**SQL Commands and its types (INSERT, UPDATE, DELETE, SELECT, ALTER)**

SQL commands are instructions used to communicate with a database to perform various operations such as creating, reading, updating, and deleting data. These commands are categorized based on their functionality into several types:

* Data Definition Language (DDL)
* Data Manipulation Language (DML)
* Data Control Language (DCL)
* Transaction Control Language (TCL)
* Data Query Language (DQL)

**Data Definition Language (DDL)**

DDL commands define and manage database schema and structure.

1. **CREATE**

Creates new database objects such as tables, views, and indexes.

CREATE TABLE employees (

id INT PRIMARY KEY,

name VARCHAR(100),

position VARCHAR(50),

salary DECIMAL(10, 2)

);

1. **DROP**

Deletes existing database objects.

DROP TABLE employees;

1. **ALTER**

Modifies existing database objects.

ALTER TABLE employees

ADD COLUMN department VARCHAR(50);

1. **TRUNCATE**

Removes all records from a table but retains its structure.

TRUNCATE TABLE employees;

**Data Manipulation Language (DML)**

DML commands manage data within schema objects.

1. **INSERT**

Adds new rows to a table.

INSERT INTO employees (id, name, position, salary)

VALUES (1, 'John Doe', 'Manager', 60000.00);

1. **UPDATE**

Modifies existing rows in a table.

UPDATE employees

SET salary = 65000.00

WHERE id = 1;

1. **DELETE**

Removes existing rows from a table.

DELETE FROM employees

WHERE id = 1;

1. **SELECT**

Retrieves data from one or more tables.

SELECT id, name, position, salary

FROM employees;

**Data Control Language (DCL)**

DCL commands manage access to data within the database.

1. **GRANT**

Gives a user access privileges to the database.

GRANT SELECT, INSERT, UPDATE ON employees TO user\_name;

1. **REVOKE**

Removes access privileges from a user.

REVOKE SELECT, INSERT, UPDATE ON employees FROM user\_name;

**Transaction Control Language (TCL)**

TCL commands manage the transactions within a database.

1. **COMMIT**

Saves all changes made in the current transaction.

COMMIT;

1. **ROLLBACK**

Undoes changes made in the current transaction.

ROLLBACK;

1. **SAVEPOINT**

Sets a point within a transaction to which you can later roll back.

SAVEPOINT savepoint\_name;

**Data Query Language (DQL)**

DQL is used to retrieve data from a database.

1. **SELECT**

SELECT column1, column2, ...

FROM table\_name;

**Operators and Aggregation in SQL**

In SQL, **operators** and **aggregation functions** are essential for performing calculations, comparisons, and data analysis. Operators allow you to manipulate data in queries, while aggregate functions summarize data. Understanding these concepts enables efficient data retrieval and manipulation in databases.

**Types of Operators**

SQL operators can be categorized into different types based on their functionality:

**1. Arithmetic Operators**

**Arithmetic operators** are used to perform mathematical calculations like addition, subtraction, multiplication, and division on numerical data.

* **Addition (+):** Adds two numbers.
* **Subtraction (-):** Subtracts one number from another.
* **Multiplication (\*):** Multiplies two numbers.
* **Division (/):** Divides one number by another.

**Example:** Calculate the total salary of an employee including a bonus.

SELECT salary + bonus AS total\_salary

FROM employees;

This query adds the salary and bonus fields to calculate the total salary.

**2. Comparison Operators**

**Comparison operators** are used to compare values and return true or false based on the condition. They are commonly used in the WHERE clause to filter data.

* **Equals (=):** Checks if two values are equal.
* **Not Equals (<>):** Checks if two values are not equal.
* **Greater Than (>):** Checks if one value is greater than another.
* **Less Than (<):** Checks if one value is less than another.

**Example:** Find employees with a salary greater than 5000.

SELECT name

FROM employees

WHERE salary > 5000;

This query retrieves the names of employees whose salary is greater than 5000.

**3. Logical Operators**

**Logical operators** are used to combine multiple conditions in SQL queries. They help refine searches and filters.

* **AND:** Returns true if all conditions are true.
* **OR:** Returns true if any condition is true.
* **NOT:** Reverses the result of a condition.

