**Cursor :**

A cursor in SQL is a database object that allows you to retrieve and manipulate data row by row, rather than fetching the entire result set at once. Cursors are typically used within stored procedures or scripts to process individual rows of data sequentially. They provide a mechanism for traversing through the result set and performing operations on each row, such as updating values, performing calculations, or applying business logic. Cursors offer finer control over data manipulation, especially in scenarios where you need to process rows individually or perform complex row-level operations.  
  
**Triggers vs Cursor :**

In SQL, triggers and stored procedures are both database objects used to execute a sequence of SQL statements. However, they serve different purposes and have distinct characteristics:

1. **\*Purpose\*:**

- **\*Stored Procedure\***: A stored procedure is a named set of SQL statements that can be invoked multiple times. It is primarily used for encapsulating business logic or complex operations that need to be executed repeatedly.

- **\*Trigger\***: A trigger is a special type of stored procedure that automatically executes in response to certain events, such as INSERT, UPDATE, or DELETE operations on a table. Triggers are often used for enforcing data integrity rules, auditing changes, or implementing business logic that should be triggered by specific database actions.

2**. \*Execution\*:**

- **\*Stored Procedure\*:** Stored procedures are explicitly called by users or applications when needed.

- **\*Trigger\***: Triggers are automatically invoked by the database engine when the associated event occurs on the table.

3. **\*Timing\*:**

- \***Stored Procedure**\*: Stored procedures are executed explicitly at the time of invocation.

- \***Trigger**\*: Triggers are executed implicitly in response to predefined events, such as data modification operations on the table.

4. \***Scope\***:

- \***Stored Procedure**\*: Stored procedures can perform a wide range of operations on one or more tables and can include conditional logic, loops, and error handling.

- \***Trigger**\*: Triggers are associated with specific tables and events, and their scope is limited to the context of the triggering event and the affected row(s) in the table.

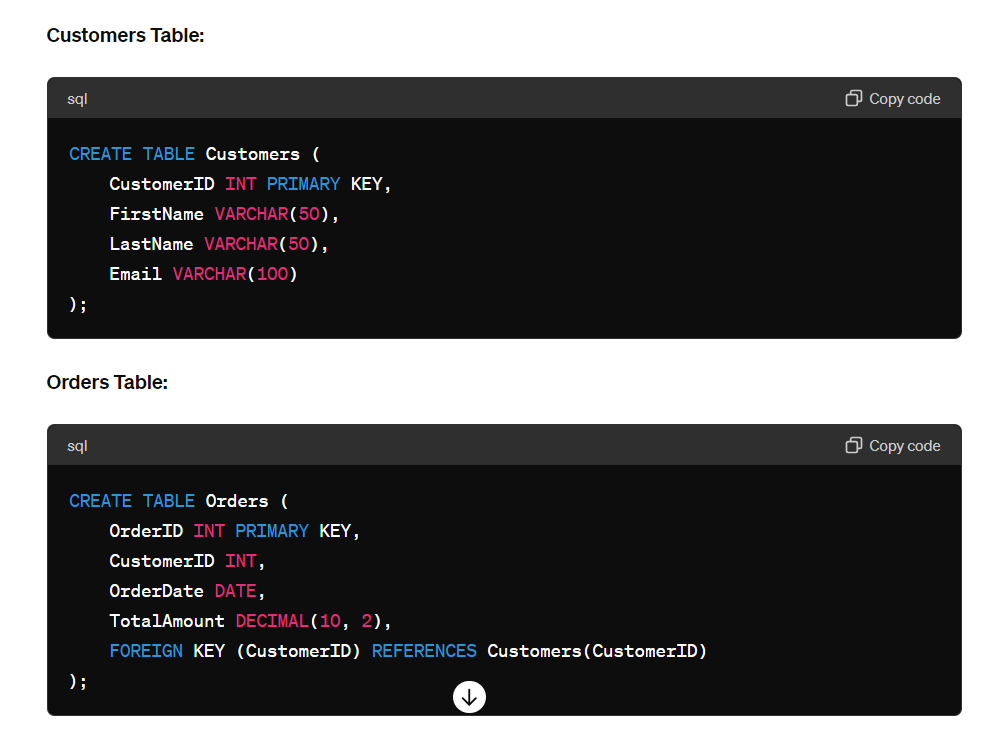
5**. \*Transaction Control\***:

- **\*Stored Procedure\*:** Stored procedures can include transaction control statements (e.g., BEGIN TRANSACTION, COMMIT, ROLLBACK) to manage transactions within the procedure.

