**CURSOR**

A cursor in MySQL is used to iterate through a result set row by row, much like how a loop works in procedural programming. It's typically used in stored procedures when you need to process each row individually. Below are detailed notes on cursors in MySQL, including syntax, usage, and important concepts.

**1. What is a Cursor?**

- A cursor allows you to retrieve rows one at a time from a result set returned by a query.

- Unlike a normal SELECT query that fetches all results at once, a cursor allows row-by-row processing.

**2. Types of Cursors in MySQL**

- READ-ONLY: You can only fetch data using this cursor; no updates or changes to the data are allowed while the cursor is open.

- NON-SCROLLABLE: You can move only in the forward direction through the result set, i.e., from the first to the last row.

**3. Cursor Lifecycle**

A cursor has a defined lifecycle in MySQL, consisting of the following phases:

**1. DECLARE: Defines the cursor with the query to retrieve the result set.**

**2. OPEN: Executes the query and makes the result set available for retrieval.**

**3. FETCH: Retrieves the next row in the result set. This can be repeated until there are no more rows.**

**4. Handling End of Data : DECLARE CONTINUE HANDLER FOR NOT FOUND SET done TRUE;**

**5. CLOSE: Releases the cursor and any associated resources.**

**4. Cursor Syntax**

Here is the general syntax of using a cursor in MySQL:

1. Declaring a Cursor

You declare a cursor using the `DECLARE` statement. The cursor must be declared after any variable declarations and before any handlers.

```sql

DECLARE cursor\_name CURSOR FOR select\_statement;

```

- `cursor\_name`: Name of the cursor.

- `select\_statement`: A SELECT query that defines the result set.

2. Opening a Cursor

Once the cursor is declared, it needs to be opened using the `OPEN` statement.

```sql

OPEN cursor\_name;

```

3. Fetching Rows from a Cursor

Use the `FETCH` statement to retrieve the next row from the cursor.

```sql

FETCH cursor\_name INTO var1, var2, ...;

```

- The result is fetched into variables (`var1`, `var2`, etc.).

4. Closing a Cursor

After fetching all rows, you must close the cursor using the `CLOSE` statement.

```sql

CLOSE cursor\_name;

```

**5. Example: Using Cursors in a Stored Procedure**

Below is an example of how to use a cursor in a MySQL stored procedure:

```sql

DELIMITER //

CREATE PROCEDURE process\_data()

BEGIN

-- Declare variables to store cursor results

DECLARE var\_id INT;

DECLARE var\_name VARCHAR(100);

-- Declare a cursor

DECLARE my\_cursor CURSOR FOR

SELECT id, name FROM employees;

-- Declare a handler to exit the loop when there are no more rows

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

-- Open the cursor

OPEN my\_cursor;

-- Fetch rows one by one

read\_loop: LOOP

FETCH my\_cursor INTO var\_id, var\_name;

IF done THEN

LEAVE read\_loop;

END IF;

-- Process each row (e.g., print to output)

SELECT var\_id, var\_name;

END LOOP;

-- Close the cursor

CLOSE my\_cursor;

END //

DELIMITER ;

```

**6. Important Points to Note**

- Handlers for End of Data: You need to define a handler to detect when the cursor has no more rows. A common way is to use the `NOT FOUND` condition to exit the loop:

```sql

DECLARE done INT DEFAULT FALSE;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

```

- Limitations:

- Cursors cannot be used in MySQL functions, only in stored procedures.

- Cursors are read-only and non-scrollable in MySQL. You can only move forward through the result set.

- Performance Considerations:

- Cursors are generally slower than set-based operations, as each row is processed individually.

- Use them only when you cannot achieve the desired functionality with set-based queries.

7. Use Cases for Cursors

- Row-by-row processing: When you need to perform complex calculations or operations on each row.

- Batch processing: Processing large data sets in smaller chunks.

- Conditional updates or operations: Applying certain logic or updates to specific rows that meet certain conditions during iteration.

8. Best Practices

- Minimize cursor usage: If you can achieve your goal with set-based SQL operations, prefer those, as they are usually faster and more efficient.

- Close cursors: Always ensure that cursors are closed to free up resources.

**Advantages of Cursors:**

• Row-by-Row Processing: Useful for operations that require processing each row individually.