**Example:** Find employees who work in the 'Sales' department and earn more than 4000.

SELECT name

FROM employees

WHERE department = 'Sales' AND salary > 4000;

This query retrieves the names of employees who are in the 'Sales' department and have a salary greater than 4000.

**Aggregate Functions in SQL**

**Aggregate functions** in SQL perform calculations on a set of values and return a single value. They are commonly used with the GROUP BY clause to summarize data.

**Types of Aggregate Functions**

* **COUNT:** Returns the number of rows that match a specified condition.
* **SUM:** Adds up the values of a specified column.
* **AVG:** Calculates the average value of a column.
* **MAX:** Returns the highest value in a column.
* **MIN:** Returns the lowest value in a column.

**Examples of Aggregate Functions**

**Example 1: COUNT**

Find the total number of employees in the company.

SELECT COUNT(\*) AS total\_employees

FROM employees;

This query counts the total number of rows in the employees table.

**Example 2: SUM**

Calculate the total sales made by the company.

SELECT SUM(sales) AS total\_sales

FROM sales\_data;

This query adds up the values in the sales column to get the total sales amount.

**Example 3: AVG**

Find the average salary of employees in the 'IT' department.

SELECT AVG(salary) AS average\_salary

FROM employees

WHERE department = 'IT';

This query calculates the average salary of employees in the 'IT' department.

**Example 4: MAX and MIN**

Get the highest and lowest salaries in the company.

SELECT MAX(salary) AS highest\_salary, MIN(salary) AS lowest\_salary

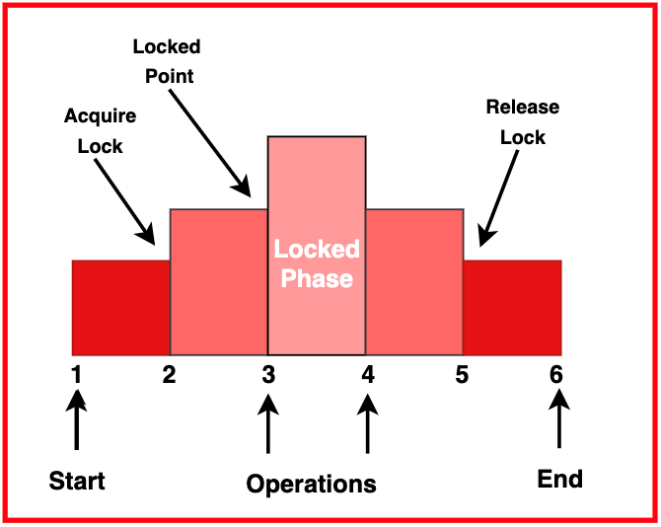
FROM employees;

This query retrieves the maximum and minimum salary values from the employees table.

**When to Use Operators and Aggregation Functions?**

Operators and aggregation functions are essential when you need to:

* Filter data based on conditions using comparison and logical operators.
* Perform calculations like adding or multiplying values using arithmetic operators.
* Summarize data using aggregate functions to find totals, averages, or maximum and minimum values