- **\*Trigger\*:** Triggers are implicitly part of the transaction that caused them to fire and cannot control the transaction explicitly.

In summary, stored procedures are reusable units of SQL logic primarily used for encapsulating business logic, while triggers are special stored procedures that automatically execute in response to predefined database events to enforce data integrity rules or implement specific behaviour.

**Stored Procedure :**

****

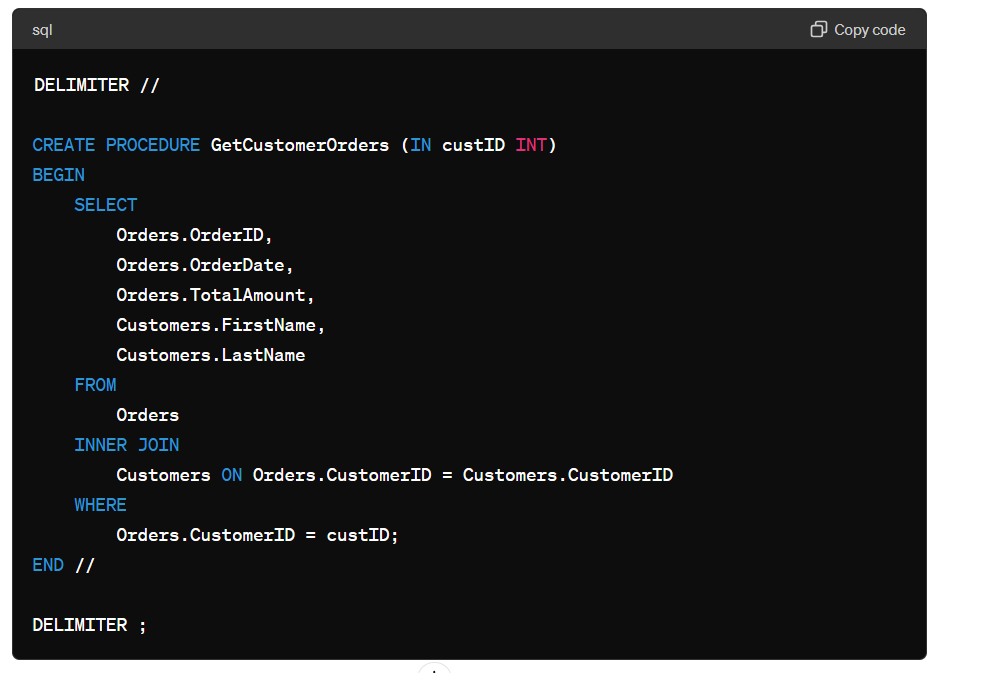
**Step 2: Insert Sample Data**

Let's insert some sample data into the **Customers** and **Orders** tables:

****

### Step 3: Create the Stored Procedure

We will create a stored procedure named **GetCustomerOrders** that takes a customer ID as an input parameter and returns the orders for that customer.

****

**Explanation**

* **DELIMITER //**: Changes the statement delimiter temporarily to **//** to allow the procedure body to contain **;**.
* **CREATE PROCEDURE GetCustomerOrders (IN custID INT)**: Defines a stored procedure named **GetCustomerOrders** with an input parameter **custID** of type INT.
* **SELECT ... FROM Orders INNER JOIN Customers ... WHERE Orders.CustomerID = custID**: The SQL query that joins the **Orders** and **Customers** tables and filters the results based on the provided customer ID.

