Triggers are another powerful feature in SQL that allow you to automatically perform a specified action in the database when certain events occur. Triggers can be used to enforce business rules, validate input, maintain audit trails, and more.

keyboard\_arrow\_down

**What is a Trigger?**

A trigger is a database object that is automatically executed or fired when certain events occur. These events can be:

* INSERT: When a new row is added to a table.
* UPDATE: When a row in a table is modified.
* DELETE: When a row is removed from a table.

Creating Triggers

To demonstrate the use of triggers, let's continue with our previous example of the employees and departments tables. We'll create a trigger to maintain an audit log of salary changes for employees.

Step-by-Step Guide to Creating a Trigger

1. Create the Audit Table

First, we need an audit table to store the salary change logs.

CREATE TABLE salary\_audit (  
    audit\_id INT AUTO\_INCREMENT PRIMARY KEY,  
    employee\_id INT,  
    old\_salary DECIMAL(10, 2),  
    new\_salary DECIMAL(10, 2),  
    change\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

2. Create the Trigger

Next, we create a trigger that logs changes to the salary field in the employees table.

DELIMITER $$  
  
CREATE TRIGGER before\_salary\_update  
BEFORE UPDATE ON employees  
FOR EACH ROW  
BEGIN  
    IF OLD.salary <> NEW.salary THEN  
        INSERT INTO salary\_audit (employee\_id, old\_salary, new\_salary)  
        VALUES (OLD.employee\_id, OLD.salary, NEW.salary);  
    END IF;  
END$$  
  
DELIMITER ;

Explanation of the Trigger

* **DELIMITER $$**: Changes the statement delimiter to $$ to allow the use of semicolons within the trigger definition.
* **CREATE TRIGGER before\_salary\_update**: Defines a new trigger named before\_salary\_update.
* **BEFORE UPDATE ON employees**: Specifies that the trigger should fire before an update on the employees table.
* **FOR EACH ROW**: Indicates that the trigger will execute once for each row affected by the update.
* **BEGIN ... END**: Marks the beginning and end of the trigger’s body.
* **IF OLD.salary <> NEW.salary THEN ... END IF**: Checks if the salary field is actually changing. If it is, inserts a record into the salary\_audit table with the old and new salary values.

Using the Trigger

After creating the trigger, any update to the salary field in the employees table will automatically create an entry in the salary\_audit table.

Example Update

Let's update an employee's salary to see the trigger in action:

UPDATE employees  
SET salary = 65000.00  
WHERE employee\_id = 1;

Verifying the Trigger

Check the salary\_audit table to verify that the change was logged:

SELECT \* FROM salary\_audit;

Full SQL Script for Trigger

Here's the complete script including creating the database, tables, inserting data, and creating the trigger:

-- Create Database  
CREATE DATABASE IF NOT EXISTS mydatabase;  
USE mydatabase;  
  
-- Create `employees` Table  
CREATE TABLE employees (  
    employee\_id INT PRIMARY KEY,  
    first\_name VARCHAR(50),  
    last\_name VARCHAR(50),  
    department\_id INT,  
    salary DECIMAL(10, 2)  
);  
  
-- Create `departments` Table  
CREATE TABLE departments (  
    department\_id INT PRIMARY KEY,  
    department\_name VARCHAR(50)  
);  
  
-- Create `projects` Table  
CREATE TABLE projects (  
    project\_id INT PRIMARY KEY,  
    project\_name VARCHAR(50),  
    manager\_id INT  
);  
  
-- Insert Data into `employees` Table  
INSERT INTO employees (employee\_id, first\_name, last\_name, department\_id, salary) VALUES  
(1, 'John', 'Doe', 101, 60000.00),  
(2, 'Jane', 'Smith', 102, 75000.00),  
(3, 'Alice', 'Johnson', 101, 50000.00),  
(4, 'Bob', 'Lee', 103, 55000.00),  
(5, 'Charlie', 'Brown', 102, 70000.00);  
  
-- Insert Data into `departments` Table  
INSERT INTO departments (department\_id, department\_name) VALUES  
(101, 'HR'),  
(102, 'IT'),  
(103, 'Finance');  
  
-- Insert Data into `projects` Table  
INSERT INTO projects (project\_id, project\_name, manager\_id) VALUES  
(1001, 'Alpha', 2),  
(1002, 'Beta', 3),  
(1003, 'Gamma', 5);  
  
-- Create `salary\_audit` Table  
CREATE TABLE salary\_audit (  
    audit\_id INT AUTO\_INCREMENT PRIMARY KEY,  
    employee\_id INT,  
    old\_salary DECIMAL(10, 2),  
    new\_salary DECIMAL(10, 2),  
    change\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Create Trigger  
DELIMITER $$  
  
CREATE TRIGGER before\_salary\_update  
BEFORE UPDATE ON employees  
FOR EACH ROW  
BEGIN  
    IF OLD.salary <> NEW.salary THEN  
        INSERT INTO salary\_audit (employee\_id, old\_salary, new\_salary)  
        VALUES (OLD.employee\_id, OLD.salary, NEW.salary);  
    END IF;  
END$$  
  
DELIMITER ;  
  
-- Example Update to Test the Trigger  
UPDATE employees  
SET salary = 65000.00  
WHERE employee\_id = 1;  
  
-- Verify the Trigger  
SELECT \* FROM salary\_audit;

Conclusion

Triggers are essential for enforcing business rules and maintaining data integrity automatically within the database. They help in automating actions like auditing, logging, and cascading changes across related tables. By using triggers, you can ensure that critical business logic is consistently applied and that your database remains reliable and secure.

