https://www.geeksforgeeks.org/mysql-cursors/

MySQL Cursors

Last Updated : 05 Aug, 2024

A **MySQL cursor** is a powerful database object designed for retrieving, processing, and managing rows from a result set one at a time. Unlike standard **SQL queries** that handle sets of rows in bulk, cursors allow for detailed row-by-row operations. In this article, We will learn about **MySQL Cursors** in detail.

What is a MySQL Cursor

- 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.
- While standard SQL queries usually operate on data sets, cursors perform operations on one row at a time.
- This can be very useful for complicated data manipulations and procedural logic.
- **Cursors** enable the user to fetch and process each row individually, allowing for detailed manipulation and analysis of data.
- By using cursors, developers can precisely control the flow of data processing, such as updating or analyzing specific rows based on custom criteria.

How to use MySQL Cursors

Declare the Cursor

A cursor is declared within a stored procedure or function using a CURSOR statement. This binds the cursor to a specified SQL query.

```
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.

Open the Cursor

You need to open the cursor before you fetch rows from it. You do this with the OPEN statement.

```
OPEN cursor_name;
```

Fetch the Data from the Cursor

The **FETCH** statement retrieves the data from the cursor and moves the cursor to the next line in the result set; it loads the data into variables.

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

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

Close Cursor

Finally, you would close the cursor after you have processed all the data, so that the resources that are allocated for it will be released.

```
CLOSE cursor_name;
```

Example 1: Complete Cursor Usage

```
DELIMITER //

CREATE PROCEDURE ProcessOrders()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE order_id INT;
    DECLARE order_amount DECIMAL(10, 2);

    -- Declare the cursor
    DECLARE order_cursor CURSOR FOR
       SELECT id, amount FROM orders WHERE status = 'pending';
    -- Declare a handler for the NOT FOUND condition
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Open the cursor
    OPEN order_cursor;

    -- Loop through the rows
    read_loop: LOOP
        FETCH order_cursor INTO order_id, order_amount;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Process each row
        UPDATE orders SET status = 'processed' WHERE id = order_id;

    END LOOP;

    -- Close the cursor
    CLOSE order_cursor;
END //

DELIMITER ;
```

Explanation:

- **DECLARE CURSOR** declares a cursor with name **order_cursor** for the following query which selects all pending orders.
- **OPEN** initializes the cursor.
- **FETCH** fetches every row and loads them into the order_amount and order_id accordingly.
- What it does is loop through all the rows processing them and updating their status.
- Finally, **CLOSE** releases the cursor resources.

Example 2: Cursor with Conditional Logic

The example below shows using a cursor to process all rows subject to conditions, in order to update them.

```
DELIMITER //

CREATE PROCEDURE UpdateOrderStatus()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE order_id INT;
    DECLARE order_amount DECIMAL(10, 2);

    -- Declare the cursor
    DECLARE order_cursor CURSOR FOR
       SELECT id, amount FROM orders WHERE status = 'pending';

    -- Declare a handler for the NOT FOUND condition
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Open the cursor
    OPEN order_cursor;

    -- Loop through the rows
    read_loop: LOOP
        FETCH order_cursor INTO order_id, order_amount;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Example logic: If order amount is greater than 1000,
update status
        IF order_amount > 1000 THEN
            UPDATE orders SET status = 'high_value' WHERE id =
order_id;
        ELSE
            UPDATE orders SET status = 'processed' WHERE id =
order_id;
        END IF;

    END LOOP;

    -- Close the cursor
    CLOSE order_cursor;
END //

DELIMITER ;
```

Explanation:

- This example uses cursors from **MySQL** to manipulate customer information.
- It creates a stored procedure called **SimpleCursorExample** that demonstrates cursor operations over the data of customers.
- It declares a **cursor** for **fetching customer** IDs and names from the customer table.
- Open the cursor and **fetch rows** one at a time; log the customer information into the **customer_log** table.
- In this example, it shows how to iterate over the result set, processing—for example, logging or data manipulation—on a row-by-row basis.

Cursor with Error Handling

This example shows how to use a cursor with error handling in place, so that any exceptions thrown off of the operations on the cursor are caught.

```
DELIMITER //

CREATE PROCEDURE ProcessSales()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE sale_id INT;
    DECLARE sale_total DECIMAL(10, 2);
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
        -- Handle SQL exceptions
        ROLLBACK;
        SELECT 'An error occurred. Transaction rolled back.';
    END;

    -- Declare the cursor
    DECLARE sales_cursor CURSOR FOR
       SELECT id, total FROM sales WHERE processed = FALSE;

    -- Declare a handler for the NOT FOUND condition
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    -- Start a transaction
    START TRANSACTION;

    -- Open the cursor
    OPEN sales_cursor;

    -- Loop through the rows
    read_loop: LOOP
        FETCH sales_cursor INTO sale_id, sale_total;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Example logic: Process each sale and handle errors
        BEGIN
            -- Some processing logic (e.g., update sale status)
            UPDATE sales SET processed = TRUE WHERE id = sale_id;
        END;

    END LOOP;

    -- Close the cursor
    CLOSE sales_cursor;

    -- Commit the transaction
    COMMIT;
END //

DELIMITER ;
```

Explanation:

- The **ProcessSales** procedure shows the use of cursors with error handling. It processes the records of sales using a cursor, together with a **CONTINUE HANDLER** for **SQL** exceptions.
- In case of failure in the process, the procedure rolls back the **transaction** to maintain data integrity and outputs an **error message**.
- This example shows how to handle exceptions and make a cursor-based operation quite reliable for transaction handling.

Conclusion

Overall, MySQL cursors provide a way to handle and process data one row at a time, which is useful for complex tasks that require detailed operations. They allow precise control over data manipulation within stored procedures. However, while powerful, cursors can impact performance compared to set-based operations, so they should be used carefully to balance detail with efficiency.

FAQs on MySQL Cursors

What is a MySQL cursor?

One of the database objects used in MySQL is a cursor. It is a database object that retrieves the rows from a result set one at a time. Processing per row can be fully detailed within a stored procedure or function.

How many types of cursors are there in MySQL?

Read-Only Cursor: This cursor is used to fetch data without changing it.

Scrollable Cursor: This cursor allows moving backward and forward through the result set. Note: MySQL does not support this directly. It is usually s