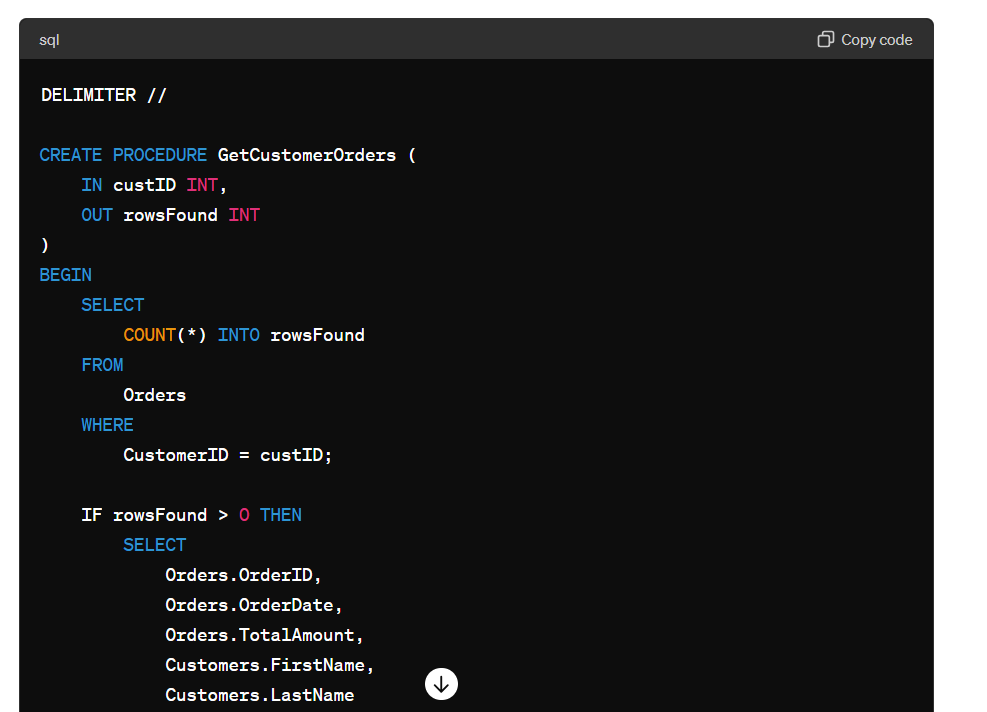
**Step 4: Call the Stored Procedure**

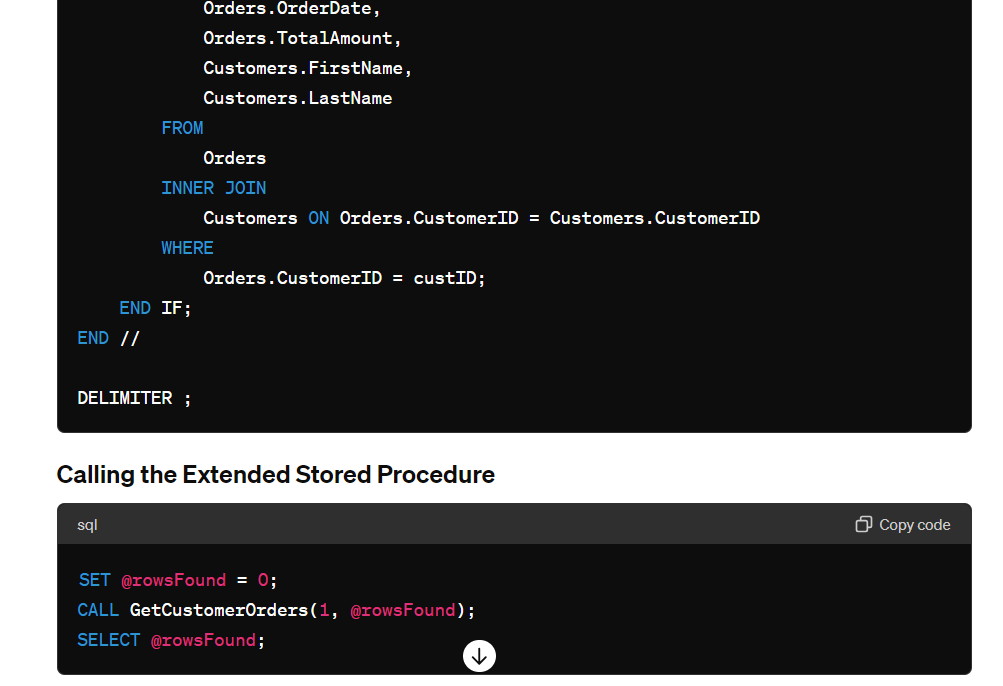
To use the stored procedure, you call it and pass the necessary parameter:

****

### Step 5: Handling No Results

To handle scenarios where no orders exist for a given customer, you can extend the procedure with additional logic, such as using an OUT parameter or signaling a message. Here’s a simple example with an OUT parameter to indicate if any rows were found:

