# WHAT IS SQL?

## SQL (Structured Query Language):-

SQL is a standardized programming language designed to manage and manipulate relational databases. SQL is used for a wide range of operations, including:

* **Data Retrieval:** Extracting data from a database using queries (SELECT).
* **Data Insertion:** Adding new rows of data to a database (INSERT).
* **Data Update:** Modifying existing data in a database (UPDATE).
* **Data Deletion:** Removing data from a database (DELETE).
* **Schema Creation:** Creating and modifying database structures, such as tables and indexes (CREATE, ALTER, DROP).
* **Data Control:** Managing access to the data using permissions and transactions (GRANT, REVOKE, BEGIN TRANSACTION, COMMIT, ROLLBACK).

## Common SQL Commands:-

* **SELECT**: Retrieve data from one or more tables.
* **INSERT**: Add new data into a table.
* **UPDATE**: Modify existing data within a table.
* **DELETE**: Remove data from a table.
* **CREATE**: Create new database objects like tables, indexes, or views.
* **ALTER**: Modify existing database objects.
* **DROP**: Delete database objects.
* **JOIN**: Combine rows from two or more tables based on a related column.
* **UNION**: Combine the results of two or more SELECT statements.
* **GROUP BY**: Group rows that have the same values in specified columns.
* **ORDER BY**: Sort the result set of a query by one or more columns.

# WHAT IS A QUERY?

A **query** in the context of databases is a request for information or data retrieval from a database. It is a way to interact with the database to retrieve, manipulate, or process the data stored within it.

## How Queries Work:-

* **Syntax:** Queries are written in SQL (Structured Query Language), which follows a specific syntax and set of rules.
* **Execution:** When a query is executed, the database management system (DBMS) processes the query, accesses the relevant data, and returns the result to the user.
* **Optimization:** The DBMS often optimizes queries to execute them efficiently, especially for large datasets.

# FACT TABLES AND DIMENSION TABLES

In data warehousing and OLAP (Online Analytical Processing), data is often organized into fact tables and dimension tables as part of a star schema or snowflake schema. These structures help in efficiently storing and querying large amounts of data for business intelligence and reporting purposes.

## Fact Tables:-

**Definition**

* A **fact table** is the central table in a star schema or snowflake schema that contains quantitative data (facts) for analysis. These facts are typically numerical values, such as sales revenue, profit, or quantity sold, that can be aggregated or summarized.

## Dimension Tables:-

**Definition:**

* A **dimension table** contains descriptive, textual or categorical data that provide context for the facts stored in fact tables. These tables typically contain information about the entities involved in business processes, such as time, products, customers, or locations.

## Relationships Between Fact and Dimension Tables:-

* **Primary Key and Foreign Key Relationship:** Dimension tables have primary keys that uniquely identify each record. These primary keys are referenced as foreign keys in the fact tables.
* **Star Schema:** A star schema is a simple data warehousing schema where the fact table is at the center and is connected to dimension tables radiating outwards.
* **Snowflake Schema:** A snowflake schema is a more complex version of the star schema where dimension tables are normalized into multiple related tables.

## Example Scenario:-

In a retail business, you might have:

* **Fact Table:** Fact\_Sales that stores data about each sale, including the amount sold, units sold, and keys linking to date, product, and store dimensions.
* **Dimension Tables:**
  + Dim\_Product for product details (name, category, manufacturer),
  + Dim\_Date for time details (year, month, day),
  + Dim\_Store for store details (location, manager, type of store).

# SELECT \* FROM Statement

The SELECT \* FROM statement in SQL is used to retrieve all columns from a specific table in a database. The asterisk (\*) is a wildcard character that tells the database to return every column in the table.

## Basic Syntax:-

*SELECT \* FROM table\_name;*

* **SELECT**: Specifies the SQL query is selecting data.
* **\***: Indicates that all columns should be selected.
* **FROM**: Specifies the table from which to retrieve the data.
* **table\_name**: The name of the table you are querying.

## Example Usage:-

**1. Selecting All Data from a Table**

If you have a table named Employees and you want to retrieve all the data, you would use:

*SELECT \* FROM Employees;*

This query would return every column for every row in the Employees table.

**2. Using SELECT \* with a WHERE Clause**

You can also filter the rows returned by using a WHERE clause:

*SELECT \* FROM Employees*

*WHERE Department = 'Sales';*

This query retrieves all columns for rows where the Department column is 'Sales'.

**3. Joining Tables with SELECT \***

You can use SELECT \* with JOINs to retrieve all columns from the joined tables:

*SELECT \* FROM Employees e*

*INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;*

This query selects all columns from both the Employees and Departments tables where there is a matching DepartmentID.

**Best Practices**

* **Use with Caution:** While SELECT \* is convenient, it's not always best practice to use it, especially in large tables, because it can retrieve more data than needed, leading to performance issues.
* **Specific Columns:** It's generally better to specify only the columns you need:

*SELECT EmployeeID, FirstName, LastName FROM Employees;*

# LIMIT Clause in SQL

The LIMIT clause in SQL is used to specify the maximum number of rows that a query should return. This is particularly useful when you want to retrieve a small sample of rows from a large table, or when you only need the first few results of a query.

## Basic Syntax:-

*SELECT column1, column2, ...*

*FROM table\_name*

*LIMIT number\_of\_rows;*

* **number\_of\_rows**: The number of rows you want to retrieve.

## Example Usage:-

**1. Selecting the First 10 Rows**

If you have a table named Employees and you want to retrieve only the first 10 rows, you would use:

*SELECT \* FROM Employees*

*LIMIT 10;*

This query returns the first 10 rows from the Employees table.

**2. Using LIMIT with ORDER BY**

You can use LIMIT together with the ORDER BY clause to get specific results, like the top N rows based on a certain column:

*SELECT \* FROM Employees*

*ORDER BY Salary DESC*

*LIMIT 5;*

This query retrieves the top 5 highest-paid employees.

**3. LIMIT with Offset (MySQL/PostgreSQL)**

In some databases like MySQL and PostgreSQL, you can use LIMIT with an offset to skip a certain number of rows before returning the results:

*SELECT \* FROM Employees*

*LIMIT 5 OFFSET 10;*

This query skips the first 10 rows and then returns the next 5 rows.

Alternatively, you can also use:

*SELECT \* FROM Employees*

*LIMIT 10, 5;*

In this case, 10 is the offset (number of rows to skip), and 5 is the limit (number of rows to return).

## Best Practices:-

* **Testing Queries:** Use LIMIT when testing queries to quickly see a sample of the data.
* **Pagination:** The LIMIT clause is commonly used for pagination, where results are displayed in chunks (e.g., showing 10 results per page).
* **Performance Consideration:** Be mindful when using LIMIT with large datasets, especially with offsets, as it may require the database to scan a large number of rows.

# SELECT DISTINCT in SQL

The SELECT DISTINCT statement is used to retrieve unique values from one or more columns in a table, eliminating duplicate rows from the result set.

