characteristics:

- ☞ Permitted for any storage engine that supports spatial columns except ARCHIVE.
- ☞ Columns can be NULL unless the index is a primary key.
- ☞ The index type for a non-SPATIAL index depends on the storage engine. Currently, B-tree is used.
- ☞ Permitted for a column that can have NULL values only for InnoDB, MyISAM, and MEMORY tables.

2.1.6 Index Options

Following the key part list, index options can be given. An index_option value can be any of the following:

- ❖ KEY_BLOCK_SIZE [=] value

For MyISAM tables, KEY_BLOCK_SIZE optionally specifies the size in bytes to use for index key blocks.

- ❖ index_type

Some storage engines permit you to specify an index type when creating an index.

Storage Engine	Permissible Index Types
InnoDB	BTREE

Storage Engine	Permissible Index Types
MyISAM	BTREE
MEMORY/HEAP	HASH, BTREE
NDB	HASH, BTREE

The index_type clause cannot be used for FULLTEXT INDEX or (prior to MySQL 8.0.12) SPATIAL INDEX specifications

- WITH PARSER parser_name

This option can be used only with FULLTEXT indexes. It associates a parser plugin with the index if full-text indexing and searching operations need special handling. InnoDB and MyISAM support full-text parser plugins.

- COMMENT 'string'

Index definitions can include an optional comment of up to 1024 characters.

- VISIBLE, INVISIBLE

Specify index visibility. Indexes are visible by default. An invisible index is not used by the optimizer. Specification of index visibility applies to indexes other than primary keys.

2.1.7 Table Copying and Locking Options

ALGORITHM and LOCK clauses may be given to influence the table copying method and level of concurrency for reading and writing the table while its indexes are being modified. They have the same meaning as for the ALTER TABLE statement.

2.2 Dropping an Index

DROP INDEX drops the index named index_name from the table tbl_name. This statement is mapped to an ALTER TABLE statement to drop the index.

```
DROP INDEX index_name ON tbl_name
    [algorithm_option | lock_option] ...

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:

    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

To drop a primary key, the index name is always PRIMARY, which must be specified as a quoted identifier because PRIMARY is a reserved word:

```
DROP INDEX `PRIMARY` ON t;
```

3 Exercise

1. Create a trigger on the table employees, which after an update or insert, converts all the values of first and last names to upper case. (Hint: Use cursors to retrieve the values of each row and modify them.)
2. Create a trigger that restores the values before an update operation on the employees table if the salary exceeds 100000. (Hint: Use the inserted and deleted tables to look at the old and new values in the table respectively.)

**********END**********