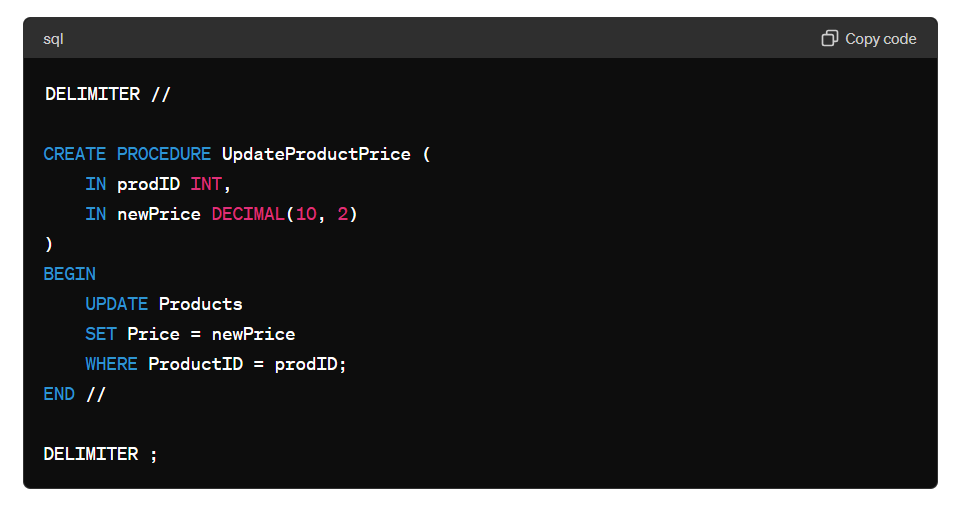
****

****

### Summary

This example demonstrates how to create, use, and extend a stored procedure in SQL to encapsulate a query that retrieves orders for a specific customer. Stored procedures help in reusing code, improving performance, and ensuring security in database operations.

**Update**

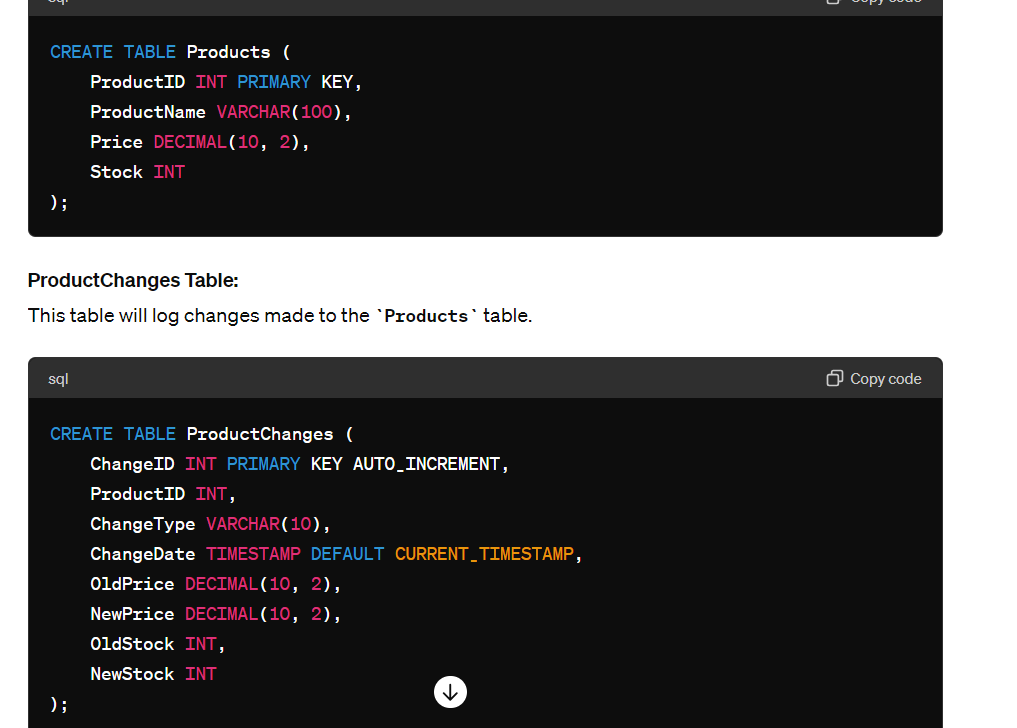
****

**Trigger :**

Certainly! Here’s an example of a simple SQL trigger that handles **INSERT**, **UPDATE**, and **DELETE** operations on a table. We will create a trigger on a hypothetical **Products** table to log changes to another table called **ProductChanges**.