• Flexibility: More control over the processing of query results.

**Disadvantages of Cursors:**

• Performance Overhead: Cursors can be slower than set-based operations because they process rows one

at a time.

• Resource Intensive: They consume more memory and resources, especially for large datasets

Reference-

<https://dev.mysql.com/doc/refman/8.4/en/cursors.html>

**MySQL `CURSOR` examples with explanations:**

# Basic Structure of MySQL Cursor

A cursor in MySQL is used to iterate through rows of a query result, processing each row individually. Below is the basic structure:

```sql

DECLARE cursor\_name CURSOR FOR SELECT\_statement;

```

You use a `DECLARE` statement to define the cursor inside a `BEGIN ... END` block, typically in a stored procedure.

# Example 1: Simple Cursor to Loop through a Table

In this example, we’ll use a cursor to iterate over a table `employees` and retrieve the employee names:

```sql

DELIMITER $$

CREATE PROCEDURE getEmployeeNames()

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE emp\_name VARCHAR(255);

-- Declare the cursor for the query

DECLARE emp\_cursor CURSOR FOR

SELECT name FROM employees;

-- Declare a handler to exit the loop

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

-- Open the cursor

OPEN emp\_cursor;

-- Loop through all rows

read\_loop: LOOP

-- Fetch the name of the employee

FETCH emp\_cursor INTO emp\_name;

-- Exit the loop if no more rows

IF done THEN

LEAVE read\_loop;

END IF;

-- Do something with emp\_name (For now, we’ll just SELECT it)

SELECT emp\_name;

END LOOP;

-- Close the cursor

CLOSE emp\_cursor;

END$$

DELIMITER ;

```

Explanation:

1. The cursor is declared with `DECLARE emp\_cursor CURSOR FOR` and a simple `SELECT` query.

2. The `CONTINUE HANDLER FOR NOT FOUND SET done = 1;` is used to exit the loop when no more rows are found.

3. Inside the loop, we fetch each row using `FETCH emp\_cursor INTO emp\_name;`.

4. The loop exits when all rows are processed, and the cursor is closed.

# Example 2: Cursor with Parameterized Query

If you want a cursor based on a parameterized query (e.g., retrieving employees from a specific department):

```sql

DELIMITER $$

CREATE PROCEDURE getEmployeesByDepartment(dept\_id INT)

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE emp\_name VARCHAR(255);

-- Declare the cursor

DECLARE emp\_cursor CURSOR FOR

SELECT name FROM employees WHERE department\_id = dept\_id;

-- Declare a handler for no rows found

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

-- Open the cursor

OPEN emp\_cursor;

-- Loop through the rows

read\_loop: LOOP

FETCH emp\_cursor INTO emp\_name;

IF done THEN

LEAVE read\_loop;

END IF;

-- Output the employee name

SELECT emp\_name;

END LOOP;

-- Close the cursor

CLOSE emp\_cursor;

END$$

DELIMITER ;

```

Explanation:

This procedure accepts a department ID (`dept\_id`) as input and retrieves the names of employees in that department using a cursor.

# Example 3: Cursor with Update Operation

Let’s modify the previous example to update the salary of employees in a certain department:

```sql

DELIMITER $$

CREATE PROCEDURE updateSalaries(dept\_id INT, increment DECIMAL(10,2))

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE emp\_id INT;

DECLARE emp\_salary DECIMAL(10,2);

-- Declare the cursor

DECLARE emp\_cursor CURSOR FOR

SELECT id, salary FROM employees WHERE department\_id = dept\_id;

-- Declare a handler

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

-- Open the cursor

OPEN emp\_cursor;

-- Loop through each employee

read\_loop: LOOP

FETCH emp\_cursor INTO emp\_id, emp\_salary;

IF done THEN

LEAVE read\_loop;

END IF;

-- Update the salary of each employee

UPDATE employees

SET salary = emp\_salary + increment

WHERE id = emp\_id;

END LOOP;

-- Close the cursor

CLOSE emp\_cursor;

END$$

DELIMITER ;

```

Explanation:

This procedure accepts a `dept\_id` and an `increment` value. It fetches each employee in the department and updates their salary by adding the increment.