## Basic Syntax:-

*SELECT DISTINCT column1, column2, ...*

*FROM table\_name;*

* **DISTINCT**: Ensures that only unique combinations of the specified columns are returned.
* **column1, column2, ...**: The columns from which you want to select distinct values.
* **table\_name**: The table you are querying.

## Example Usage:-

**1. Selecting Distinct Values from a Single Column**

If you have a table named Employees and you want to find all unique departments, you would use:

*SELECT DISTINCT Department*

*FROM Employees;*

This query returns a list of all unique departments in the Employees table.

**2. Selecting Distinct Values from Multiple Columns**

You can also use SELECT DISTINCT with multiple columns. For example, to find unique combinations of Department and JobTitle, you would use:

*SELECT DISTINCT Department, JobTitle*

*FROM Employees;*

This query returns a list of unique combinations of Department and JobTitle across all rows in the Employees table.

**How It Works**

* When you use SELECT DISTINCT with a single column, the query returns all unique values in that column.
* When used with multiple columns, SELECT DISTINCT considers each unique combination of the specified columns as distinct, returning only those combinations.

**Considerations**

* **Performance:** Using SELECT DISTINCT can be resource-intensive on large datasets, as it requires the database to compare each row and eliminate duplicates.
* **Column Order:** The order of columns in the SELECT DISTINCT query can affect the result, as it considers the combination of values across all specified columns.

**Example Scenario**

Suppose you have the following data in a Customers table:

| **CustomerID** | **City** | **Country** |
| --- | --- | --- |
| 1 | New York | USA |
| 2 | Los Angeles | USA |
| 3 | New York | USA |
| 4 | London | UK |
| 5 | New York | USA |
| 6 | London | UK |

If you want to retrieve all unique cities, you would write:

*SELECT DISTINCT City*

*FROM Customers;*

This query would return:

* New York
* Los Angeles
* London

# WHERE Clause in SQL

The WHERE clause in SQL is used to filter records in a query based on specified conditions. It allows you to specify which rows should be included in the results of a SELECT, UPDATE, DELETE, or INSERT statement.

## Basic Syntax

*SELECT column1, column2, ...*

*FROM table\_name*

*WHERE condition;*

* **condition**: The criteria that must be met for rows to be included in the result set.

## Example Usage

**1. Basic WHERE Clause**

To select rows where a condition is true, such as finding employees in the "Sales" department:

*SELECT \* FROM Employees*

*WHERE Department = 'Sales';*

This query returns all columns for rows where the Department column is "Sales".

**2. Using WHERE with Comparison Operators**

You can use comparison operators like =, !=, <, >, <=, >=:

*SELECT \* FROM Employees*

*WHERE Salary > 50000;*

This query returns all employees with a salary greater than $50,000.

**3. Using WHERE with Logical Operators**

You can combine multiple conditions using logical operators like AND, OR, and NOT:

* **AND**: Requires all conditions to be true.

*SELECT \* FROM Employees*

*WHERE Department = 'Sales' AND Salary > 50000;*

This query returns employees in the "Sales" department with a salary greater than $50,000.

* **OR**: Requires at least one condition to be true.

*SELECT \* FROM Employees*

*WHERE Department = 'Sales' OR Department = 'Marketing';*

This query returns employees in either the "Sales" or "Marketing" departments.

* **NOT**: Negates a condition.

*SELECT \* FROM Employees*

*WHERE NOT Department = 'Sales';*

This query returns all employees except those in the "Sales" department.

**4. Using WHERE with IN**

The IN operator allows you to specify multiple possible values for a column:

*SELECT \* FROM Employees*

*WHERE Department IN ('Sales', 'Marketing', 'HR');*

This query returns employees whose department is either "Sales", "Marketing", or "HR".

**5. Using WHERE with BETWEEN**

The BETWEEN operator is used to filter the result set within a range of values:

*SELECT \* FROM Employees*

*WHERE Salary BETWEEN 40000 AND 60000;*

This query returns employees with a salary between $40,000 and $60,000 inclusive.

**6. Using WHERE with LIKE**

The LIKE operator is used for pattern matching in string columns:

*SELECT \* FROM Employees*

*WHERE FirstName LIKE 'A%';*

This query returns employees whose first name starts with "A".

# Comments in SQL

Comments in SQL are used to add explanatory notes or disable certain parts of SQL code from being executed. Comments are ignored by the SQL engine and do not affect the execution of the code. They are useful for documenting your SQL queries, explaining complex logic, or temporarily disabling parts of your code during development and testing.

## Types of Comments in SQL

1. **Single-Line Comments:**
   * These are comments that occupy a single line and are typically used for brief explanations.
   * **Syntax:** Use -- (double hyphen) to start a single-line comment. Everything after -- on that line is treated as a comment.

**Example:**

*SELECT \* FROM Employees; -- This query selects all columns from the Employees table*

1. **Multi-Line Comments:**
   * These comments can span multiple lines and are used for more detailed explanations or when commenting out larger blocks of code.
   * **Syntax:** Enclose the comment with /\* to start and \*/ to end.

**Example:**

*/\**

*This query selects all columns from the Employees table.*

*It retrieves data for analysis purposes.*

*\*/*

*SELECT \* FROM Employees;*

* + *Multi-line comments can also be used within a single line:*

*SELECT \* /\* This comment is inline \*/ FROM Employees;*

**Usage Scenarios**

* **Documentation:** Use comments to explain the purpose of queries, functions, or complex logic within your SQL code.
* **Debugging:** Temporarily disable parts of your SQL code by commenting them out to test different query configurations.
* **Collaboration:** Provide context for other developers who may work on the same SQL codebase, making it easier to understand.

**Example**

Here’s a more detailed example with both single-line and multi-line comments:

*-- Select the first 10 employees from the Employees table*

*SELECT \**

*FROM Employees*

*WHERE Department = 'Sales' -- Only include employees from the Sales department*

*LIMIT 10;*

*/\**

*This section of code calculates the average salary*

*across all departments and retrieves the result.*

*The result is useful for salary comparison and budgeting.*

*\*/*

*SELECT Department, AVG(Salary) AS AverageSalary*

*FROM Employees*

*GROUP BY Department;*

# ORDER BY Clause in SQL

The ORDER BY clause is used to sort the result set of a SQL query in either ascending or descending order. This sorting can be done based on one or more columns. It is often used to make query results more readable or to present data in a specific sequence.

## Basic Syntax:-

*SELECT column1, column2, ...*

*FROM table\_name*

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

* **column1, column2, ...**: The columns by which the result set should be sorted.
* **ASC**: Sorts the results in ascending order (default).
* **DESC**: Sorts the results in descending order.

## Example Usage:-

**1. Sorting by a Single Column**

To retrieve employees sorted by their last names in ascending order:

*SELECT \* FROM Employees*

*ORDER BY LastName;*

By default, if you don't specify ASC or DESC, the results will be sorted in ascending order.

**2. Sorting by a Single Column in Descending Order**

To sort employees by their salary in descending order:

*SELECT \* FROM Employees*

*ORDER BY Salary DESC;*

This query returns employees with the highest salary first.