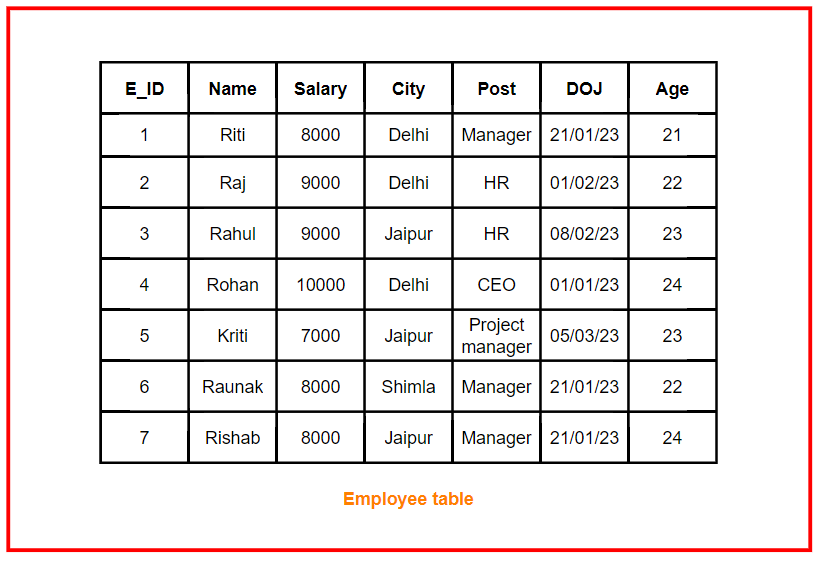


**SQL clauses (WHERE, GROUP BY, HAVING, ORDER BY, LIMIT)**

SQL clauses are used to perform various operations on data within a database. They help retrieve a set or bundles of records from the table and specify a condition on the columns or the records of a table.

**Employee Table**

We will be using the Employee table to run different SQL clauses and analyze their outputs.



**WHERE CLAUSE**

The WHERE clause is used to filter records that meet certain criteria. It is used in SELECT, UPDATE, DELETE, and INSERT statements.

**Syntax:**

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

**Performance Considerations:**

* Ensure that columns used in the WHERE clause are indexed to improve query performance.
* Avoid complex conditions that can slow down the query.
* Use proper comparison operators and avoid using functions on indexed columns in the WHERE clause as this can prevent using indexes.

**GROUP BY CLAUSE**

The GROUP BY clause groups rows that have the same values in specified columns into summary rows, often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG).

**Syntax:**

SELECT column1, COUNT(\*)

FROM table\_name

GROUP BY column1;

**Performance Considerations:**

* Indexes on the columns used in GROUP BY can improve performance.
* Grouping large datasets can be resource-intensive, so ensure that the necessary resources are available.

**HAVING CLAUSE**

The HAVING clause filters groups of rows created by the GROUP BY clause based on a specified condition, often used with aggregate functions.

**Syntax:**

SELECT column1, aggregate\_function(column2)

FROM table\_name

GROUP BY column1

HAVING condition;

**Performance Considerations:**

* The HAVING clause is generally less efficient than the WHERE clause because it operates on the result set after all rows have been processed and grouped.
* To optimize performance, filter as much data as possible using the WHERE clause before applying the HAVING clause.
* Indexing the columns used in the GROUP BY clause can help improve performance.

**ORDER BY CLAUSE**

The ORDER BY clause sorts the result set of a query by one or more columns in ascending (ASC) or descending (DESC) order.

**Syntax:**

SELECT column1, column2, ...

FROM table\_name

ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;

**Performance Considerations:**

* Sorting large datasets can be resource-intensive and slow down query performance.
* Indexing the columns used in the ORDER BY clause can significantly improve performance.
* Use LIMIT to reduce the number of rows returned and minimize the sorting overhead.

Example: This query selects all columns from the Employee table and sorts the results by the Salary column in descending order, showing the highest salaries first.

**LIMIT CLAUSE**

The LIMIT clause specifies the number of records to return from the result set, often used with ORDER BY for pagination or to get the top N records.

**Syntax:**

SELECT column1, column2, ...

FROM table\_name

ORDER BY column\_name [ASC|DESC]

LIMIT number\_of\_records [OFFSET offset\_value];

**Performance Considerations:**

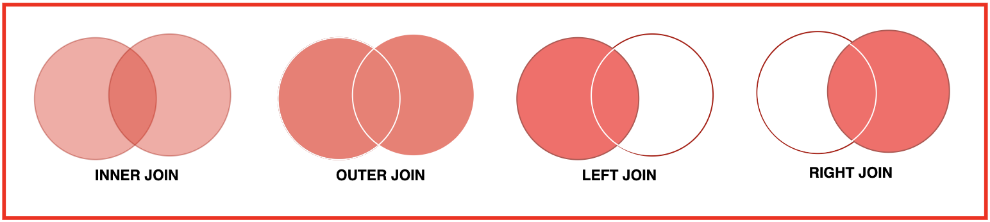
* The LIMIT clause is efficient when you only need a subset of rows, as it reduces the amount of data the database needs to return.
* Using OFFSET with large values can impact performance because the database still needs to process and sort the entire result set before applying the offset.
* Ensure that the columns used in the ORDER BY clause are indexed, especially when dealing with large datasets.