**Stored procedures and triggers difference**

**Stored procedures and triggers** are both fundamental components in SQL databases, but they serve different purposes and are used in different contexts. Below, I'll outline the key differences between stored procedures and triggers, and explain their use cases and functionalities:

keyboard\_arrow\_down

### Key Differences

#### 1. Invocation

* **Stored Procedures**:
  + **Explicit Invocation**: Stored procedures are explicitly called by a user or an application.
  + **Syntax Example**: CALL procedure\_name();
  + **Use Cases**: Performing batch processing, complex calculations, data validation, or encapsulating business logic that can be reused across different parts of an application.
* **Triggers**:
  + **Implicit Invocation**: Triggers are automatically executed in response to specific events on a particular table, such as INSERT, UPDATE, or DELETE.
  + **Syntax Example**: Automatically fires when an event occurs, e.g., BEFORE INSERT ON table\_name.
  + **Use Cases**: Enforcing data integrity, maintaining audit trails, and automating system responses to data changes.

#### 2. Scope

* **Stored Procedures**:
  + **General Scope**: Can operate on multiple tables and can be called with parameters to perform operations based on input values.
  + **Example**: A procedure to calculate and update employee bonuses across multiple tables based on performance metrics.
* **Triggers**:
  + **Table-Specific Scope**: Attached to a specific table and operates within the context of the triggering event.
  + **Example**: A trigger that logs changes to a specific column in an audit table whenever an update occurs.

#### 3. Complexity and Control Flow

* **Stored Procedures**:
  + **More Complex Logic**: Can include loops, conditional statements, and error handling. They can also call other procedures and functions.
  + **Example**: A procedure to generate monthly financial reports, which involves multiple steps and calculations.
* **Triggers**:
  + **Simpler Logic**: Designed to handle event-driven tasks and usually contain simpler logic compared to procedures.
  + **Example**: A trigger to ensure that a new record's timestamp is automatically set to the current time on insertion.

#### 4. Use Cases and Examples

##### **Stored Procedures Use Case**

**Example: Stored Procedure for Monthly Payroll Calculation**

DELIMITER $$  
  
CREATE PROCEDURE CalculateMonthlyPayroll()  
BEGIN  
    DECLARE total\_salary DECIMAL(10,2);  
  
    -- Calculate total salary for each department  
    SELECT SUM(salary) INTO total\_salary  
    FROM employees  
    WHERE department\_id = 101;  
  
    -- Insert the result into payroll\_summary table  
    INSERT INTO payroll\_summary (department\_id, total\_salary, payroll\_date)  
    VALUES (101, total\_salary, NOW());  
  
    -- Repeat for other departments as necessary  
    -- ... additional logic ...  
  
END$$  
  
DELIMITER ;

**Invocation**:

CALL CalculateMonthlyPayroll();

##### **Trigger Use Case**

**Example: Trigger for Auditing Salary Changes**

DELIMITER $$  
  
CREATE TRIGGER before\_salary\_update  
BEFORE UPDATE ON employees  
FOR EACH ROW  
BEGIN  
    IF OLD.salary <> NEW.salary THEN  
        INSERT INTO salary\_audit (employee\_id, old\_salary, new\_salary, change\_date)  
        VALUES (OLD.employee\_id, OLD.salary, NEW.salary, NOW());  
    END IF;  
END$$  
  
DELIMITER ;

**Automatic Invocation**:

UPDATE employees  
SET salary = 65000.00  
WHERE employee\_id = 1;

This update will automatically trigger the before\_salary\_update trigger if the salary changes.

### Summary

* **Stored Procedures**:
  + **Explicitly invoked** by users or applications.
  + **Used for complex operations** involving multiple steps, conditional logic, and calculations.
  + **Scope is broader** and can operate on multiple tables and be called with parameters.
* **Triggers**:
  + **Automatically executed** in response to specific table events.
  + **Used for maintaining data integrity** and automating simple, repetitive tasks tied to table events.
  + **Scope is narrower**, tied to specific tables and events with simpler logic.

Both stored procedures and triggers are essential tools for managing and manipulating data within a database. Understanding when and how to use them effectively can greatly enhance the performance, maintainability, and reliability of your database applications.