**3. Sorting by Multiple Columns**

To sort employees first by department and then by salary within each department:

*SELECT \* FROM Employees*

*ORDER BY Department, Salary DESC;*

This query sorts the results primarily by Department in ascending order and then by Salary in descending order within each department.

**4. Sorting with a WHERE Clause**

You can use ORDER BY in combination with WHERE to sort filtered results:

*SELECT \* FROM Employees*

*WHERE Department = 'Sales'*

*ORDER BY LastName;*

This query first filters employees to include only those in the "Sales" department and then sorts them by LastName.

**Considerations**

* **Performance:** Sorting can be resource-intensive, especially with large datasets. Indexes on columns used in ORDER BY can help improve performance.
* **NULL Values:** The placement of NULL values in the sort order depends on the database system:
  + In ascending order, NULL values often appear first.
  + In descending order, NULL values often appear last.
* **Sorting for Pagination:** When implementing pagination, ORDER BY is often used to ensure results are presented in a consistent order:

*SELECT \* FROM Employees*

*ORDER BY LastName*

*LIMIT 10 OFFSET 20;*

This query retrieves 10 rows starting from the 21st row (useful for page 3 if showing 10 results per page).

# Order of SQL Clauses in Commands

SQL commands typically follow a specific order of clauses to construct a complete query. This sequence ensures that the SQL engine processes the query correctly and returns the desired results. Below is the standard order in which SQL clauses are written:

Here is an example combining all the clauses in the correct order:

*SELECT Department, AVG(Salary) AS AverageSalary*

*FROM Employees*

*INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID*

*WHERE Employees.Status = 'Active'*

*GROUP BY Department*

*HAVING AVG(Salary) > 50000*

*ORDER BY AverageSalary DESC*

*LIMIT 10;*

* **SELECT**: Chooses the columns to be displayed.
* **FROM**: Specifies the main table.
* **INNER JOIN**: Joins additional tables.
* **WHERE**: Filters rows based on conditions.
* **GROUP BY**: Groups rows by department.
* **HAVING**: Filters groups based on aggregated values.
* **ORDER BY**: Sorts the results.
* **LIMIT**: Limits the number of rows returned.

# Comparisons in SQL

In SQL, comparisons are used to evaluate conditions and filter data based on specific criteria. Comparison operators are used in the WHERE clause, as well as in other clauses, to compare column values against constants or other column values. Here’s an overview of common comparison operators and their usage:

## Comparison Operators:-

1. **Equality (=)**
   * **Description:** Checks if two values are equal.

*SELECT \* FROM Employees*

*WHERE Department = 'Sales';*

* + This query retrieves all employees whose department is 'Sales'.

1. **Inequality (!= or <>)**
   * **Description:** Checks if two values are not equal. Both != and <> are commonly used, with <> being the SQL standard.

*SELECT \* FROM Employees*

*WHERE Department <> 'Sales';*

* + This query retrieves all employees whose department is not 'Sales'.

1. **Greater Than (>)**
   * **Description:** Checks if a value is greater than another value.

*SELECT \* FROM Employees*

*WHERE Salary > 50000;*

* + This query retrieves all employees with a salary greater than $50,000.

1. **Less Than (<)**
   * **Description:** Checks if a value is less than another value.

*SELECT \* FROM Employees*

*WHERE Salary < 30000;*

* + This query retrieves all employees with a salary less than $30,000.

1. **Greater Than or Equal To (>=)**
   * **Description:** Checks if a value is greater than or equal to another value.

*SELECT \* FROM Employees*

*WHERE Salary >= 50000;*

* + This query retrieves all employees with a salary of $50,000 or more.

1. **Less Than or Equal To (<=)**
   * **Description:** Checks if a value is less than or equal to another value.

*SELECT \* FROM Employees*

*WHERE Salary <= 30000;*

* + This query retrieves all employees with a salary of $30,000 or less.

1. **BETWEEN**
   * **Description:** Checks if a value is within a specified range (inclusive).

*SELECT \* FROM Employees*

*WHERE Salary BETWEEN 40000 AND 60000;*

* + This query retrieves all employees with a salary between $40,000 and $60,000, inclusive.

1. **IN**
   * **Description:** Checks if a value matches any value in a list of values.

*SELECT \* FROM Employees*

*WHERE Department IN ('Sales', 'Marketing', 'HR');*

* + This query retrieves all employees who work in the 'Sales', 'Marketing', or 'HR' departments.

1. **LIKE**
   * **Description:** Checks if a value matches a specified pattern. Useful for string comparisons with wildcard characters.
   * **Wildcards:**
     + %: Matches zero or more characters.
     + \_: Matches exactly one character.
   * **Example:**

*SELECT \* FROM Employees*

*WHERE LastName LIKE 'S%';*

* + This query retrieves all employees whose last name starts with 'S'.

1. **IS NULL**
   * **Description:** Checks if a value is NULL.

*SELECT \* FROM Employees*

*WHERE ManagerID IS NULL;*

* + This query retrieves all employees who do not have a manager (i.e., ManagerID is NULL).

1. **IS NOT NULL**
   * **Description:** Checks if a value is not NULL.

*SELECT \* FROM Employees*

*WHERE ManagerID IS NOT NULL;*

* + This query retrieves all employees who have a manager (i.e., ManagerID is not NULL).

## Combining Comparison Operators:-

* **AND**: Combines multiple conditions, all of which must be true.

*SELECT \* FROM Employees*

*WHERE Salary > 30000 AND Department = 'Sales';*

* **OR**: Combines multiple conditions, at least one of which must be true.

*SELECT \* FROM Employees*

*WHERE Department = 'Sales' OR Department = 'Marketing';*

* **NOT**: Negates a condition.

*SELECT \* FROM Employees*

*WHERE NOT Department = 'Sales';*

# Wildcards in SQL

Wildcards are special characters used in SQL LIKE expressions to search for patterns within columns. They allow for more flexible queries by matching various forms of data, which is especially useful for searching text data.

## Common Wildcards:-

1. **Percent Sign (%)**
   * **Description:** Represents zero or more characters.
   * **Usage:** Used when you want to match any sequence of characters.

*SELECT \* FROM Employees*

*WHERE LastName LIKE 'S%';*

* + - This query retrieves all employees whose last name starts with 'S'.

1. **Underscore (\_)**
   * **Description:** Represents exactly one character.
   * **Usage:** Used when you want to match a single character in a specific position.

*SELECT \* FROM Employees*

*WHERE LastName LIKE 'S\_n';*

* + - This query retrieves all employees whose last name starts with 'S' and ends with 'n', with any single character in between (e.g., 'Swan').

1. **Square Brackets ([])** (SQL Server-specific)
   * **Description:** Matches any single character within the brackets.
   * **Usage:** Useful for matching specific sets of characters.

*SELECT \* FROM Employees*

*WHERE LastName LIKE 'S[aeiou]n';*

* + - This query retrieves employees whose last name starts with 'S', ends with 'n', and has exactly one vowel in between (e.g., 'San').

