**Table**

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department_id INT,
    salary DECIMAL(10,2),
    hire_date DATE
);
INSERT INTO employees (first_name, last_name, department_id, salary,
hire_date) VALUES
('Amit', 'Sharma', 1, 55000, '2020-01-15'),
('Priya', 'Patel', 2, 65000, '2019-03-22'),
('Raj', 'Verma', 1, 50000, '2018-07-11'),
('Sneha', 'Singh', 3, 75000, '2021-05-30'),
('Anil', 'Nair', 2, 70000, '2017-11-05');
```

## Basic Stored procedures

Stored procedures are reusable routines stored in a database, typically written in SQL. They allow you to encapsulate complex SQL logic into a single callable function, which can simplify code, improve security, and optimize performance. Here's an overview of how to create and use basic stored procedures:

### 1. Simple Stored Procedure Syntax

Here's a basic stored procedure in SQL that retrieves data from a table called employees.

```
CREATE PROCEDURE GetAllEmployees()
BEGIN
    SELECT * FROM employees;
END;
```

- **Usage**: To call the procedure, use:

  ```
  CALL GetAllEmployees();
  ```

### 2. Stored Procedure with Parameters

Parameters make stored procedures more flexible. Here's an example of a stored procedure that takes a parameter to filter results:

```
CREATE PROCEDURE GetEmployeeByDepartment(IN dept_id INT)
```

```
BEGIN
   SELECT * FROM employees WHERE department_id = dept_id;
END;
```

- **Usage**:

  ```
  CALL GetEmployeeByDepartment(3);
  ```

  - Here, dept_id is an IN parameter, meaning it's provided by the caller.

**3. Stored Procedure with Multiple Parameters**

We can define multiple parameters for more control.

```
CREATE PROCEDURE GetEmployeesBySalary(IN min_salary DECIMAL(10,2), IN
max_salary DECIMAL(10,2))
BEGIN
   SELECT * FROM employees WHERE salary BETWEEN min_salary AND
max_salary;
END;
```

- **Usage**:

  ```
  CALL GetEmployeesBySalary(50000, 100000);
  ```

**4. Using OUT Parameters**

The OUT parameter allows you to pass data back to the calling code.

```
CREATE PROCEDURE GetEmployeeCountByDepartment(IN dept_id INT, OUT
emp_count INT)
BEGIN
   SELECT COUNT(*) INTO emp_count FROM employees WHERE department_id
= dept_id;
END;
```

- **Usage**:

  ```
  CALL GetEmployeeCountByDepartment(3, @count);
  SELECT @count;  -- This will show the output parameter value
  ```

**5. Stored Procedure with Control Flow**

You can use conditional statements (IF, WHILE, CASE) within stored procedures for more complex logic.

```
CREATE PROCEDURE UpdateEmployeeSalary(IN emp_id INT, IN new_salary DECIMAL(10,2))
BEGIN
   DECLARE current_salary DECIMAL(10,2);

   -- Get the current salary
   SELECT salary INTO current_salary FROM employees WHERE employee_id = emp_id;

   -- Check if the new salary is higher
   IF new_salary > current_salary THEN
      UPDATE employees SET salary = new_salary WHERE employee_id = emp_id;
   END IF;
END;
```

- **Usage**:

  ```
  CALL UpdateEmployeeSalary(101, 75000);
  ```

**Key Points to Remember**

- **IN, OUT, INOUT** parameters: Control how data is passed in and out of the stored procedure.
- **Error Handling**: Some databases support TRY...CATCH blocks or error-handling mechanisms.
- **Transactions**: Stored procedures can include transactions (BEGIN TRANSACTION, COMMIT, ROLLBACK) for atomicity.

Stored procedures are excellent for encapsulating logic in the database, making it easier to maintain, secure, and optimize queries.