Unit II

Advanced MySQL

- ## Control Statements:

In advanced MySQL, control flow statements are used within stored procedures, functions, or triggers to control the flow of execution based on conditions, loops, or branching.
These statements allow you to execute different pieces of code depending on conditions, iterate over collections, or exit/continue loops.
These control flow statements are essential for implementing complex logic within MySQL.

Here is a detailed explanation of the most commonly used control flow statements in advanced MySQL:

1. IF Statement

The IF statement is used for conditional branching in MySQL. It allows you to execute a block of SQL statements based on a specified condition.

Syntax:

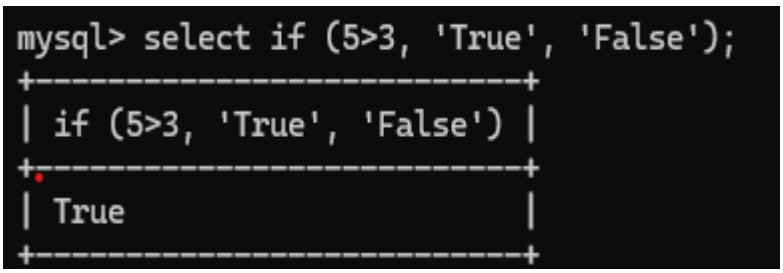
```
IF condition THEN
  -- Statements if condition is true
ELSE
  -- Statements if condition is false
END IF;
```

Example of an IF statement in MySQL:

a. Using IF inside a MySQL query :
Query :

```
select if (5>3, 'True', 'False');
```

Output :



It will return ‘True’ because the condition 5 > 3 is true.

b. Using IF inside a stored procedure in MySQL :

For create procedure in mysql command line client first we want to delimiter $$ as follows:

```
mysql> delimiter $$
mysql>
```

DELIMITER $$ is used to change the statement delimiter because the default semicolon (;) is used to terminate each statement in MySQL, and you need to use a different delimiter to define stored procedures that contain multiple statemets.

Now, create a procedure,

Source Code :

```
create procedure check_discount(in total_amount int)
 begin
 if total_amount > 100 then
 select 'Discount Applied';
 else
 select 'No Discount';
 end if;
 end $$
```

After creating a procedure apply DELIMITER ; which is used to change statement delimiter because the default $$ is used to terminate each statement.

Use delimiter ; as follows :

```
mysql> delimiter ;
mysql>
```

Call the Stored Procedure :

After creating the procedure, you can call it with different values :

```
call check_discount(90);
```

Output :

```
mysql> call check_discount(90);
+-------------+
| No Discount |
+-------------+
| No Discount |
+-------------+
```

```
call check_discount(110);
```

Output :

```
mysql> call check_discount(110);
+------------------+
| Discount Applied |
+------------------+
| Discount Applied |
+------------------+
```

In this example,
If the total_amount is greater than 100, the message "Discount Applied" is returned.
If the condition is not true, the message "No Discount" is returned.

2. Case statement :

The CASE statement is used when you need to check multiple conditions and execute different results based on the condition that is met. It is similar to an IF-ELSE block but is more concise and easier to read when you have multiple conditions.

Syntax:

```
SELECT CASE
   WHEN condition1 THEN result1
   WHEN condition2 THEN result2
   ELSE result3
END;
```

Use DELIMITER $$

```
mysql> delimiter $$
mysql>
```

Example :
Source Code :

```
CREATE PROCEDURE check_grade(IN score INT)
BEGIN
   SELECT CASE
      WHEN score >= 90 THEN 'A'
      WHEN score >= 80 THEN 'B'
      WHEN score >= 70 THEN 'C'
      WHEN score >= 60 THEN 'D'
      ELSE 'F'
   END AS grade;
```

END $$

Use delimiter ;

```
mysql> delimiter ;
mysql>
```

Call the Procedure :
Call check_grade(65);

Output :

```
mysql> call check_grade(65);
+-------+
| grade |
+-------+
| D     |
+-------+
1 row in set (0.01 sec)
```

The CASE statement checks the value of score and returns a corresponding grade based on the conditions provided.

3. Loop statement :

The LOOP statement allows you to execute a block of code repeatedly until an explicit exit condition is met. The LEAVE statement is used to exit the loop.

