Sure, let's create an SQL procedure using the provided dataset. We'll write a stored procedure that performs a specific operation. For this example, let's create a procedure that retrieves the highest salary for each department and the employee details who earns that salary.

Step-by-Step Guide to Creating an SQL Procedure

1. Creating the Procedure

First, we will define the procedure in SQL. Here’s an example:

DELIMITER $$  
  
CREATE PROCEDURE GetHighestSalaryPerDepartment()  
BEGIN  
    SELECT e.department\_id, d.department\_name, e.employee\_id, e.first\_name, e.last\_name, e.salary  
    FROM employees e  
    JOIN departments d ON e.department\_id = d.department\_id  
    WHERE e.salary = (SELECT MAX(salary) FROM employees WHERE department\_id = e.department\_id);  
END$$  
  
DELIMITER ;

2. Explanation of the Procedure

* **DELIMITER $$**: Changes the statement delimiter to $$ to allow for the use of semicolons within the procedure.
* **CREATE PROCEDURE GetHighestSalaryPerDepartment()**: Defines a new stored procedure named GetHighestSalaryPerDepartment.
* **BEGIN ... END**: Marks the beginning and end of the procedure’s body.
* **SELECT ... FROM ... JOIN ... WHERE ...**: Retrieves the highest salary for each department and the details of the employee who earns that salary by using a correlated subquery.

3. Using the Procedure

Once the procedure is created, you can call it to execute the logic defined within it. Here’s how you can call the procedure:

CALL GetHighestSalaryPerDepartment();

Full SQL Script for the Procedure

Here is the complete script including creating the database, tables, inserting data, and the stored procedure:

-- 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 Procedure  
DELIMITER $$  
  
CREATE PROCEDURE GetHighestSalaryPerDepartment()  
BEGIN  
    SELECT e.department\_id, d.department\_name, e.employee\_id, e.first\_name, e.last\_name, e.salary  
    FROM employees e  
    JOIN departments d ON e.department\_id = d.department\_id  
    WHERE e.salary = (SELECT MAX(salary) FROM employees WHERE department\_id = e.department\_id);  
END$$  
  
DELIMITER ;  
  
-- Call Procedure  
 CALL GetHighestSalaryPerDepartment();

Explanation of the Procedure Execution

* **CALL GetHighestSalaryPerDepartment();**: This statement executes the stored procedure. The result will display the highest salary for each department and the employee details who earns that salary.

By following these steps, you can create and use stored procedures in MySQL to encapsulate complex queries and operations, making your SQL code more modular and reusable.

**why use procedure?**

Stored procedures are highly useful in real-world projects for several reasons. They encapsulate complex business logic, improve performance, enhance security, and provide maintainability. Here are some key use cases and benefits of using stored procedures in real-world applications:

1. Encapsulation of Business Logic

Stored procedures allow you to encapsulate business logic in a centralized, reusable way. This means that complex calculations, data manipulations, and validation rules can be stored on the database server rather than being scattered across different parts of an application.

**Example:** A payroll system might use stored procedures to calculate employee salaries, taxes, bonuses, and deductions. By encapsulating this logic in stored procedures, the payroll calculations are consistent and easily maintainable.

2. Performance Optimization

Stored procedures can enhance performance by reducing the amount of data transferred between the application and the database. Since the operations are performed on the server side, the network overhead is minimized.

**Example:** A reporting system that generates complex reports can benefit from stored procedures. By performing data aggregation and filtering on the server side, the application can retrieve only the final results, reducing data transfer and speeding up the report generation process.

3. Security

Stored procedures provide an additional layer of security. By granting permissions to execute specific procedures rather than direct access to tables, you can control how data is accessed and modified. This helps in protecting sensitive data.

**Example:** In a financial application, users might have permission to execute stored procedures for account transactions, but not to directly access the account balances table. This ensures that all data modifications go through controlled, audited procedures.

4. Maintainability

Stored procedures help in maintaining the code by centralizing data access logic. Changes to business rules or data structures can be made in one place without requiring updates to application code.