**Joins in SQL(INNER JOIN, OUTER JOIN, LEFT JOIN RIGHT JOIN)**

In SQL, **joins** are used to combine records from two or more tables based on a related column between them. They help to bring data together, making it possible to query data from multiple tables in a relational database.

There are different types of joins that are commonly used, each with its own purpose and behavior. These include:

* INNER JOIN
* OUTER JOIN
* LEFT JOIN
* RIGHT JOIN



**1. INNER JOIN**

The **INNER JOIN** returns only those records that have matching values in both tables. It excludes rows that do not have a match in one of the tables.

**Usage:** Use INNER JOIN when you want to select only the rows that have corresponding matches in both tables.

**Syntax:**

SELECT column\_name(s)

FROM table1

INNER JOIN table2

ON table1.common\_column = table2.common\_column;

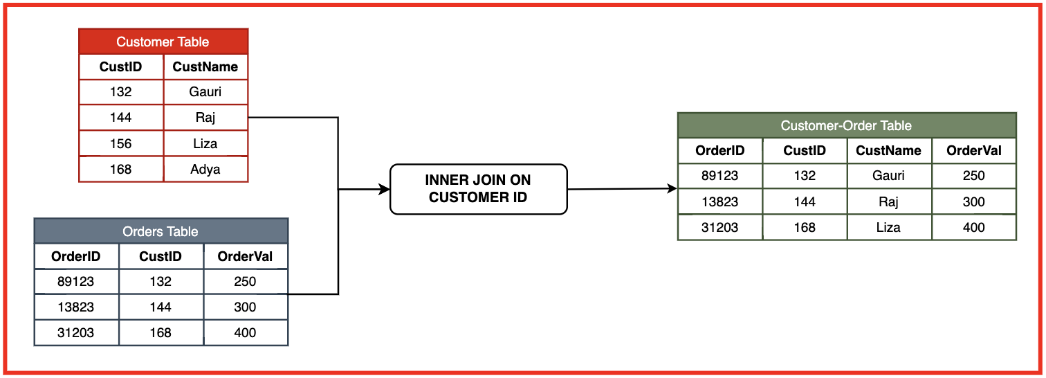
**Example:** Get the list of customers who have placed orders.

SELECT customers.customer\_id, customers.customer\_name, orders.order\_id

FROM customers INNER JOIN orders

ON customers.customer\_id = orders.customer\_id;

**Performance Consideration:** INNER JOIN can be efficient when tables have a strong relationship through foreign keys. However, it may perform poorly if the dataset is large and the join condition is complex.

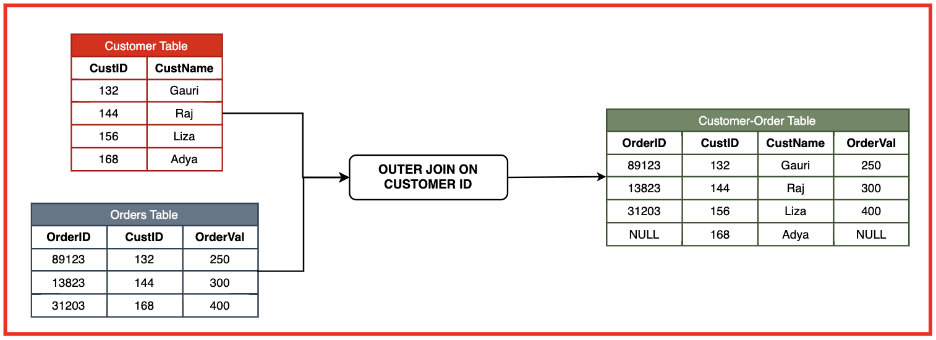


**2. OUTER JOIN**

An **OUTER JOIN** returns all records when there is a match in either left or right table records. There are two types of OUTER JOIN:

* LEFT JOIN (or LEFT OUTER JOIN)
* RIGHT JOIN (or RIGHT OUTER JOIN)

**Usage:** Use OUTER JOIN when you want to retain all rows from one or both tables, even if there is no match between them.



**3. LEFT JOIN**

The **LEFT JOIN** returns all records from the left table and the matched records from the right table. If there is no match, the result is NULL on the side of the right table.

