Views, Triggers, Functions- Calling a function, return type, Stored Procedures, Indexing and Joins

JOINS

- Joins are used to query data from two or more tables, based on a relationship between certain columns in these tables

- Types of Joins

LEFT JOIN:
Return all rows from the left table, even if there are no matches in the right table.

RIGHT JOIN:
Return all rows from the right table, even if there are no matches in the left table

FULL JOIN: Return rows when there is a match in one of the tables

INNER JOIN

- The INNER JOIN keyword returns rows when there is at least one match in both tables

- **INNER JOIN Syntax**

```
SELECT column_name(s)
FROM  table_name1
INNER JOIN  table_name2
ON table_name1.column_name=table_name2.column_name;
```

- If there are rows in “table_name1 " that do not have matches in " table_name2 ", those rows will NOT be listed

LEFT JOIN

- The LEFT JOIN keyword returns all rows from the left table (table_name1), even if ther are no matches in the right table (table_name2)

- **LEFT JOIN Syntax**

```
SELECT column_name(s)
FROM table_name1
LEFT JOIN table_name2
ON table_name1.column_name=table_name2.column_name;
```

RIGHT JOIN

- The RIGHT JOIN keyword returns all the rows from the right table (table_name2), even if there are no matches in the left table (table_name1)

- **RIGHT JOIN Syntax**

```
SELECT column_name(s)
FROM table_name1
RIGHT JOIN table_name2
```

FULL JOIN

- The FULL JOIN keyword return rows when there is a match in one of the tables

FULL JOIN Syntax

```
SELECT column_name(s)
FROM table_name1
FULL JOIN table_name2
ON table_name1.column_name=table_name2.column_name;
```

- The FULL JOIN keyword returns all the rows from the left table (Table1), and all the rows from the right table (Table2). If there are rows in " Table1 " that do not have matches in " Table2", or if there are rows in " Table2" that do not have matches in " Table1 ", those rows will be listed as well.

VIEWS

- The **view** is a virtual table. It does not physically exist. Rather, it is created by a query joining one or more tables.
- 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

Creating an SQL VIEW

Syntax:

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

View Creation - Example

- **View Creation - Example**

  CREATE VIEW sup_orders

  AS SELECT suppliers.supplier_id, orders.quantity, orders.price

  FROM suppliers, orders

  WHERE suppliers.supplier_id = orders.supplier_id and suppliers.supplier_name = 'IBM';

- This View (Create statement) would create a virtual table based on the result set of the select statement. You can now query the view as follows

  SELECT * FROM sup_orders;

Updating VIEW

- You can modify the definition of a VIEW without dropping it by using the following syntax

  CREATE OR REPLACE VIEW view_name

  AS SELECT columns

  FROM table

  WHERE predicates;

- **View Modify - Example**

  CREATE or REPLACE VIEW sup_orders

  AS SELECT suppliers.supplier_id, orders.quantity, orders.price

  FROM suppliers, orders

  WHERE suppliers.supplier_id = orders.supplier_id

  and suppliers.supplier_name = 'Microsoft';

Dropping VIEW

- The syntax for dropping a VIEW :

```
DROP  VIEW view_name;
```

- **View Drop - Example**

```
DROP  VIEW  sup_orders;
```

Question: Can you update the data in an view?

Answer : A view is created by joining one or more tables. When you update record(s) in a view, it updates the records in the underlying tables that make up the View.

So, yes, you can update the data in View providing you have the proper privileges to the underlying tables.

Question: Does the SQL View exist if the table is dropped from the database?

Answer: Yes, View continues to exist even after one of the tables (that the SQL View is based on)

Functions

- A Stored Function in MySQL is a user-defined function that you create and store in the database.
- It performs a calculation or operation and returns a single value.
- Unlike Stored Procedures, functions must return a value and can be used inside SQL queries (like built-in functions SUM(), ROUND(), etc.).

```
CREATE FUNCTION function_name(func_parameter1, func_parameter2, ..)
        RETURN datatype [characteristics]
        func_body
```

```
CREATE FUNCTION function_name (parameters)
RETURNS data_type
[DETERMINISTIC | NOT DETERMINISTIC]
BEGIN
    -- declare local variables
    DECLARE variable_name data_type;

    -- SQL statements  as logic to compute the result
    RETURN value;
END $$
DELIMITER ;
```

CREATE FUNCTION function_name = defines the function name.