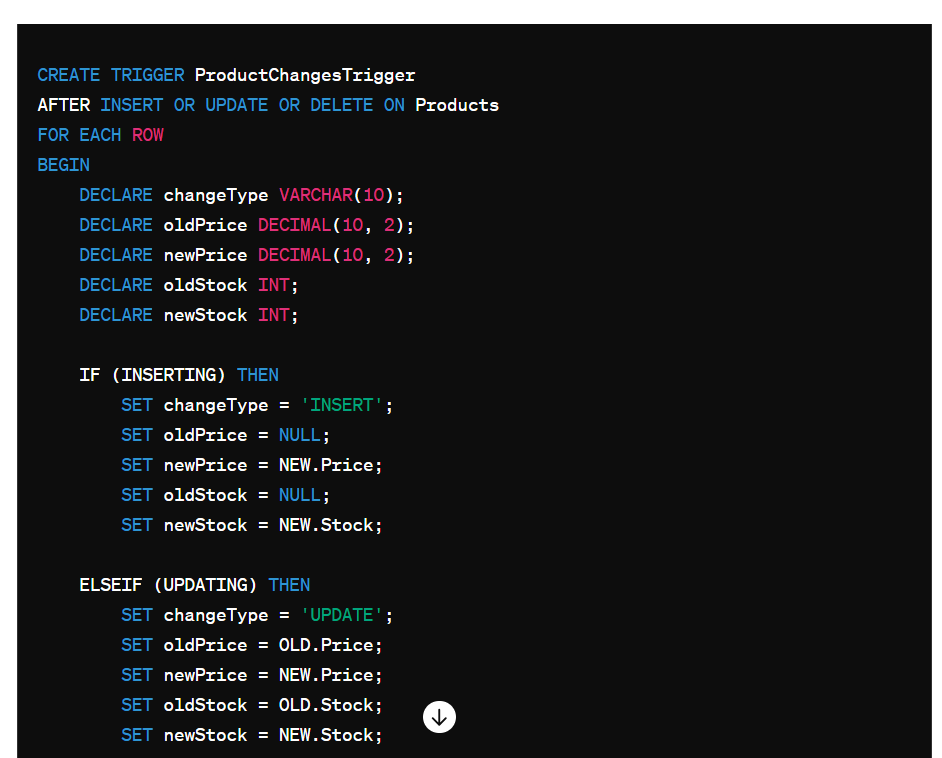
### Step 1: Define the Tables

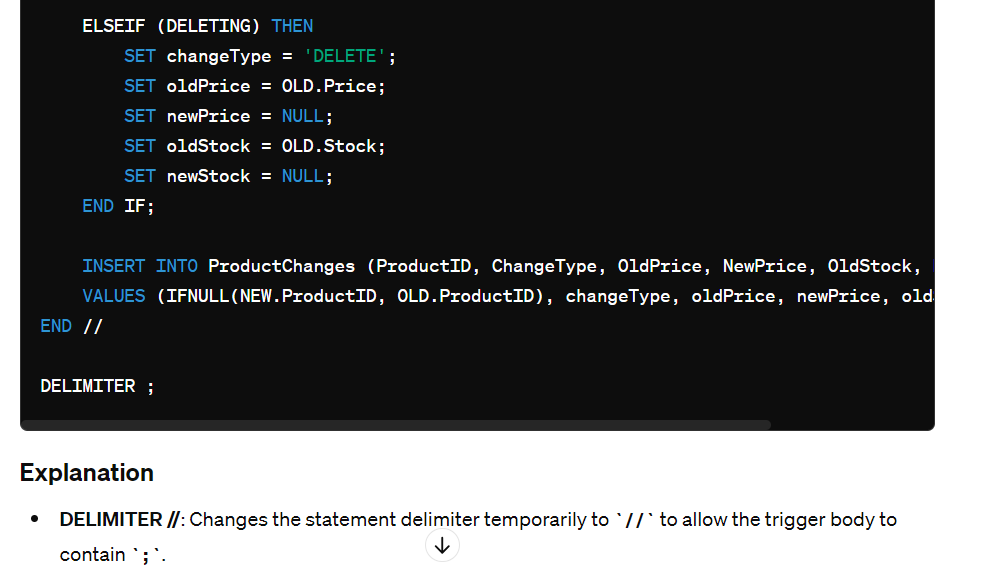
**Products Table:**

****

**Step 2: Create the Trigger**

We will create a single trigger that fires on **INSERT**, **UPDATE**, and **DELETE** operations on the **Products** table. This trigger will insert a record into the **ProductChanges** table describing the change.