**Syntax:**

SELECT column\_name(s)

FROM table1

LEFT JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:** Retrieve all customers and their orders, including customers who have not placed any orders.

SELECT customers.customer\_id, customers.customer\_name, orders.order\_id

FROM customers

LEFT JOIN orders ON customers.customer\_id = orders.customer\_id;

**Performance Consideration:** LEFT JOIN is generally used when the left table has many rows that should be retained even if there are no corresponding records in the right table. It may be slower than INNER JOIN because it has to retain unmatched rows.

**4. RIGHT JOIN**

The **RIGHT JOIN** returns all records from the right table and the matched records from the left table. If there is no match, the result is NULL on the side of the left table.

**Syntax:**

SELECT column\_name(s)

FROM table1

RIGHT JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:** Retrieve all orders and the customers who placed them, including orders that do not have associated customer details.

SELECT customers.customer\_id, customers.customer\_name, orders.order\_id

FROM customers

RIGHT JOIN orders ON customers.customer\_id = orders.customer\_id;

**Performance Consideration:** RIGHT JOIN can be useful when all rows from the right table are required, even if there are no matching rows in the left table. Similar to LEFT JOIN, it can be slower than INNER JOIN.

**5. FULL OUTER JOIN**

The **FULL OUTER JOIN** returns all records when there is a match in either left or right table records. This join combines the result of LEFT JOIN and RIGHT JOIN, returning rows that have matches in both tables and rows with unmatched data in either table.

**Syntax:**

SELECT column\_name(s)

FROM table1

FULL OUTER JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:** Get all customers and all orders, including those that do not have a corresponding match in the other table.

SELECT customers.customer\_id, customers.customer\_name, orders.order\_id

FROM customers

FULL OUTER JOIN orders ON customers.customer\_id = orders.customer\_id;

**Performance Consideration:** FULL OUTER JOIN can be resource-intensive and slower because it has to combine unmatched records from both tables. It is less commonly used than other joins

**Unions in SQL**

UNION and UNION ALL are SQL operators used to combine the results of two or more SELECT statements.

* **UNION:** Removes duplicate records from the result set, ensuring each row is unique.
* **UNION ALL:** Includes all duplicates, showing all results from both queries.

**The basic syntax of UNION:**

SELECT column1, column2, ...

FROM table1

UNION

SELECT column1, column2, ...

FROM table2;

Combines the result sets of two SELECT statements and removes duplicate records.

**The basic syntax of UNION ALL:**

SELECT column1, column2, ...

FROM table1

UNION ALL

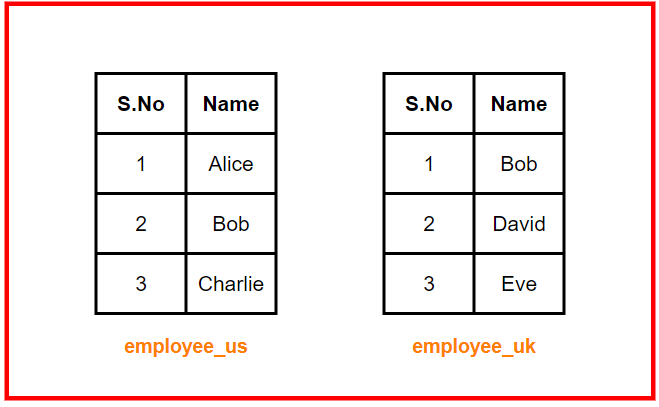
SELECT column1, column2, ...

FROM table2;

Combines the result sets of two SELECT statements and includes all duplicates.

**Example:**

There are two tables:



**UNION:**



**UNION ALL:**

****

We observe that in UNION ALL, “Bob” is not duplicated in output even though it is present in both tables. However, using UNION, all the records are seen in the table.