1. **Caret (^)** (SQL Server-specific)
   * **Description:** Specifies characters that should not be matched within square brackets.
   * **Usage:** Used to exclude certain characters from the match.

*SELECT \* FROM Employees*

*WHERE LastName LIKE 'S[^aeiou]n';*

* + - This query retrieves employees whose last name starts with 'S', ends with 'n', and has any non-vowel character in between (e.g., 'Sun').

1. **Hyphen (-)** (SQL Server-specific)
   * **Description:** Specifies a range of characters within square brackets.
   * **Usage:** Useful for matching a range of characters.

*SELECT \* FROM Employees*

*WHERE LastName LIKE 'S[0-9]n';*

* + - This query retrieves employees whose last name starts with 'S', ends with 'n', and has a single digit in between (e.g., 'S2n').

1. **Escape Characters**
   * **Description:** Allows you to search for wildcard characters as literal values by escaping them.
   * **Usage:** Useful when you need to find rows containing literal percent signs (%) or underscores (\_).

*SELECT \* FROM Products*

*WHERE ProductName LIKE '100\%' ESCAPE '\';*

* + - This query retrieves products with names that include '100%' as a literal string.

## Examples of Wildcards in Queries:-

1. **Search for Names Starting with a Specific Letter**

*SELECT \* FROM Employees*

*WHERE FirstName LIKE 'A%';*

* + Retrieves employees whose first names start with 'A'.

1. **Search for Names Ending with a Specific Letter**

*SELECT \* FROM Employees*

*WHERE LastName LIKE '%s';*

* + Retrieves employees whose last names end with 's'.

1. **Search for Names with a Specific Pattern**

*SELECT \* FROM Employees*

*WHERE MiddleName LIKE '\_a%';*

* + Retrieves employees whose middle names have 'a' as the second character.

1. **Search for a Range of Characters**

*SELECT \* FROM Products*

*WHERE ProductCode LIKE 'A[1-3]';*

* + Retrieves products where the product code starts with 'A' followed by a digit between 1 and 3.

1. **Search for Names Containing Specific Characters**

*SELECT \* FROM Employees*

*WHERE LastName LIKE '%a%';*

* + Retrieves employees whose last names contain at least one 'a'.

# SQL Aliases (AS)

In SQL, aliases are temporary names given to tables or columns for the duration of a query. Aliases simplify query writing and improve readability, especially when dealing with complex queries or multiple tables.

## Using Aliases for Columns:-

Aliases for columns are useful when you want to rename a column in the result set or use a more readable or descriptive name for a column.

**Basic Syntax**

*SELECT column\_name AS alias\_name*

*FROM table\_name;*

* **column\_name**: The original column name.
* **alias\_name**: The new name you want to use for the column in the result set.

**Example**

*SELECT FirstName AS 'First Name', LastName AS 'Last Name'*

*FROM Employees;*

* This query renames the FirstName column to 'First Name' and the LastName column to 'Last Name' in the result set.

## Using Aliases for Tables:-

Aliases for tables are helpful in simplifying complex queries, especially those involving multiple joins or subqueries.

**Basic Syntax**

*SELECT column\_name*

*FROM table\_name AS alias\_name;*

* **table\_name**: The original table name.
* **alias\_name**: The new name you want to use for the table in the query.

**Example**

*SELECT e.FirstName, e.LastName, d.DepartmentName*

*FROM Employees AS e*

*INNER JOIN Departments AS d ON e.DepartmentID = d.DepartmentID;*

* Here, Employees is aliased as e, and Departments is aliased as d. This makes the query shorter and more readable.

## Using Aliases in GROUP BY and ORDER BY:-

Aliases can be used in GROUP BY and ORDER BY clauses to refer to the column names that were defined in the SELECT clause.

**Example**

*SELECT Department, COUNT(\*) AS EmployeeCount*

*FROM Employees*

*GROUP BY Department*

*ORDER BY EmployeeCount DESC;*

* In this query, EmployeeCount is an alias for the result of COUNT(\*). The ORDER BY clause uses this alias to sort the results.

# SQL Operations

## Arithmetic Operations in SQL:-

Arithmetic operations in SQL are used to perform calculations on numeric data within queries. These operations can be applied to columns, literals, or results of other expressions. Here’s a guide to common arithmetic operations and how to use them in SQL:

## Common Arithmetic Operators:-

1. **Addition (+)**
   * **Description:** Adds two numbers or values.

*SELECT column1 + column2 AS result*

*FROM table\_name;*

* + **Example:**

*SELECT Salary + Bonus AS TotalCompensation*

*FROM Employees;*

* + This query calculates the total compensation by adding the Salary and Bonus columns.

1. **Subtraction (-)**
   * **Description:** Subtracts one number or value from another.

*SELECT column1 - column2 AS result*

*FROM table\_name;*

* + **Example:**

*SELECT Salary - Deductions AS NetSalary*

*FROM Employees;*

* + This query calculates the net salary by subtracting Deductions from Salary.

1. **Multiplication (\*)**
   * **Description:** Multiplies two numbers or values.

*SELECT column1 \* column2 AS result*

*FROM table\_name;*

* + **Example:**

*SELECT Quantity \* UnitPrice AS TotalPrice*

*FROM OrderDetails;*

* + This query calculates the total price by multiplying Quantity by UnitPrice.

1. **Division (/)**
   * **Description:** Divides one number or value by another.

*SELECT column1 / column2 AS result*

*FROM table\_name;*

* + **Example:**

*SELECT TotalAmount / NumberOfItems AS AveragePrice*

*FROM Orders;*

* + This query calculates the average price by dividing TotalAmount by NumberOfItems.

1. **Modulus (%)**
   * **Description:** Returns the remainder of a division operation.

*SELECT column1 % column2 AS result*

*FROM table\_name;*

* + **Example:**

*SELECT TotalAmount % 100 AS Remainder*

*FROM Orders;*

* + This query calculates the remainder when TotalAmount is divided by 100.

## Using Arithmetic Operations in SQL Queries:-

Arithmetic operations can be used in various parts of SQL queries, such as SELECT, WHERE, HAVING, and ORDER BY clauses.

**In SELECT Clause**

*SELECT ProductName, UnitPrice, Quantity, (UnitPrice \* Quantity) AS TotalPrice*

*FROM OrderDetails;*

* This query selects product names, unit prices, quantities, and calculates the total price for each product.

**In WHERE Clause**

*SELECT \* FROM Employees*

*WHERE Salary - Deductions > 50000;*

* This query retrieves employees whose net salary (after deductions) is greater than $50,000.

**In HAVING Clause**

*SELECT Department, AVG(Salary) AS AverageSalary*

*FROM Employees*

*GROUP BY Department*

*HAVING AVG(Salary) > 60000;*

* This query groups employees by department, calculates the average salary for each department, and filters departments where the average salary is greater than $60,000.