Syntax:

```
LOOP
   -- Statements to be repeated
   IF condition THEN
      LEAVE;  -- Exits the loop
   END IF;
END LOOP;
```

Use DELIMITER $$

```
mysql> delimiter $$
mysql>
```

Example :

```
CREATE PROCEDURE print_numbers()
BEGIN
```

```
    DECLARE counter INT DEFAULT 1;

    loop_start:
    LOOP
        SELECT counter;
        SET counter = counter + 1;

        IF counter > 5 THEN
            LEAVE loop_start;
        END IF;
    END LOOP;
END $$
```

Use delimiter ;

```
mysql> delimiter ;
mysql>
```

Call the Procedure :

call print_numbers( );

Or

call print_numbers;

```
mysql> call print_numbers();
+---------+
| counter |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)

+---------+
| counter |
+---------+
|       2 |
+---------+
1 row in set (0.00 sec)

+---------+
| counter |
+---------+
|       3 |
+---------+
1 row in set (0.00 sec)

+---------+
| counter |
+---------+
|       4 |
+---------+
1 row in set (0.00 sec)

+---------+
| counter |
+---------+
|       5 |
+---------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

* Block Structure in MySQL

In MySQL, block structures refer to sections of code or logic that can be grouped together, often in the form of stored procedures or functions. These blocks allow you to encapsulate multiple SQL statements into a single unit, which can be executed or called in various contexts. They are useful for organizing complex logic, reusing code, and improving performance by reducing the need for repetitive SQL execution.

The block structure typically includes:

1. **Variable Declarations**: Declaring variables inside the block.
2. **Control Flow**: Using conditional statements (IF, LOOP, etc.) and error handling.
3. **Execution Statements**: The SQL queries that are executed within the block.

The stored procedure is one of the primary ways to implement block structures in MySQL.

Stored Procedures in MySQL

A stored procedure is a set of SQL statements that can be stored and executed in MySQL. It is created using the CREATE PROCEDURE statement and can be executed using the CALL statement.

Stored procedures can accept parameters (input, output, or input-output) to make them more dynamic.

Basic Syntax of Creating a Stored Procedure:

```
CREATE PROCEDURE procedure_name (parameter1 datatype, parameter2 datatype, ...)
BEGIN
   -- SQL statements go here
   -- You can declare variables, loops, and control flow
END;
```

Parameters:

1. **IN**: Input parameters, used to pass values into the procedure.
2. **OUT**: Output parameters, used to return values from the procedure.
3. **INOUT**: Used for both input and output.

Creating and Executing Stored Procedures Without Parameters

1. Create a Stored Procedure Without Parameters:

Here is an example of a simple stored procedure that doesn't take any parameters:

```
DELIMITER $$

CREATE PROCEDURE show_message()
BEGIN
   SELECT 'Hello, World!' AS message;
END $$

DELIMITER ;
```

- DELIMITER $$ changes the statement delimiter temporarily to $$ so that you can define the procedure without MySQL interpreting the semicolons (;) inside the procedure body.
- DELIMITER ; restores the delimiter back to ;.

2. Executing the Stored Procedure Without Parameters:

```
CALL show_message();
```

Output:

```
mysql> CALL show_message();
+---------------+
| message       |
+---------------+
| Hello, World! |
+---------------+
1 row in set (0.04 sec)

Query OK, 0 rows affected (0.04 sec)
```

This will execute the procedure and display the message "Hello, World!".

Creating and Executing Stored Procedures With Parameters

1. Create a Stored Procedure With Input Parameters:

Here's an example where we define a procedure that accepts an input parameter to display a personalized message:

```
DELIMITER $$
```

```
CREATE PROCEDURE greet_user(IN username VARCHAR(50))
BEGIN
   SELECT CONCAT('Hello, ', username, '!') AS greeting;
END $$

DELIMITER ;
```

- In this example, the procedure greet_user takes an input parameter username of type VARCHAR(50) and concatenates it with a greeting message.

2. Executing the Stored Procedure With Parameters:

You can call the stored procedure and pass the value for the input parameter:

```
CALL greet_user('Alice');
```

Output:

```
mysql> CALL greet_user('Alice');
+---------------+
| greeting      |
+---------------+
| Hello, Alice! |
+---------------+
1 row in set (0.01 sec)
```

In this example, the procedure greet_user takes an input parameter username of type VARCHAR(50) and concatenates it with a greeting message.

Stored Procedure with Output Parameters

You can also create a procedure that returns an output parameter.

1. Create a Stored Procedure With Output Parameters:

```
DELIMITER $$

CREATE PROCEDURE get_user_info(IN user_id INT, OUT user_name VARCHAR(50))
BEGIN
   SELECT name INTO user_name
   FROM users
   WHERE id = user_id;
END $$

DELIMITER ;
```

- This example defines a procedure get_user_info that takes an input parameter user_id and returns the corresponding user_name as an output parameter.

2. Executing the Stored Procedure With Output Parameters:

First, declare a variable to store the output, then call the procedure:

-- **Declare a variable to store the output**
SET @username = '';

-- **Execute the procedure**
CALL get_user_info(1, @username);

-- **Show the result**
SELECT @username;

```
mysql> SET @username = 'ram';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL get_user_info(1, @username);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @username;
+-----------+
| @username |
+-----------+
| ram       |
+-----------+
1 row in set (0.00 sec)
```

This will execute the procedure, retrieve the user_name for user_id = 1, and store it in the @username variable, which is then displayed.

Example of a Stored Procedure with Input and Output Parameters

You can also define a stored procedure with both input and output parameters.

1. Create a Stored Procedure with Input and Output Parameters:

```
DELIMITER $$

CREATE PROCEDURE update_user_info(IN user_id INT, IN new_name VARCHAR(50),
OUT updated_name VARCHAR(50))
BEGIN
   UPDATE users
   SET name = new_name
   WHERE id = user_id;
```

```
    SELECT name INTO updated_name
    FROM users
    WHERE id = user_id;
END $$

DELIMITER ;
```

2. **Executing the Stored Procedure with Both Input and Output Parameters:**

```
-- Declare a variable to store the output
SET @updated_name = '';

-- Execute the procedure
CALL update_user_info(1, 'NewName', @updated_name);

-- Show the updated name
SELECT @updated_name;
```

This procedure updates the name of the user with user_id = 1 and returns the updated name as the output parameter.

```
mysql> SET @updated_name = '';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL update_user_info(1, 'NewName', @updated_name);
Query OK, 1 row affected (0.33 sec)

mysql> SELECT @updated_name;
+---------------+
| @updated_name |
+---------------+
| NewName       |
+---------------+
1 row in set (0.00 sec)
```

Summary of Execution Process

1. Create the stored procedure using CREATE PROCEDURE.
2. Execute the procedure using CALL procedure_name().
3. Pass parameters if required when calling the procedure.

Stored procedures allow for cleaner, more reusable code, and they can encapsulate complex logic and interactions with the database.

* The MySQL Cursors -

A MySQL cursor is a database object that enables the end-user to retrieve, process, and scroll through rows of the result set one at a time.

A MySQL cursor is a pointer that is used to iterate through a table's records. They are used within stored programs such as procedures and functions and have the following features −

A. READ ONLY − Cursors only allow you to read data; you can't make changes to it.

B. Non-Scrollable − Cursors move through records in one direction, from the top to the bottom.

C. Asensitive − Cursors are sensitive to the changes made in the table. Any modification done in the table will be reflected in the cursor.

The following four steps/operations are used to manage cursors in MySQL:

1. Declare Cursor
2. Open Cursor
3. Fetch Cursor
4. Close Cursor

1. Declare Cursor

The DECLARE statement is used to declare a cursor in a MySQL. Once declared, it is then associated with a SELECT statement to retrieve the records from a table.

syntax :-

```
DECLARE cursor_name CURSOR FOR select_statement;
```

- **cursor_name**: This will be the name given to your cursor.
- **select_statement**: SQL statement to define a result set for the cursor.

2. Open Cursor

The OPEN statement is used to initialize the cursor to retrieve the data after it has been declared.

syntax :-

```
OPEN cursor_name;
```

3. Fetch Cursor

The FETCH statement is then used to retrieve the record pointed by the cursor. Once retrieved, the cursor moves to the next record.

syntax :-