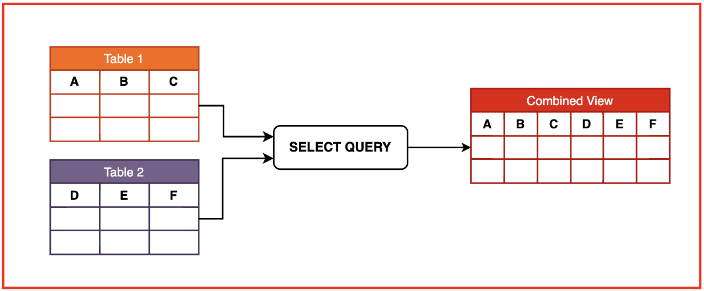
**Views in SQL**

**Views in SQL: Definition and Purpose**

A **view** in SQL is a virtual table that is based on the result set of an SQL query. Views do not store the actual data but present data retrieved from one or more tables. They are used to simplify complex queries, enhance security by limiting access to specific data, and encapsulate logic.

**Purpose:** Views serve several purposes in SQL:

* Simplifying complex queries by hiding underlying joins and filters.
* Restricting access to specific data for security purposes.
* Presenting aggregated or calculated results in a simplified form.
* Ensuring backward compatibility when the structure of underlying tables changes.



**Syntax for Creating Views**

The basic syntax for creating a view is as follows:

CREATE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

In this syntax:

* **view\_name:** The name of the view you are creating.
* **SELECT column1, column2:** The columns you want to include in the view.
* **table\_name:** The table from which data is being retrieved.
* **WHERE condition:** The condition for filtering the data.

**Example:** Create a view that shows only the employees from the Sales department:

CREATE VIEW SalesEmployees AS

SELECT employee\_id, name, salary

FROM employees

WHERE department = 'Sales';

**Updating Views**

Views can be updated under certain conditions, but not all views are updatable, especially complex views that include joins or aggregations. The syntax for updating data through a view is similar to updating a table.

UPDATE view\_name

SET column\_name = value

WHERE condition;

**Example:** Update the salary of a sales employee in the SalesEmployees view:

UPDATE SalesEmployees

SET salary = 60000

WHERE employee\_id = 101;

Note that updates are applied to the underlying table through the view.

**Dropping Views**

To remove a view from the database, you can use the DROP VIEW statement. Dropping a view does not affect the underlying data.

DROP VIEW view\_name;

**Example:** Drop the SalesEmployees view:

DROP VIEW SalesEmployees;

**Types of Views**

**1. Simple Views**

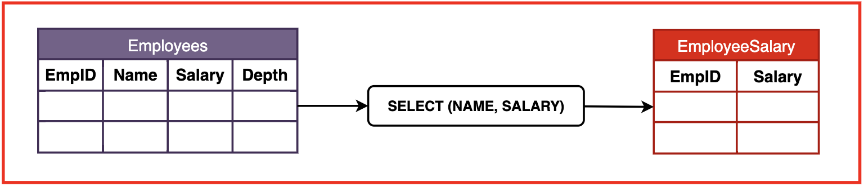
A **simple view** is based on a single table and does not involve any complex logic such as joins or aggregations. Simple views are usually updatable.

**Example:** Create a simple view showing the names and salaries of all employees:

CREATE VIEW EmployeeSalaries AS

SELECT name, salary

FROM employees;



**2. Complex Views**

A **complex view** involves multiple tables, joins, or aggregations, and is usually not updatable. These views are useful for presenting complex relationships or aggregated data in a simplified way.

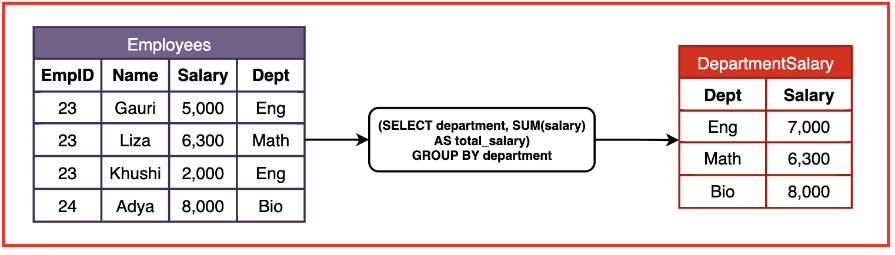
**Example:** Create a complex view that shows the total salary for each department:

CREATE VIEW DepartmentSalaries AS

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department;



**3. Inline Views**

An **inline view** is not a permanent view but a subquery in the FROM clause of a SQL query. It behaves like a temporary view and can simplify the query structure.

