SELF JOIN: Joins a table to itself, often used to compare rows within the same table.

- **Self Join Example:**

  ```
  SELECT
     e1.employee_name AS Employee,
     e2.employee_name AS Manager
  FROM
     employees e1
  INNER JOIN
     employees e2 ON e1.manager_id = e2.employee_id;
  ```

- **Explanation:** Here, the employees table is joined with itself to match employees to their managers. e1 represents the employee, and e2 represents the manager. This self-referencing join allows for pairing each employee with their corresponding manager.
- **Example Data:**
  - Employee Table:

employee_id	employee_name	department_id	manager_id
1	Alice	101	NULL
2	Bob	102	1
3	Charlie	103	1
4	David	104	2
5	Eva	105	3

  - Employee INNER JOIN Employee:

employee_id	employee_name	department_id	manager_id	manager_name
2	Bob	102	1	Alice
3	Charlie	103	1	Alice

4	David	104	2	Bob
5	Eva	105	3	Charlie

- **Explanation of the Result:**
    - Alice has no manager (manager_id is NULL).
    - Bob and Charlie report to Alice (manager_id = 1), so their manager_name is Alice.
    - David reports to Bob (manager_id = 2), so his manager_name is Bob.
    - Eva reports to Charlie (manager_id = 3), so her manager_name is Charlie.

Subqueries

A query nested within another query returns results to the outer query.

- **Example:**

```
SELECT
  order_id,
  customer_id
FROM
  orders
WHERE
  customer_id IN (
    SELECT
      customer_id
    FROM
      customers
    WHERE
      customer_status = 'Active'
  );
```

SQL Join Questions (All Types)

1. INNER JOIN:
   Write a SQL query to list all employees and their corresponding department names. Only include employees who are assigned to a department.
2. LEFT JOIN:
   Write a SQL query to list all employees and their department names. Include employees who might not be assigned to any department.
3. RIGHT JOIN:
   Write a SQL query to list all department names and the number of employees assigned to each. Include departments that currently have no employees, showing a count of 0 for them.
4. FULL OUTER JOIN (Emulation for MySQL):
   Write a SQL query to show all employees and all departments, matching them where possible. Include employees without a department and departments without any employees.
5. CROSS JOIN:
   Write a SQL query to generate all possible combinations of product names and category names.
6. SELF JOIN:
   Write a SQL query to list each employee and, if they have one, their manager's name.

Answers:

1. **INNER JOIN Answer:**

```
SELECT
   e.employee_name,
   d.department_name
FROM
   employees e
INNER JOIN
   departments d ON e.department_id = d.department_id;
```

2. **LEFT JOIN Answer:**

```
SELECT
   e.employee_name,
   d.department_name
FROM
   employees e
```

```
LEFT JOIN
    departments d ON e.department_id = d.department_id;
```

3. **RIGHT JOIN Answer:**

```
SELECT
    d.department_name,
    COALESCE(COUNT(e.employee_id), 0) AS employee_count
FROM
    employees e
RIGHT JOIN
    departments d ON e.department_id = d.department_id
GROUP BY
    d.department_name
ORDER BY
    d.department_name;
```

4. **FULL OUTER JOIN Answer (Emulation for MySQL):**

```
SELECT
    e.employee_name,
    d.department_name
FROM
    employees e
LEFT JOIN
    departments d ON e.department_id = d.department_id
UNION
SELECT
    e.employee_name,
    d.department_name
FROM
    employees e
RIGHT JOIN
    departments d ON e.department_id = d.department_id;
```

5. **CROSS JOIN Answer:**

```
SELECT
    p.product_name,
    c.category_name
FROM
```

```
    products p
CROSS JOIN
    categories c;
```

6. **SELF JOIN Answer:**

```
SELECT
    e1.employee_name AS Employee,
    e2.employee_name AS Manager
FROM
    employees e1
LEFT JOIN
    employees e2 ON e1.manager_id = e2.employee_id;
```

Join vs. Subquery

- **Example 1: Employees with Salaries Above the Average Salary**
  - **Using a Subquery (Better Approach):**

    ```
    SELECT
        employee_id,
        employee_name,
        salary
    FROM
        employees
    WHERE
        salary > (
            SELECT
                AVG(salary)
            FROM
                employees
        );
    ```

  - **Using a Join:**

    ```
    SELECT
        e.employee_id,
    ```

```
    e.employee_name,
    e.salary
FROM
    employees e
JOIN
    (
        SELECT
            AVG(salary) AS avg_salary
        FROM
            employees
    ) avg_table ON e.salary > avg_table.avg_salary;
```

- **Explanation:** Using a join adds complexity by introducing an unnecessary derived table (avg_table). It doesn't offer any performance gain and makes the query harder to read, especially when the data involved is already from a single table.

- **Example 2: Listing Products with Their Categories (Including Products Without Categories)**
    - **Using a Join (Better Approach):**

```
SELECT
    p.product_id,
    p.product_name,
    c.category_name
FROM
    products p
LEFT JOIN
    categories c ON p.category_id = c.category_id;
```

- **Using a Subquery:**

```
SELECT
    product_id,
    product_name,
    (
        SELECT
            category_name
        FROM
            categories
        WHERE
```

```
            categories.category_id = products.category_id
    ) AS category_name
FROM
    products;
```

- **Explanation:** The subquery runs once for each product, which can be inefficient for large datasets. Additionally, it complicates the query by embedding a subquery for each product lookup. A LEFT JOIN is more efficient and readable in this case, particularly for scenarios involving many rows.