****

****

Certainly! Here’s an example of a simple SQL trigger that handles **INSERT**, **UPDATE**, and **DELETE** operations on a table. We will create a trigger on a hypothetical **Products** table to log changes to another table called **ProductChanges**.

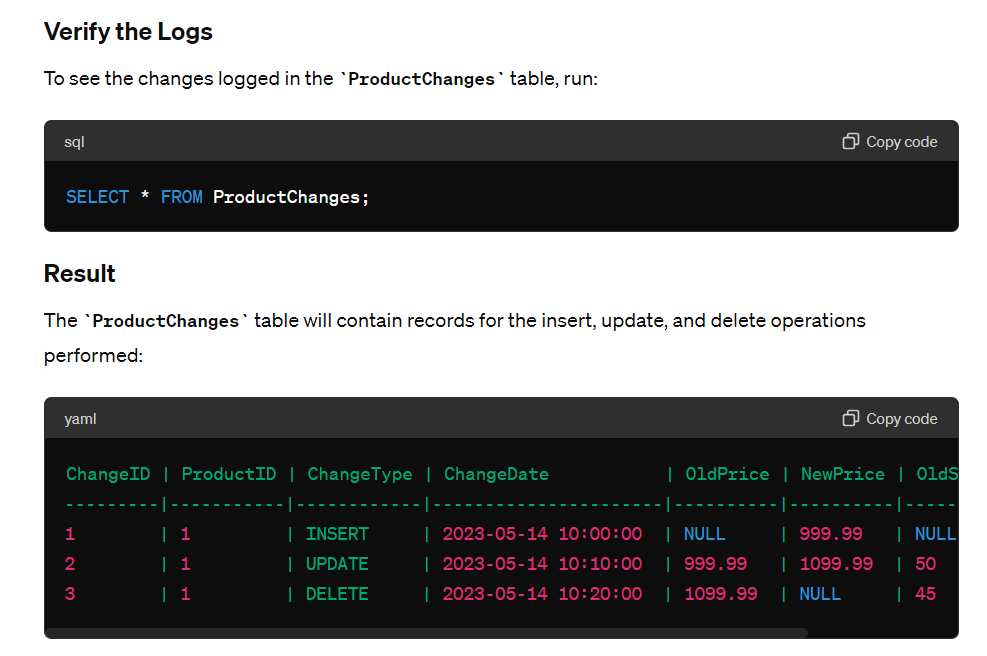
**Explanation**

* **DELIMITER //**: Changes the statement delimiter temporarily to **//** to allow the trigger body to contain **;**.
* **CREATE TRIGGER ProductChangesTrigger AFTER INSERT OR UPDATE OR DELETE ON Products FOR EACH ROW**: Defines a trigger named **ProductChangesTrigger** that fires after insert, update, or delete operations on the **Products** table.
* **DECLARE**: Declares variables to store change details.
* **IF (INSERTING) THEN ... ELSEIF (UPDATING) THEN ... ELSEIF (DELETING) THEN ... END IF**: Checks the type of operation being performed and sets the appropriate variables.
* **INSERT INTO ProductChanges**: Inserts a new record into the **ProductChanges** table with details of the change.