**In ORDER BY Clause**

*SELECT ProductName, Quantity, UnitPrice, (Quantity \* UnitPrice) AS TotalValue*

*FROM OrderDetails*

*ORDER BY TotalValue DESC;*

* This query sorts products by the total value, calculated as Quantity multiplied by UnitPrice, in descending order.

# Aggregation in SQL

Aggregation in SQL involves summarizing data to produce a single value from multiple rows. This is useful for analyzing and reporting on datasets by calculating totals, averages, counts, and other summary statistics. SQL provides several aggregate functions to perform these operations.

## Common Aggregate Functions:-

1. **COUNT()**
   * **Description:** Returns the number of rows that match a specified condition.

*SELECT COUNT(column\_name) AS count\_alias*

*FROM table\_name*

*WHERE condition;*

* + **Example:**

*SELECT COUNT(\*) AS TotalEmployees*

*FROM Employees;*

* + This query counts the total number of rows in the Employees table.

1. **SUM()**
   * **Description:** Returns the total sum of a numeric column.
   * **Syntax:**

*SELECT SUM(column\_name) AS sum\_alias*

*FROM table\_name*

*WHERE condition;*

* + **Example:**

*SELECT SUM(Salary) AS TotalSalaries*

*FROM Employees;*

* + This query calculates the total sum of all salaries in the Employees table.

1. **AVG()**
   * **Description:** Returns the average value of a numeric column.

*SELECT AVG(column\_name) AS avg\_alias*

*FROM table\_name*

*WHERE condition;*

* + **Example:**

*SELECT AVG(Salary) AS AverageSalary*

*FROM Employees;*

* + This query calculates the average salary of all employees.

1. **MIN()**
   * **Description:** Returns the smallest value in a column.

*SELECT MIN(column\_name) AS min\_alias*

*FROM table\_name*

*WHERE condition;*

* + **Example:**

*SELECT MIN(Salary) AS LowestSalary*

*FROM Employees;*

* + This query retrieves the lowest salary from the Employees table.

1. **MAX()**
   * **Description:** Returns the largest value in a column.

*SELECT MAX(column\_name) AS max\_alias*

*FROM table\_name*

*WHERE condition;*

* + **Example:**

*SELECT MAX(Salary) AS HighestSalary*

*FROM Employees;*

* + This query retrieves the highest salary from the Employees table.

## Using Aggregation with GROUP BY:-

Aggregation functions are often used with the GROUP BY clause to summarize data across different groups.

**Syntax**

*SELECT column\_name, aggregate\_function(column\_name) AS aggregate\_alias*

*FROM table\_name*

*GROUP BY column\_name;*

**Example**

*SELECT Department, AVG(Salary) AS AverageSalary*

*FROM Employees*

*GROUP BY Department;*

* This query calculates the average salary for each department by grouping the results by the Department column.

## Using Aggregation with HAVING:-

The HAVING clause is used to filter results after aggregation, similar to how WHERE filters results before aggregation.

**Syntax**

*SELECT column\_name, aggregate\_function(column\_name) AS aggregate\_alias*

*FROM table\_name*

*GROUP BY column\_name*

*HAVING aggregate\_function(column\_name) condition;*

**Example**

*SELECT Department, COUNT(\*) AS NumberOfEmployees*

*FROM Employees*

*GROUP BY Department*

*HAVING COUNT(\*) > 10;*

* This query retrieves departments with more than 10 employees by counting the number of employees in each department and filtering based on this count.

## Combining Multiple Aggregation Functions:-

You can use multiple aggregation functions in a single query to get various summary statistics.

**Example**

*SELECT Department,*

*COUNT(\*) AS NumberOfEmployees,*

*SUM(Salary) AS TotalSalaries,*

*AVG(Salary) AS AverageSalary,*

*MIN(Salary) AS LowestSalary,*

*MAX(Salary) AS HighestSalary*

*FROM Employees*

*GROUP BY Department;*

* This query provides a comprehensive summary of employees in each department, including the number of employees, total salaries, average salary, lowest salary, and highest salary.

# GROUP BY and HAVING Clauses in SQL

The GROUP BY and HAVING clauses are used in SQL to group rows that share common values and to filter these groups based on certain conditions. They are commonly used with aggregate functions to summarize data.

## GROUP BY Clause:-

The GROUP BY clause is used to arrange identical data into groups. It is often used with aggregate functions (such as COUNT(), SUM(), AVG(), MIN(), and MAX()) to perform calculations on each group of data.

**Syntax**

*SELECT column1, aggregate\_function(column2) AS aggregate\_alias*

*FROM table\_name*

*GROUP BY column1;*

**Example**

Suppose we have a table Sales with columns Salesperson, Region, and Amount.

To get the total sales amount for each salesperson:

*SELECT Salesperson, SUM(Amount) AS TotalSales*

*FROM Sales*

*GROUP BY Salesperson;*

* This query groups rows by Salesperson and calculates the total sales amount for each salesperson.

## HAVING Clause:-

The HAVING clause is used to filter groups based on aggregate functions. Unlike the WHERE clause, which filters rows before grouping, HAVING filters groups after the aggregation is performed.

**Syntax**

*SELECT column1, aggregate\_function(column2) AS aggregate\_alias*

*FROM table\_name*

*GROUP BY column1*

*HAVING aggregate\_function(column2) condition;*

**Example**

To get the salespeople who have a total sales amount greater than $10,000:

*SELECT Salesperson, SUM(Amount) AS TotalSales*

*FROM Sales*

*GROUP BY Salesperson*

*HAVING SUM(Amount) > 10000;*

* This query first groups the data by Salesperson, calculates the total sales for each, and then filters the results to include only those salespeople with total sales greater than $10,000.

## Combining GROUP BY and HAVING:-

You can use GROUP BY and HAVING together to group data and then filter those groups based on aggregated results.

**Example 1: Grouping and Filtering by Aggregate Results**

To find departments with more than 5 employees and whose average salary is greater than $60,000:

*SELECT Department, COUNT(\*) AS NumberOfEmployees, AVG(Salary) AS AverageSalary*

*FROM Employees*

*GROUP BY Department*

*HAVING COUNT(\*) > 5 AND AVG(Salary) > 60000;*

* This query groups employees by Department, calculates the number of employees and the average salary for each department, and filters out departments where the number of employees is 5 or fewer and the average salary is $60,000 or less.

**Example 2: Filtering Before Grouping with WHERE, and After Grouping with HAVING**

To find the average salary of employees in each department where the department has more than 10 employees and where the total salary of all employees in that department is more than $100,000:

*SELECT Department, AVG(Salary) AS AverageSalary*

*FROM Employees*

*GROUP BY Department*

*HAVING COUNT(\*) > 10 AND SUM(Salary) > 100000;*

* This query first groups employees by Department, calculates the average salary for each department, and then uses the HAVING clause to filter departments with more than 10 employees and where the total salary is greater than $100,000.

# Handling NULL Values in SQL

In SQL, NULL represents a missing or undefined value. It is important to handle NULL values properly in SQL queries to ensure accurate data manipulation and retrieval. Here’s a guide on how to work with NULL values in SQL:

## Checking for NULL Values:-

* **IS NULL**: Checks if a column or expression is NULL.