**Example:** If the tax calculation rules change, updating the relevant stored procedure ensures that all parts of the application using this procedure automatically adopt the new rules, without the need for widespread code changes.

5. Reusability

Stored procedures promote reusability. Once a procedure is written and tested, it can be used by different applications or parts of an application without duplication.

**Example:** A stored procedure that checks user credentials can be reused across multiple services and applications within an organization, ensuring consistent authentication logic.

Real-World Use Cases

1. E-commerce Systems

* **Order Processing:** Stored procedures can handle order validation, inventory checks, and payment processing in a transactional manner.
* **Reporting:** Generating sales reports, customer purchase history, and inventory status can be efficiently handled by stored procedures.

2. Banking and Financial Services

* **Transaction Management:** Procedures can ensure that complex transactions, such as fund transfers, are executed correctly and consistently.
* **Compliance Reporting:** Generating regulatory compliance reports with accurate and aggregated data from multiple tables.

3. Healthcare Systems

* **Patient Data Management:** Procedures can manage patient records, ensuring data consistency and integrity across various operations.
* **Billing:** Calculating patient bills, applying insurance policies, and generating invoices can be efficiently done using stored procedures.

4. Content Management Systems

* **Content Publishing:** Handling the workflow of content approval, publication, and archiving.
* **User Management:** Procedures can manage user roles, permissions, and activity logging to ensure secure access and modifications.

Example Scenario: E-commerce Order Processing

**Stored Procedure for Placing an Order:**

DELIMITER $$  
  
CREATE PROCEDURE PlaceOrder(  
    IN customerId INT,  
    IN productId INT,  
    IN quantity INT,  
    OUT orderId INT  
)  
BEGIN  
    DECLARE stock INT;  
    DECLARE price DECIMAL(10, 2);  
  
    -- Check stock availability  
    SELECT product\_stock INTO stock  
    FROM products  
    WHERE product\_id = productId;  
  
    IF stock < quantity THEN  
        SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient stock';  
    ELSE  
        -- Deduct stock  
        UPDATE products  
        SET product\_stock = product\_stock - quantity  
        WHERE product\_id = productId;  
  
        -- Get product price  
        SELECT product\_price INTO price  
        FROM products  
        WHERE product\_id = productId;  
  
        -- Create order  
        INSERT INTO orders (customer\_id, order\_date, total\_amount)  
        VALUES (customerId, NOW(), price \* quantity);  
  
        SET orderId = LAST\_INSERT\_ID();  
  
        -- Create order details  
        INSERT INTO order\_details (order\_id, product\_id, quantity, price)  
        VALUES (orderId, productId, quantity, price);  
    END IF;  
END$$  
  
DELIMITER ;

Explanation:

1. **Stock Check:** The procedure checks if the requested quantity is available in stock.
2. **Stock Deduction:** If stock is sufficient, it deducts the quantity from the product's stock.
3. **Order Creation:** Inserts a new order record.
4. **Order Details:** Adds details of the order, including the product and quantity.

Conclusion

Stored procedures are an essential tool in database management and application development. They help in centralizing and encapsulating business logic, improving performance, enhancing security, and ensuring maintainability. By leveraging stored procedures, organizations can build robust, scalable, and maintainable systems.

**Difference between procedure and function**

Stored procedures and functions are both reusable routines in SQL that encapsulate SQL code and business logic, but they have key differences in terms of usage, return values, and invocation. Here’s a detailed comparison to help understand their differences:

keyboard\_arrow\_down

### Key Differences

#### 1. Return Type

* **Stored Procedures**:
  + **Return Type**: Procedures do not return a value directly. They perform actions such as inserting, updating, or deleting records, and can return multiple values using output parameters.
  + **Usage**: Typically used for operations that modify the state of the database (e.g., DML operations like INSERT, UPDATE, DELETE).
  + **Example**: A procedure to update employee salaries based on their performance metrics.
* **Functions**:
  + **Return Type**: Functions must return a single value (scalar, table, or row). They are used to compute and return a value.
  + **Usage**: Generally used for computations and returning a value. They can be called within SQL statements.
  + **Example**: A function to calculate and return the tax amount for a given salary.