(parameters) = input parameters (optional).

RETURNS data_type = defines what type of value will be returned (e.g., INT, DECIMAL, VARCHAR).

DETERMINISTIC = means function always returns the same result for the same input (recommended).

DECLARE = used for local variables inside function.

RETURN = returns the final value.

Why do we use DELIMITER in MySQL?

- By default, MySQL uses ; (semicolon) as the end of a statement.

But when we create a Stored Procedure or Stored Function, the code itself contains many ; inside the body (for multiple SQL statements).

- If we don't change the delimiter, MySQL will think the function ended at the first ;, and throw an error.

- DELIMITER $$ → tells MySQL: "Don't end the statement at ;, wait until you see $$."
- END$$ → marks the true end of the function.

Eg:

Function to return the square of a number:

```
DELIMITER $$

CREATE FUNCTION GetSquare (num INT)
RETURNS INT
DETERMINISTIC
BEGIN
    RETURN num * num;
END$$

DELIMITER ;

-- Calling of function below:
SELECT GetSquare(6) AS SquareResult;
```

Calling Functions

- A function call is an expression with the following syntax:

  ***function-object*(*arguments*)**

- ***function-object*** It is most often the function's name.
- The parentheses denote the function-call operation itself.
- ***Arguments***, in the simplest case, is a series of zero or more expressions separated by commas (,), giving values for the function's corresponding formal parameters

Stored procedure

- In a database management system (DBMS), a stored procedure is a set of Structured Query Language (SQL) statements with an assigned name that's stored in the database in compiled form so that it can be shared by a number of programs.

- preserving data integrity (information is entered in a consistent manner)

Data integrity means the correctness and consistency of data

Enforcing data integrity ensures the quality of the data in the database

Consider following two examples of data integriry in a database

1) If an employee is entered with an employee_id value of 123, the database should not allow another employee to have an ID with the same value

2) If you have an employee_rating column intended to have values ranging from 1 to 5, the database should not accept a value of 6

TRIGGER

- A trigger is a block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.
- Syntax of Triggers:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- Trigger body (logic to execute)
END;
```

trigger_name → unique name for the trigger

BEFORE/AFTER → timing of execution

INSERT/UPDATE/DELETE → event type

FOR EACH ROW → executes once per row affected

NEW → refers to the new row being inserted/updated

OLD → refers to the existing row before update/delete

MySQL supports **6 trigger types** (based on timing and event):

BEFORE INSERT – Executes before inserting data

AFTER INSERT – Executes after inserting data

BEFORE UPDATE – Executes before updating data

AFTER UPDATE – Executes after updating data

BEFORE DELETE – Executes before deleting data

AFTER DELETE – Executes after deleting data

(1) Meaning of NEW

- NEW refers to the new row values being inserted or updated.
- You can use it in INSERT and UPDATE triggers.

(2) Meaning of OLD

- OLD refers to the existing row values before update or delete.
- You can use it in UPDATE and DELETE triggers.

When you can use NEW / OLD

INSERT Trigger → only NEW (since no old row exists yet).

UPDATE Trigger → both NEW and OLD available.

DELETE Trigger → only OLD (since no new row is created).

TRIGGER

- The syntax for a dropping a Trigger is:

DROP TRIGGER *trigger_name ON tbl_name;*

- The syntax for a disabling a Trigger is:

ALTER TRIGGER trigger_name DISABLE;

- The syntax for a enabling a Trigger is:

Example

- Creating Trigger

```
CREATE TRIGGER deleted_detailss
BEFORE
DELETE on tbl_customer
  FOR EACH ROW
EXECUTE PROCEDURE customerss_delete();
```

- Drop Trigger