*SELECT \* FROM table\_name*

*WHERE column\_name IS NULL;*

* + **Example**:

*SELECT \* FROM Employees*

*WHERE ManagerID IS NULL;*

* + - Retrieves employees who do not have a manager assigned (i.e., ManagerID is NULL).
* **IS NOT NULL**: Checks if a column or expression is not NULL.

*SELECT \* FROM table\_name*

*WHERE column\_name IS NOT NULL;*

* + **Example**:

*SELECT \* FROM Employees*

*WHERE ManagerID IS NOT NULL;*

* + - Retrieves employees who have a manager assigned (i.e., ManagerID is not NULL).

# SQL Joins

SQL joins are used to combine rows from two or more tables based on a related column between them. Joins are essential for querying data from multiple tables and can be categorized into several types based on how they combine the data.

## INNER JOIN:-

* + **Description**: Retrieves only the rows that have matching values in both tables.

*SELECT columns*

*FROM table1*

*INNER JOIN table2 ON table1.column = table2.column;*

* + **Example**:

*SELECT Employees.FirstName, Departments.DepartmentName*

*FROM Employees*

*INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;*

* + This query returns employees along with their department names where there is a match between DepartmentID in both tables.

## LEFT JOIN (or LEFT OUTER JOIN):-

* + **Description**: Retrieves all rows from the left table and the matched rows from the right table. If no match is found, NULL values are returned for columns from the right table.

*SELECT columns*

*FROM table1*

*LEFT JOIN table2 ON table1.column = table2.column;*

* + **Example**:

*SELECT Employees.FirstName, Departments.DepartmentName*

*FROM Employees*

*LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;*

* + This query returns all employees and their department names if available. Employees without a department will still be included, with NULL in the DepartmentName column.

## RIGHT JOIN (or RIGHT OUTER JOIN):-

* + **Description**: Retrieves all rows from the right table and the matched rows from the left table. If no match is found, NULL values are returned for columns from the left table.

*SELECT columns*

*FROM table1*

*RIGHT JOIN table2 ON table1.column = table2.column;*

* + **Example**:

*SELECT Employees.FirstName, Departments.DepartmentName*

*FROM Employees*

*RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;*

* + This query returns all departments and their employees if available. Departments without employees will still be included, with NULL in the FirstName column.

## FULL JOIN (or FULL OUTER JOIN):-

* + **Description**: Retrieves all rows from both tables. Rows from the left table without a match in the right table and rows from the right table without a match in the left table will have NULL values for the columns of the table without a match.

*SELECT columns*

*FROM table1*

*FULL JOIN table2 ON table1.column = table2.column;*

* + **Example**:

*SELECT Employees.FirstName, Departments.DepartmentName*

*FROM Employees*

*FULL JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;*

* + This query returns all employees and all departments, with NULL values where there is no match in either table.

## CROSS JOIN:-

* + **Description**: Produces a Cartesian product of the two tables, i.e., every row of the first table is combined with every row of the second table.

*SELECT columns*

*FROM table1*

*CROSS JOIN table2;*

* + **Example**:

*SELECT Employees.FirstName, Departments.DepartmentName*

*FROM Employees*

*CROSS JOIN Departments;*

* + This query returns every combination of employees and departments, which can result in a large number of rows if both tables are large.

## Self-Join:-

* **Description**: A self-join is a regular join, but the table is joined with itself.

*SELECT a.column1, b.column2*

*FROM table a*

*INNER JOIN table b ON a.column = b.column;*

* **Example**:

*SELECT e1.EmployeeName AS Employee, e2.EmployeeName AS Manager*

*FROM Employees e1*

*INNER JOIN Employees e2 ON e1.ManagerID = e2.EmployeeID;*

* This query finds pairs of employees and their managers where both employees are from the same Employees table.

## Summary of Join Types:-

* **INNER JOIN**: Matches rows between tables; only returns rows with matching values.
* **LEFT JOIN**: Returns all rows from the left table and matched rows from the right table; includes unmatched rows from the left table with NULL values for the right table's columns.
* **RIGHT JOIN**: Returns all rows from the right table and matched rows from the left table; includes unmatched rows from the right table with NULL values for the left table's columns.
* **FULL JOIN**: Returns all rows from both tables; includes unmatched rows from both tables with NULL values.
* **CROSS JOIN**: Returns a Cartesian product of both tables; every row from the first table is combined with every row from the second table.
* **Self-Join**: Joins a table with itself to find relationships within the same table.

Joins are crucial for relational databases, enabling complex queries and efficient data retrieval by linking related data across multiple tables.

# Order of Execution in SQL Queries

SQL queries are executed in a specific order, which determines how the various clauses interact and affect the final result. Understanding the order of execution is crucial for writing correct and efficient SQL queries. Here's a detailed breakdown of the SQL query execution process:

## Example Query with Order of Execution:-

Consider the following query:

*SELECT DepartmentName, COUNT(EmployeeID) AS NumberOfEmployees*

*FROM Employees*

*INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID*

*WHERE Salary > 50000*

*GROUP BY DepartmentName*

*HAVING COUNT(EmployeeID) > 10*

*ORDER BY NumberOfEmployees DESC;*

## Execution Order:-

1. **FROM Clause**: Employees and Departments tables are specified, and the INNER JOIN is performed.
2. **ON Clause**: The join condition Employees.DepartmentID = Departments.DepartmentID is applied.
3. **JOIN Clause**: The tables are combined based on the join condition.
4. **WHERE Clause**: Filters rows where Salary > 50000.
5. **GROUP BY Clause**: Groups the results by DepartmentName.
6. **HAVING Clause**: Filters groups having more than 10 employees.
7. **SELECT Clause**: Specifies the columns and aggregate functions to be included in the result.
8. **DISTINCT Clause**: (Not used in this query)
9. **ORDER BY Clause**: Sorts the result set by NumberOfEmployees in descending order.
10. **LIMIT / OFFSET Clause**: (Not used in this query)

# SQL Data Types

In SQL, data types define the kind of data that can be stored in a column of a table. Choosing the appropriate data type is crucial for efficient storage, performance, and ensuring the integrity of the data. Below is a summary of common SQL data types, grouped into categories.

## 1. Numeric Data Types:-

These are used to store numeric values.

* **INT** (or INTEGER):
  + Stores whole numbers without decimals.
  + Example: INT, SMALLINT, TINYINT, BIGINT.
  + **Usage**: Counts, IDs, quantities.
* **DECIMAL(p, s)** (or NUMERIC(p, s)):
  + Stores fixed-point numbers with precision p and scale s.
  + Example: DECIMAL(10, 2) stores a number up to 10 digits long, with 2 digits after the decimal.
  + **Usage**: Currency, financial data.
* **FLOAT** and **REAL**:
  + Stores floating-point numbers (approximate numeric data).
  + Example: FLOAT(7) for single precision, FLOAT(15) for double precision.
  + **Usage**: Scientific calculations, measurements.
* **BIT**:
  + Stores bit values (0 or 1).
  + Example: BIT, BIT(8) stores a sequence of bits.
  + **Usage**: Boolean flags, binary data.