#### 2. Invocation

* **Stored Procedures**:
  + **Invocation**: Procedures are explicitly called using the CALL or EXEC statement.
  + **Syntax Example**: CALL procedure\_name(parameters);
  + **Example**: CALL UpdateEmployeeSalary(1, 50000);
* **Functions**:
  + **Invocation**: Functions can be called from within SQL statements (such as SELECT, WHERE, HAVING, ORDER BY) or from other SQL code.
  + **Syntax Example**: SELECT function\_name(parameters);
  + **Example**: SELECT CalculateTax(salary) FROM employees;

#### 3. Side Effects

* **Stored Procedures**:
  + **Side Effects**: Procedures can have side effects, such as modifying database tables.
  + **Example**: A procedure that updates multiple tables to reflect a new status of an order.
* **Functions**:
  + **Side Effects**: Functions are expected to have no side effects and should not modify database tables (especially in standard SQL). They are deterministic, meaning the same inputs should produce the same outputs.
  + **Example**: A function that calculates the distance between two points based on their coordinates.

#### 4. Usage Context

* **Stored Procedures**:
  + **Usage Context**: Used for tasks that require complex business logic, multiple operations, and transactional control. They can call other procedures and functions.
  + **Example**: A stored procedure for processing an entire batch of orders.
* **Functions**:
  + **Usage Context**: Used for tasks that return a computed value. They are used in queries to simplify and modularize code.
  + **Example**: A function for formatting phone numbers in a consistent manner.

#### 5. Syntax and Structure

* **Stored Procedures**:
  + **Syntax**:

DELIMITER $$  
  
CREATE PROCEDURE procedure\_name(IN param1 INT, OUT param2 VARCHAR(50))  
BEGIN  
    -- Procedure logic here  
    SELECT column INTO param2 FROM table WHERE id = param1;  
END$$  
  
DELIMITER ;

* **Functions**:
  + **Syntax**:

DELIMITER $$  
  
CREATE FUNCTION function\_name(param1 INT)  
RETURNS VARCHAR(50)  
BEGIN  
    DECLARE result VARCHAR(50);  
    -- Function logic here  
    SELECT column INTO result FROM table WHERE id = param1;  
    RETURN result;  
END$$  
  
DELIMITER ;

### Example Scenarios

#### Stored Procedure Example

**Creating a Procedure to Update Employee Salary**

DELIMITER $$  
  
CREATE PROCEDURE UpdateEmployeeSalary(  
    IN employeeId INT,  
    IN newSalary DECIMAL(10, 2)  
)  
BEGIN  
    UPDATE employees  
    SET salary = newSalary  
    WHERE employee\_id = employeeId;  
END$$  
  
DELIMITER ;  
  
-- Calling the procedure  
CALL UpdateEmployeeSalary(1, 65000.00);

#### Function Example

**Creating a Function to Calculate Tax**

DELIMITER $$  
  
CREATE FUNCTION CalculateTax(  
    salary DECIMAL(10, 2)  
)  
RETURNS DECIMAL(10, 2)  
BEGIN  
    DECLARE tax DECIMAL(10, 2);  
    SET tax = salary \* 0.20; -- Assuming a flat tax rate of 20%  
    RETURN tax;  
END$$  
  
DELIMITER ;  
  
-- Using the function in a SELECT statement  
SELECT employee\_id, salary, CalculateTax(salary) AS tax\_amount  
FROM employees;

### Summary

* **Stored Procedures**:
  + Perform actions such as modifying data.
  + Do not return a value directly (use output parameters if needed).
  + Called explicitly using CALL or EXEC.
  + Can contain complex logic and operations involving multiple steps.
* **Functions**:
  + Return a single value.
  + Used for computations and can be called within SQL statements.
  + Do not modify the database state (no side effects).
  + Called within queries or other SQL code.

Both stored procedures and functions are essential tools in SQL for encapsulating and reusing logic. Understanding when to use each can lead to more efficient, maintainable, and secure database applications.