1. Sub-Queries

A **Sub-query** (also known as a **Nested Query**) is a query embedded inside another query. It is used to retrieve data that will be used in the main query.

Types of Sub-Queries:

1. **Single-row sub-query**: Returns a single value.
2. **Multiple-row sub-query**: Returns multiple values.
3. **Multiple-column sub-query**: Returns multiple columns.
4. **Correlated sub-query**: A sub-query that refers to columns from the outer query.

General Syntax of Sub-query:

```
sql
Copy
SELECT column1, column2
FROM table1
WHERE column1 = (SELECT column1 FROM table2 WHERE condition);
```

Example 1: Single-row Sub-query

To find employees who have the highest salary in the company:

```
sql
Copy
SELECT Name, Salary
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees);
```

Here, the sub-query `(SELECT MAX(Salary) FROM Employees)` retrieves the maximum salary, and the outer query retrieves the employee(s) who earn that salary.

Example 2: Multiple-row Sub-query

To find employees who work in departments with more than 5 employees:

```
sql
Copy
SELECT Name
FROM Employees
WHERE Department IN (SELECT Department
                     FROM Employees
                     GROUP BY Department
                     HAVING COUNT(*) > 5);
```

In this case, the sub-query returns departments that have more than 5 employees, and the main query retrieves employees working in those departments.

Example 3: Correlated Sub-query

A **correlated sub-query** references a column from the outer query. For example, to find employees whose salary is greater than the average salary in their own department:

```
sql
Copy
SELECT Name, Salary, Department
FROM Employees e1
WHERE Salary > (SELECT AVG(Salary)
                FROM Employees e2
                WHERE e1.Department = e2.Department);
```

Here, the sub-query uses `e1.Department` to compare each employee's salary against the average salary in the same department.

---

2. Triggers

A **Trigger** is a special kind of stored procedure that is automatically executed or "triggered" by the database when certain events occur on a specified table or view. Triggers are used for enforcing business rules, auditing data, or automatically updating tables.

Types of Triggers:

- **BEFORE Trigger**: Executes before an insert, update, or delete operation.
- **AFTER Trigger**: Executes after an insert, update, or delete operation.
- **INSTEAD OF Trigger**: Executes in place of an insert, update, or delete operation (mainly used in views).

Trigger Syntax:

```
sql
Copy
CREATE TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT | UPDATE | DELETE }
ON table_name
FOR EACH ROW
BEGIN
  -- Trigger logic here
END;
```

Example 1: AFTER INSERT Trigger

Let’s say you want to keep an audit log whenever a new employee is added to the `Employees` table. You can create an **AFTER INSERT** trigger:

```
sql
Copy
CREATE TRIGGER after_employee_insert
AFTER INSERT
ON Employees
FOR EACH ROW
BEGIN
   INSERT INTO EmployeeAudit (EmployeeID, Name, Action, ActionDate)
   VALUES (NEW.EmployeeID, NEW.Name, 'INSERT', NOW());
END;
```

- **Explanation**: This trigger is fired after a new employee is inserted into the `Employees` table. It inserts a record into the `EmployeeAudit` table with the employee details and the action performed (INSERT).

Example 2: BEFORE UPDATE Trigger

Suppose you want to prevent the salary of an employee from being updated to a value lower than their current salary. You can create a **BEFORE UPDATE** trigger:

```
sql
Copy
CREATE TRIGGER prevent_salary_decrease
BEFORE UPDATE
ON Employees
FOR EACH ROW
BEGIN
   IF NEW.Salary < OLD.Salary THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Salary cannot be decreased';
   END IF;
END;
```

- **Explanation**: This trigger checks if the new salary is less than the old salary before the update is performed. If true, it raises an error and prevents the update.

Example 3: AFTER DELETE Trigger

If you want to log when an employee record is deleted, you can create an **AFTER DELETE** trigger:

```
sql
Copy
CREATE TRIGGER after_employee_delete
AFTER DELETE
ON Employees
FOR EACH ROW
BEGIN
   INSERT INTO EmployeeAudit (EmployeeID, Name, Action, ActionDate)
   VALUES (OLD.EmployeeID, OLD.Name, 'DELETE', NOW());
END;
```

- **Explanation**: This trigger is fired after an employee record is deleted from the `Employees` table. It logs the employee's details in the `EmployeeAudit` table, recording the action as 'DELETE'.

Use Cases for Triggers:

1. **Audit Logging**: To automatically log changes to critical tables (e.g., who, when, and what was changed).
2. **Enforcing Data Integrity**: To enforce business rules, such as ensuring valid data is inserted or updated.

3. **Synchronizing Data**: To automatically update other tables based on changes in one table (e.g., updating inventory when a product is sold).
4. **Preventing Invalid Operations**: To prevent certain operations, such as preventing salary decreases or deleting records that are linked to other records.

Summary of Key Concepts:

Sub-queries:

- **Definition**: A query inside another query that retrieves data for the main query.
- **Types**: Single-row, multiple-row, correlated.
- **Use cases**: To filter results based on conditions derived from other queries.

Triggers:

- **Definition**: Automatically executed SQL code in response to certain events (INSERT, UPDATE, DELETE).
- **Types**: BEFORE, AFTER, INSTEAD OF.
- **Use cases**: To enforce rules, audit data, automatically update related tables, or prevent invalid operations.

Both **sub-queries** and **triggers** are powerful tools for performing complex database operations, ensuring data integrity, and automating business logic within the database.