## 2. Character and String Data Types:-

These are used to store text or strings.

* **CHAR(n)** (or CHARACTER(n)):
  + Stores fixed-length strings.
  + Example: CHAR(10) always stores exactly 10 characters, padding with spaces if necessary.
  + **Usage**: Fixed-length codes, abbreviations.
* **VARCHAR(n)** (or CHARACTER VARYING(n)):
  + Stores variable-length strings, up to n characters.
  + Example: VARCHAR(50) stores up to 50 characters.
  + **Usage**: Names, descriptions, addresses.
* **TEXT**:
  + Stores variable-length text with no specified maximum length.
  + Example: TEXT can store large text entries like comments or descriptions.
  + **Usage**: Blog posts, comments, long descriptions.

## 3. Date and Time Data Types:-

These are used to store dates, times, and timestamps.

* **DATE**:
  + Stores dates (year, month, day).
  + Example: DATE stores values like 2024-08-05.
  + **Usage**: Birth dates, hire dates, event dates.
* **TIME**:
  + Stores times (hour, minute, second).
  + Example: TIME stores values like 14:30:00.
  + **Usage**: Appointment times, start/end times.
* **DATETIME**:
  + Stores both date and time.
  + Example: DATETIME stores values like 2024-08-05 14:30:00.
  + **Usage**: Event timestamps, logging time.
* **TIMESTAMP**:
  + Similar to DATETIME, but may include timezone information (varies by database).
  + Example: TIMESTAMP stores values like 2024-08-05 14:30:00.
  + **Usage**: Tracking when records are created or modified.
* **INTERVAL**:
  + Represents a time span, often used in conjunction with date/time arithmetic.
  + Example: INTERVAL '5' DAY adds 5 days to a date.
  + **Usage**: Calculating durations, adding/subtracting time.

## 4. Binary Data Types:-

These are used to store binary data, such as images, files, or encrypted data.

* **BINARY(n)**:
  + Stores fixed-length binary data.
  + Example: BINARY(16) stores exactly 16 bytes.
  + **Usage**: Binary keys, encoded data.
* **VARBINARY(n)**:
  + Stores variable-length binary data.
  + Example: VARBINARY(255) stores up to 255 bytes.
  + **Usage**: Image data, file contents.
* **BLOB** (Binary Large Object):
  + Stores large binary objects.
  + Example: BLOB can store multimedia files like images or videos.
  + **Usage**: Images, audio files, large documents.

## 5. Boolean Data Types:-

These are used to store boolean values.

* **BOOLEAN**:
  + Stores TRUE or FALSE.
  + Example: BOOLEAN stores values like TRUE or FALSE.
  + **Usage**: Flags, yes/no fields, toggles.

# Manipulating Tables in SQL

Manipulating tables in SQL involves creating, modifying, and deleting tables, as well as adding, updating, or removing data within them. Below is a summary of the SQL commands commonly used for these operations.

## 1. Creating Tables:-

* **CREATE TABLE**: Used to create a new table in the database.

**Syntax**:

*CREATE TABLE table\_name (*

*column1 datatype constraint,*

*column2 datatype constraint,*

*...*

*);*

**Example**:

*CREATE TABLE Employees (*

*EmployeeID INT PRIMARY KEY,*

*FirstName VARCHAR(50),*

*LastName VARCHAR(50),*

*DepartmentID INT,*

*Salary DECIMAL(10, 2)*

*);*

## 2. Altering Tables:-

* **ALTER TABLE**: Used to modify an existing table structure by adding, modifying, or dropping columns and constraints.

**Adding a Column**:

*ALTER TABLE table\_name*

*ADD column\_name datatype;*

**Example**:

*ALTER TABLE Employees*

*ADD DateOfBirth DATE;*

**Modifying a Column**:

*ALTER TABLE table\_name*

*MODIFY COLUMN column\_name new\_datatype;*

**Example**:

*ALTER TABLE Employees*

*MODIFY COLUMN Salary DECIMAL(12, 2);*

**Dropping a Column**:

*ALTER TABLE table\_name*

*DROP COLUMN column\_name;*

**Example**:

*ALTER TABLE Employees*

*DROP COLUMN DateOfBirth;*

**Adding a Constraint**:

*ALTER TABLE table\_name*

*ADD CONSTRAINT constraint\_name constraint\_type(column\_name);*

**Example**:

*ALTER TABLE Employees*

*ADD CONSTRAINT FK\_Department FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID);*

**Dropping a Constraint**:

*ALTER TABLE table\_name*

*DROP CONSTRAINT constraint\_name;*

**Example**:

*ALTER TABLE Employees*

*DROP CONSTRAINT FK\_Department;*

## 3. Dropping Tables:-

* **DROP TABLE**: Deletes an entire table and all its data from the database.

**Syntax**:

*DROP TABLE table\_name;*

**Example**:

*DROP TABLE Employees;*

## 4. Renaming Tables:-

* **RENAME TABLE**: Renames an existing table.

**Syntax**:

*RENAME TABLE old\_table\_name TO new\_table\_name;*

**Example**:

*RENAME TABLE Employees TO Staff;*

## 5. Inserting Data into Tables:-

* **INSERT INTO**: Adds new rows of data into a table.

**Syntax**:

*INSERT INTO table\_name (column1, column2, ...)*

*VALUES (value1, value2, ...);*

**Example**:

*INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, Salary)*

*VALUES (1, 'John', 'Doe', 2, 50000.00);*

**Inserting Multiple Rows**:

*INSERT INTO table\_name (column1, column2, ...)*

*VALUES*

*(value1, value2, ...),*

*(value1, value2, ...),*

*...;*

**Example**:

*INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, Salary)*

*VALUES*

*(2, 'Jane', 'Smith', 3, 60000.00),*

*(3, 'Emily', 'Johnson', 2, 55000.00);*

## 6. Updating Data in Tables:-

* **UPDATE**: Modifies existing data within a table.

**Syntax**:

*UPDATE table\_name*

*SET column1 = value1, column2 = value2, ...*

*WHERE condition;*

**Example**:

*UPDATE Employees*

*SET Salary = 62000.00*

*WHERE EmployeeID = 1;*

**Updating Multiple Columns**:

*UPDATE Employees*

*SET Salary = 62000.00, DepartmentID = 3*

*WHERE EmployeeID = 1;*

**Important**: Always use a WHERE clause to avoid updating all rows in the table.

## 7. Deleting Data from Tables:-

* **DELETE FROM**: Removes rows from a table based on a condition.

**Syntax**:

*DELETE FROM table\_name*

*WHERE condition;*

**Example**:

*DELETE FROM Employees*

*WHERE EmployeeID = 3;*

**Deleting All Rows**:

*DELETE FROM table\_name;*

**Example**:

*DELETE FROM Employees*;

**Important**: If you omit the WHERE clause, all rows in the table will be deleted.

## 8. Truncating Tables:-

* **TRUNCATE TABLE**: Quickly removes all rows from a table, but keeps the table structure intact.