Stored procedures are named, pre-compiled SQL code units stored and executed within the DBMS, offering reusability for complex tasks without repeated transmission. They enhance maintainability by centralizing logic updates and boost performance through cached execution plans, avoiding repetitive parsing. Security is improved by controlling access via execute permissions, and business rules can be enforced at the database level. Stored procedures streamline database-driven applications by improving organization, performance, security, and maintenance.

Benefits of Using Stored Procedures:

- **Reusability:** Execute a stored procedure multiple times across various applications and users without rewriting the SQL code.
- **Modularity:** Break down intricate database operations into smaller, more manageable units, enhancing code organization.
- **Performance Enhancement:** Reduce network traffic as the stored procedure's code is stored and executed on the database server.
- **Improved Security:** Control data access by granting execute permissions on the stored procedure instead of direct table access.

General Syntax for Creating a Stored Procedure (MySQL):

```
DELIMITER $$
```

```
CREATE PROCEDURE procedure_name (
    -- Optional: Declare input and output parameters
    IN parameter1 datatype,
    OUT parameter2 datatype
)
BEGIN
    -- Optional: Declare variables
    DECLARE variable1 datatype;

    -- SQL statements (procedure logic)
    SELECT ...;
    INSERT ...;
    UPDATE ...;
    DELETE ...;

    -- Optional: Control flow statements
    IF ... THEN ... ELSE ... END IF;
    WHILE ... DO ... END WHILE;
    FOR ... DO ... END FOR;
END $$

DELIMITER ;
```

Key Components Explained:

- **`DELIMITER $$` and `DELIMITER ;`**: MySQL uses `;` as the standard statement terminator. Within a stored procedure, multiple SQL statements exist. To prevent premature termination, the delimiter is temporarily changed (e.g., to `$$`) before the `CREATE PROCEDURE` statement and then reset to `;` after the `END $$` statement.

- **`CREATE PROCEDURE procedure_name (...)`**: This statement initiates the creation of a stored procedure with a specified `procedure_name`. The parentheses can include a list of parameters.
- **Parameters (Optional)**:
    - **`IN parameter1 datatype`**: An input parameter used to pass values *into* the stored procedure. The procedure reads the value of an `IN` parameter.
    - **`OUT parameter2 datatype`**: An output parameter used to pass values *out* of the stored procedure. The procedure sets the value of an `OUT` parameter, which can be retrieved by the caller.
    - Parameter names and their corresponding data types (e.g., `IN student_id INT`) are defined within the parentheses.
- **`BEGIN ... END`**: This block encapsulates the SQL statements that constitute the stored procedure's logic. This is where the actual operations are performed.
- **`DECLARE variable1 datatype;`**: Variables can be declared within the stored procedure to hold intermediate results or for use in calculations.
- **SQL Statements**: Any valid SQL statements (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`) can be included within the `BEGIN ... END` block.
- **Control Flow Statements (Optional)**: Statements like `IF ... THEN ... ELSE ... END IF`, `WHILE ... DO ... END WHILE`, and `FOR ... DO ... END FOR` can be used to implement conditional logic and loops within the stored procedure, adding flexibility.

Example: Retrieving Students by City

Consider a `Students` table with columns: `rollId`, `name`, `dept`, `city`, and `percentage_in_12th`.

```
DELIMITER $$

CREATE PROCEDURE GetStudentsByCity (
   IN input_city VARCHAR(50)
)
BEGIN
   SELECT rollId, name, dept, city, percentage_in_12th
   FROM Students
   WHERE city = input_city;
END $$

DELIMITER ;
```

Executing the Stored Procedure:

To execute the `GetStudentsByCity` stored procedure and retrieve students from 'Delhi':

```
CALL GetStudentsByCity('Delhi');
```

Code

https://onecompiler.com/mysql/43hu5ew66

```
CREATE TABLE Students (
  rollId INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  dept TEXT NOT NULL
);

ALTER TABLE Students
ADD COLUMN city VARCHAR(50);

ALTER TABLE Students
ADD COLUMN percentage_in_12th DECIMAL(5, 2);

ALTER TABLE Students
MODIFY COLUMN dept VARCHAR(100);

-- Insert dummy values
INSERT INTO Students (rollId, name, dept, city, percentage_in_12th) VALUES
(0001, 'kunal ', 'CSE', 'Delhi', 92.10),
(0002, 'aayush ', 'EE', 'Mumbai', 89.60),
(0003, 'alekya ', 'EE', 'Bangalore', 94.30),
(0004, 'anil ', 'CSE', 'Chennai', 87.80),
(005, 'Upasna ', 'Mech', 'Kolkata', 90.25);

--  Show the students table
SELECT * from Students;

-- Filter people having percentage greater than 80
SELECT *
FROM Students
WHERE percentage_in_12th > 80;

-- Tell the top 3 people
SELECT *
FROM Students
ORDER BY percentage_in_12th DESC
LIMIT 3;

-- Get the average percentage of all people
SELECT AVG(percentage_in_12th)
```

```
FROM Students;
```

```
DELIMITER $$

CREATE PROCEDURE GetStudentsByCity (
   IN input_city VARCHAR(50)
)
BEGIN
   SELECT rollId, name, dept, city, percentage_in_12th
   FROM Students
   WHERE city = input_city;
END $$

DELIMITER ;
CALL GetStudentsByCity('Delhi');
```