```
drop trigger delete_details on tbl_customer;
```

CURSOR

- **cursor** in MySQL is a **database object** that allows you to **fetch rows one by one** from a query result set, and then process each row individually inside a stored procedure or function.

- Normally, SQL works on **sets of data** (all rows at once). But sometimes you need **row-by-row processing**, and that's where **cursors** are used.

- While standard d SQL queries usually operate on data sets, cursors perform operations on one row at a tim

- This can be very useful for complicated data manipulations and procedural logic.

Why do we need cursors?

- When you need to iterate through query results row by row.
- When a task cannot be easily solved with a single SQL query.
- Common in reporting, batch processing, and data migration.
- Example: Sending personalized emails to each customer, updating stock one product at a time, calculating totals per customer.
- Cursors = row-by-row iteration tool in MySQL.
- Needed when you want loop-like behavior in SQL.
- Useful for batch processing, reporting, or automation tasks.

Syntax:

```
-- Step 1: Declare variables (to hold values from cursor)
DECLARE variable_name datatype;

-- Step 2: Declare the cursor
DECLARE cursor_name CURSOR FOR
   SELECT column1, column2, ...
   FROM table_name  WHERE condition;

-- Step 3: Declare a handler (to handle end of cursor)
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

-- Step 4: Open the cursor
OPEN cursor_name;

-- Step 5: Fetch rows in a loop
FETCH cursor_name INTO variable1, variable2, ...;

-- Step 6: Close the cursor
CLOSE cursor_name;
```

Exception Handling

- In MySQL, exceptions (or errors) are handled inside stored programs (procedures, functions, triggers, events) using handlers.
- When something goes wrong (like division by zero, duplicate key, no data found), MySQL raises an error.
- Instead of letting the program crash, you can catch the error using a handler and decide what to do.

Declaring a Handler

Syntax:

DECLARE **action** HANDLER FOR **condition** VALUE **statement;**

action → What MySQL should do when the exception occurs.

CONTINUE → Continue executing the next statement.

EXIT → Exit the current block immediately.

UNDO → (Not supported in MySQL, only in other SQL standards).

condition → The error you want to handle.

SQLEXCEPTION → Handles all errors.

SQLWARNING → Handles all warnings.

NOT FOUND → Handles cases when no rows are found (common with cursors).

Specific error codes (like 1062 for duplicate entry).

statement → What to execute when the error occurs (can be SET, INSERT, etc.).

Custom Error Handling

SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Not enough stock';

(*Meaning: Raise a custom error with SQLSTATE code 45000, and display the message "Not enough stock".)*

(a) SIGNAL

SIGNAL is a MySQL statement used to raise (throw) an error intentionally.

It allows you to create your own custom error messages instead of relying only on system errors.

(b) SQLSTATE '45000'

It is a 5-character code that represents an error condition.'45000' is a generic SQLSTATE code for "unhandled user defined exception". You can use it whenever you want to signal a custom error that doesn't fit into other predefined codes.

Think of it like saying: "This is a custom error I'm raising."

(c) SET MESSAGE_TEXT = 'Not enough stock'

This defines the actual error message that MySQL should display when the exception is raised.

In this case, if someone tries to order more than available stock,

INDEX

- The CREATE INDEX statement is used to create indexes in tables
- Indexes allow the database application to find data fast; without reading the whole table
- An index can be created in a table to find data more quickly and efficiently
- The users cannot see the indexes, they are just used to speed up searches/queries

INDEX

- **CREATE INDEX Syntax**

  Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column_name);
```

- **CREATE UNIQUE INDEX Syntax**

  Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name);
```

- UNIQUE indicates that the combination of values in the indexed columns must be

INDEX

- **Rename an Index**

  The syntax for renaming an index is:

  ```
  ALTER INDEX index_name RENAME TO new_index_name;
  ```

- **Drop an Index**

  The syntax for dropping an index is:

  ```
  DROP INDEX index_name;
  ```