**Example:** Use an inline view to find the average salary of employees in each department:

SELECT department, avg\_salary

FROM (SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department);

**4. Materialized Views**

A **materialized view** stores the results of a query physically, unlike regular views that store only the query. Materialized views are used when performance is critical, as they allow faster access to precomputed data. However, they need to be refreshed periodically to ensure the data is up-to-date.

**Example:** Create a materialized view for quickly accessing total sales data:

CREATE MATERIALIZED VIEW TotalSales AS

SELECT SUM(amount) AS total\_sales

FROM sales;

**Advantages of Views**

* **Data Security:** Views can limit access to specific columns or rows, improving security by hiding sensitive data.
* **Query Simplification:** Complex queries can be simplified through views, allowing users to retrieve relevant data without needing to understand the underlying complexity.
* **Logical Data Independence:** Views allow users to access data in a consistent format, even when the structure of the underlying tables changes.
* **Performance:** Materialized views improve performance by storing precomputed results.

**Disadvantages of Views**

* **Performance Overhead:** Regular views can slow down queries since they need to be recomputed every time they are accessed.
* **Limited Updatability:** Not all views are updatable, especially complex ones involving joins and aggregations.
* **Storage:** Materialized views require additional storage since they physically store data

**SQL Sub Queries**

SQL subqueries are queries embedded within another SQL query. They are typically used to retrieve data that will be used in the outer query to perform further operations. Subqueries allow for more complex data retrieval and manipulation.

**Purpose:** Subqueries are used for a variety of purposes, including:

* Filtering data with conditions that depend on other tables.
* Performing calculations before returning the final results.
* Using the result of a subquery as a value for the outer query.
* Creating complex queries without using joins.

**Types of Subqueries**

Subqueries can be categorized into different types based on their structure and the number of rows they return. The main types include:

**1. Single-row Subquery**

A **single-row subquery** returns only one row as its result. It is commonly used with comparison operators like =, >, <, and <=.

**Example:** Find the employee with the highest salary.

SELECT \*

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees);

This query retrieves the details of the employee whose salary is equal to the highest salary in the employees table.

**2. Multiple-row Subquery**

A **multiple-row subquery** returns more than one row as its result. It is often used with operators like IN, ANY, and ALL.

**Example:** Get the names of employees who earn a salary higher than the average salary of their department.

SELECT name

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees GROUP BY department\_id);

This query retrieves the names of employees whose salaries are greater than the average salary in their respective departments.

**3. Correlated Subquery**

A **correlated subquery** references columns from the outer query and is executed once for each row processed by the outer query. This type of subquery depends on the outer query for its values.

**Example:** Find employees whose salaries are above the average salary in their respective departments.

SELECT name, salary

FROM employees e1

WHERE salary > (SELECT AVG(salary)

FROM employees e2

WHERE e1.department\_id = e2.department\_id);

In this query, the subquery calculates the average salary for each department, and the outer query checks if an employee's salary is higher than that average.

**4. Nested Subquery**

A **nested subquery** is a subquery that is nested inside another subquery. It allows for more complex queries by nesting multiple levels of subqueries.

**Example:** Find the names of employees who have a higher salary than the lowest salary in the 'Sales' department.

SELECT name

FROM employees

WHERE salary > (SELECT MIN(salary)

FROM employees

WHERE department = 'Sales');

This query retrieves the names of employees whose salaries are greater than the minimum salary in the 'Sales' department.

**When to Use SQL Subqueries?**

Subqueries can be especially useful in the following scenarios:

* When a query needs to perform a calculation or retrieve a value from another table without using a join.
* When a query's condition depends on aggregated data.
* When combining results from multiple tables in complex ways.
* When working with conditional logic, such as retrieving data that meets certain criteria based on other data.

**Advantages of SQL Subqueries**

* **Simplicity:** Subqueries can make complex queries easier to read and understand by breaking them down into smaller parts.
* **Modularity:** Subqueries can be reused and modified without affecting the outer query.
* **Flexibility:** Subqueries provide a way to retrieve data without the need for joins.

**Disadvantages of SQL Subqueries**

* **Performance:** Subqueries can be less efficient than joins, especially in large datasets, as they may execute multiple times.
* **Complexity:** Correlated subqueries can become complex and difficult to optimize.