**Step 3: Test the Trigger**

Insert a new product:

****

****

### Summary

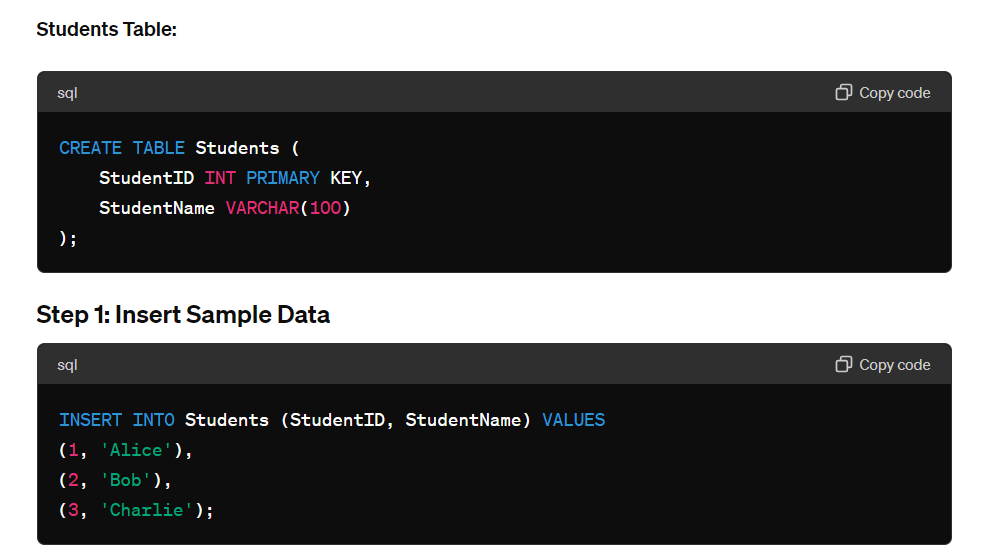
This example demonstrates how to create a single trigger that handles insert, update, and delete operations on a table, and logs these changes into a separate table. This approach can help maintain an audit trail of changes made to critical data in your database.