```
FETCH cursor_name INTO variable_list;
                    OR
FETCH cursor_name INTO variable1, variable2, ...;
```

- cursor_name : Name of the cursor.
- variable1, variable2, … : Variables in which the fetched data has to be stored.

4. Close Cursor

The CLOSE statement is used to release the memory associated with the cursor after all the records have been retrieved.

syntax :-

```
CLOSE cursor_name;
```

Examples :-

Examples :-

1. **Employees sample table -:**

Create table:-

```
CREATE TABLE employees ( id INT,
    name VARCHAR(50),
    salary INT
);
```

Insert Record -

```
INSERT INTO employees VALUES (1, 'Alice',
50000),
(2, 'Bob', 60000),
(3, 'Charlie', 55000);
```

Cursor example (Stored Procedure)

```
DELIMITER $$
CREATE PROCEDURE simple_employee_cursor() BEGIN
    DECLARE done INT DEFAULT 0; DECLARE
    emp_name VARCHAR(50);
    DECLARE emp_salary INT;
    -- Cursor declaration
    DECLARE emp_cursor CURSOR FOR SELECT name,
```

```
    salary FROM employees;
  -- End of cursor handler
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
  -- Open cursor OPEN
  emp_cursor;
  -- Loop through rows read_loop: LOOP
    FETCH emp_cursor INTO emp_name, emp_salary; IF done
    THEN
      LEAVE read_loop; END IF;
    -- Display employee info
    SELECT emp_name AS Name, emp_salary AS Salary; END LOOP;
 -- Close cursor

 CLOSE emp_cursor; END$$
DELIMITER ;
```

Call the procedure:-

```
CALL simple_employee_cursor();
```

The procedure will:
Open the cursor.
Fetch the first row: (Alice, 50000) → SELECT 'Alice' AS Name, 50000 AS Salary; Fetch the second

row: (Bob, 60000) → SELECT 'Bob' AS Name, 60000 AS Salary; Fetch the third row: (Charlie,

55000) → SELECT 'Charlie' AS Name, 55000 AS Salary; Done → exit loop and close cursor.

output-

Name	Salary
Alice	50000
Bob	60000
Charlie	55000

2.Example

Step 1: Create the product table

```
CREATE TABLE product ( product_id INT
  PRIMARY KEY, product_name
  VARCHAR(50), price DECIMAL(10,2)
);
```

Step 2: Insert sample data

```
INSERT INTO product (product_id, product_name, price) VALUES (1, 'Laptop',
800.00),
```

(2, 'Smartphone', 500.00),
(3, 'Headphones', 150.00);

Step 3: Create a cursor to iterate over products

We'll create a stored procedure that uses a cursor to print the products whose price is above 200.

```
DELIMITER //
CREATE PROCEDURE list_expensive_products() BEGIN
    -- Declare variables to hold cursor values DECLARE p_id
    INT;
    DECLARE p_name VARCHAR(50); DECLARE
    p_price DECIMAL(10,2);
    -- Declare cursor
    DECLARE product_cursor CURSOR FOR
      SELECT product_id, product_name, price FROM product;
    -- Handler for end of cursor DECLARE done
    INT DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    -- Open cursor
    OPEN product_cursor; read_loop:
    LOOP
      FETCH product_cursor INTO p_id, p_name, p_price; IF done
      THEN
        LEAVE read_loop; END IF;
      -- Check condition
      IF p_price > 200 THEN
       SELECT CONCAT('Product ID: ', p_id, ', Name: ', p_name, ', Price: $', p_price) AS product_info;
      END IF; END
    LOOP;
  CLOSE product_cursor; END //
DELIMITER ;
```

Step 4: Call the stored procedure

```
CALL list_expensive_products();
```

Output :-

product_info
Product ID: 1, Name: Laptop, Price: $800.00
ProductID:2,Name:Smartphone,Price:$500.00

Explanation

Cursor Declaration:

DECLARE product_cursor CURSOR FOR SELECT ...;

This defines a cursor to loop over all rows in the product table. Handler:

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

Handles the "end of data" situation so the loop knows when to stop. Fetching Data:

FETCH product_cursor INTO p_id, p_name, p_price; This fetches the

current row into variables.

Looping:
The LOOP fetches each row, checks the price, and outputs only the products with price > 200.

Closing Cursor:

CLOSE product_cursor;