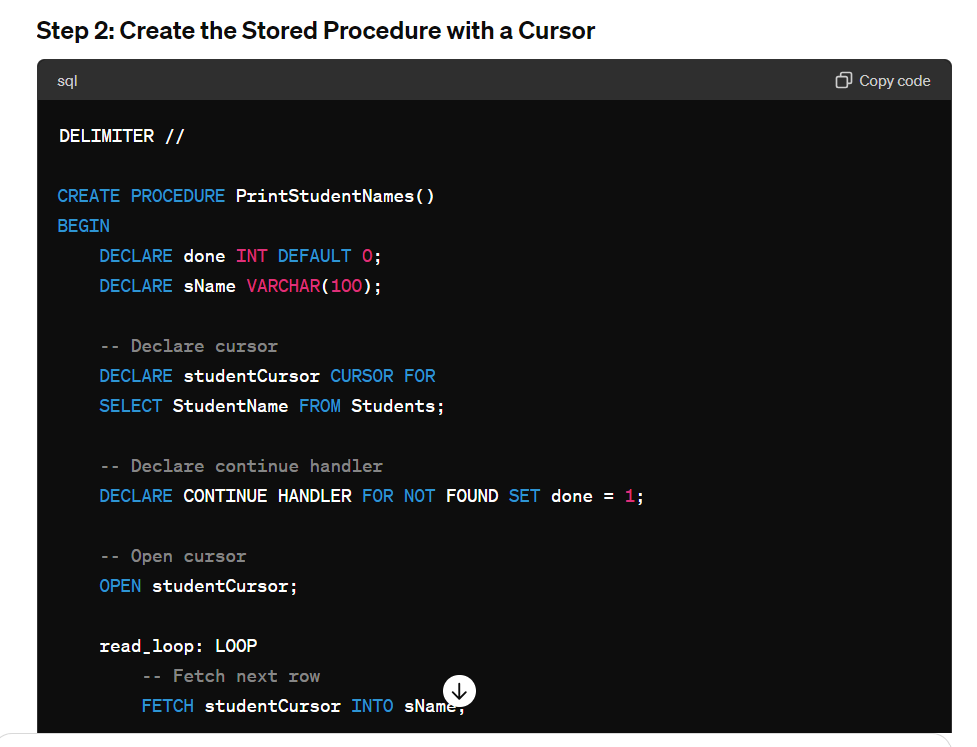
**Cursor :**

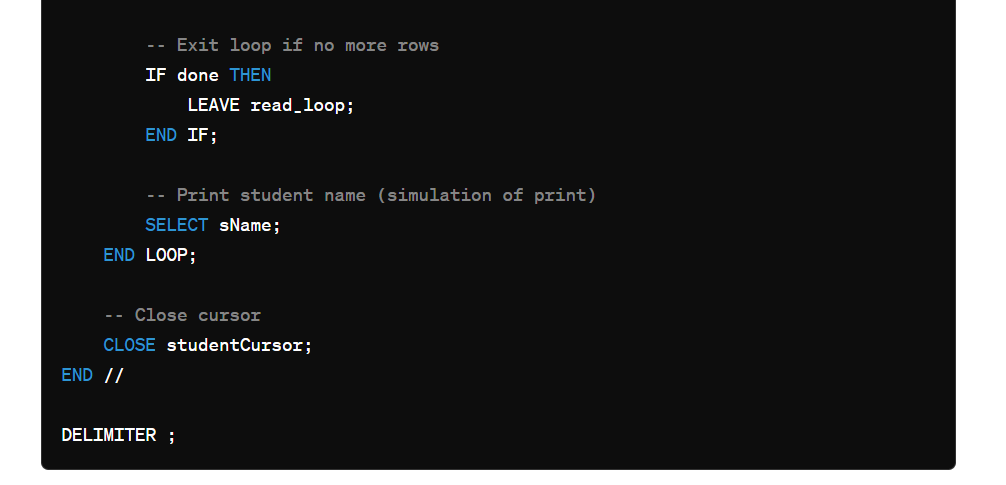
Sure! Here’s a simple example to illustrate the use of cursors in SQL. A cursor allows you to iterate through a result set row by row.

### Scenario

We have a **Students** table, and we want to create a stored procedure that uses a cursor to print the names of all students.

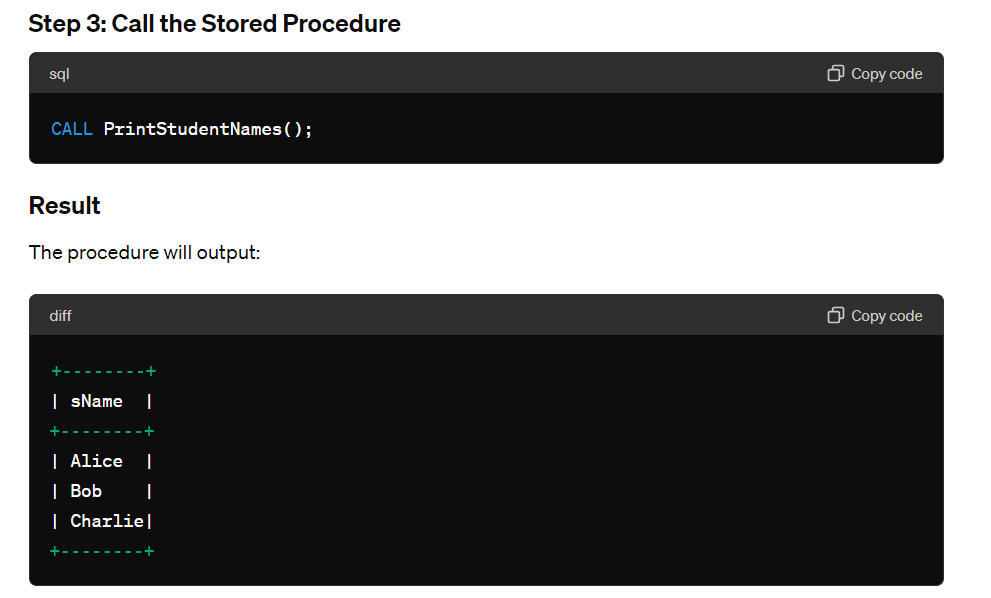
****

****

****

**Explanation**

1. **Declare Variables**: **done** is used to indicate if the cursor has fetched all rows. **sName** stores the student name from each row.
2. **Declare Cursor**: **studentCursor** is defined to select student names from the **Students** table.
3. **Declare Handler**: A handler sets **done** to 1 when there are no more rows to fetch.
4. **Open Cursor**: Initializes the cursor to start fetching rows.
5. **Loop through Rows**: Fetches each row into **sName** and exits the loop when no more rows are found.
6. **Print/Simulate Print**: In a real application, you might process the data. Here, we use **SELECT sName** to simulate printing.
7. **Close Cursor**: Closes the cursor to free resources.

****

### Summary

This example demonstrates how to use a cursor to iterate through a result set in SQL, processing each row individually. Cursors are useful for row-by-row operations when set-based operations are not feasible.