**Syntax**:

*TRUNCATE TABLE table\_name;*

**Example**:

*TRUNCATE TABLE Employees;*

**Difference from DELETE**: TRUNCATE is faster and resets any auto-increment counters. It cannot be rolled back if used within a transaction.

# SQL Date Functions

SQL provides a variety of built-in functions to manipulate and retrieve information from date and time data types. These functions are crucial for tasks involving date calculations, formatting, and extraction of specific parts of dates. Below is an overview of commonly used SQL date functions.

## 1. ::date:-

* **Purpose**: Casts a timestamp or datetime value to a date, effectively removing the time portion.
* **Syntax**:

*SELECT timestamp\_column::date;*

* **Example**:

*SELECT '2024-08-05 14:30:00'::date;*

* + **Result**: 2024-08-05 (only the date part is returned).

## 2. AT TIME ZONE:-

* **Purpose**: Converts a timestamp from one time zone to another. This function is useful for handling date and time values across different time zones.
* **Syntax**:

*SELECT timestamp\_column AT TIME ZONE 'source\_timezone' AT TIME ZONE 'target\_timezone';*

* **Example**:

*SELECT '2024-08-05 14:30:00' AT TIME ZONE 'UTC' AT TIME ZONE 'America/New\_York';*

* + **Result**: Converts the UTC time to Eastern Time (considering any daylight saving adjustments).

## 3. EXTRACT():-

* **Purpose**: Extracts a specific part (such as year, month, day, etc.) from a date or timestamp.
* **Syntax**:

*SELECT EXTRACT(part FROM date\_or\_timestamp);*

* **Example**:

*SELECT EXTRACT(YEAR FROM '2024-08-05 14:30:00');*

* + **Result**: 2024 (extracts the year from the given timestamp).

*SELECT EXTRACT(DAY FROM '2024-08-05 14:30:00');*

* + **Result**: 5 (extracts the day of the month).

# SQL CASE Expression

The CASE expression in SQL is a powerful tool used to perform conditional logic within queries. It allows you to create different outcomes based on specific conditions, similar to IF-THEN-ELSE statements in programming.

**Syntax**

There are two forms of the CASE expression: **simple** and **searched**.

## 1. Simple CASE Expression:-

This form compares an expression to a set of simple expressions to determine the result.

**Syntax**:

*CASE expression*

*WHEN value1 THEN result1*

*WHEN value2 THEN result2*

*...*

*ELSE default\_result*

*END*

* **expression**: The value you want to compare.
* **WHEN value THEN result**: Specifies a condition and the result if the condition is true.
* **ELSE default\_result**: Specifies a default result if no conditions are met.
* **END**: Ends the CASE expression.

**Example**:

*SELECT*

*EmployeeID,*

*Salary,*

*CASE DepartmentID*

*WHEN 1 THEN 'Sales'*

*WHEN 2 THEN 'Marketing'*

*WHEN 3 THEN 'IT'*

*ELSE 'Unknown'*

*END AS DepartmentName*

*FROM Employees;*

* **Explanation**: This query converts DepartmentID into a department name.

**2. Searched CASE Expression**

This form allows for more complex conditions using boolean logic.

**Syntax**:

*CASE*

*WHEN condition1 THEN result1*

*WHEN condition2 THEN result2*

*...*

*ELSE default\_result*

*END*

* **WHEN condition THEN result**: Specifies a condition to evaluate.
* **ELSE default\_result**: Specifies a result if no conditions are met.
* **END**: Ends the CASE expression.

**Example**:

*SELECT*

*EmployeeID,*

*Salary,*

*CASE*

*WHEN Salary > 100000 THEN 'High'*

*WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'*

*ELSE 'Low'*

*END AS SalaryLevel*

*FROM Employees;*

* **Explanation**: This query categorizes employees' salaries into 'High', 'Medium', or 'Low' based on their salary amount.

# SQL Subqueries and Common Table Expressions (CTEs)

Both subqueries and Common Table Expressions (CTEs) are techniques used in SQL to simplify complex queries, break them down into manageable parts, or reuse intermediate results.

## 1. Subqueries:-

A subquery, also known as an inner query or nested query, is a query nested inside another SQL query. Subqueries can be used in various places within a SQL statement, such as in the SELECT, FROM, WHERE, or HAVING clauses.

**Types of Subqueries**

1. **Single-Row Subquery**:
   * Returns a single value.
   * Typically used in WHERE or HAVING clauses.

**Example**:

*SELECT EmployeeID, Salary*

*FROM Employees*

*WHERE Salary > (SELECT AVG(Salary) FROM Employees);*

* + **Explanation**: The subquery calculates the average salary, and the main query selects employees with a salary greater than this average.

1. **Multi-Row Subquery**:
   * Returns multiple values or rows.
   * Often used with IN, ANY, ALL, or EXISTS operators.

**Example**:

*SELECT EmployeeID, FirstName*

*FROM Employees*

*WHERE DepartmentID IN (SELECT DepartmentID FROM Departments WHERE Location = 'New York');*

* + **Explanation**: The subquery returns all DepartmentIDs for New York, and the main query selects employees in those departments.

1. **Correlated Subquery**:
   * Depends on the outer query for its values.
   * Evaluated once for each row processed by the outer query.

**Example**:

*SELECT e1.EmployeeID, e1.Salary*

*FROM Employees e1*

*WHERE e1.Salary > (SELECT AVG(e2.Salary) FROM Employees e2 WHERE e2.DepartmentID = e1.DepartmentID);*

* + **Explanation**: This query finds employees whose salary is above the average salary in their department.

**Subquery in FROM Clause**

* Also known as a derived table, a subquery in the FROM clause treats the result set of the subquery as a temporary table.

**Example**:

*SELECT dept.DepartmentName, avg\_salaries.AvgSalary*

*FROM Departments dept*

*JOIN (*

*SELECT DepartmentID, AVG(Salary) AS AvgSalary*

*FROM Employees*

*GROUP BY DepartmentID*

*) avg\_salaries ON dept.DepartmentID = avg\_salaries.DepartmentID;*

* **Explanation**: The subquery calculates the average salary by department, and the main query joins this result with the Departments table.

## 2. Common Table Expressions (CTEs):-

A Common Table Expression (CTE) is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs are often used to improve query readability and organization, especially for complex queries.

**Syntax of CTE**

*WITH cte\_name AS (*

*-- Your SQL query here*

*)*

*SELECT \* FROM cte\_name;*

**Example**:

*WITH AvgSalaries AS (*

*SELECT DepartmentID, AVG(Salary) AS AvgSalary*

*FROM Employees*

*GROUP BY DepartmentID*

*)*

*SELECT dept.DepartmentName, AvgSalaries.AvgSalary*

*FROM Departments dept*

*JOIN AvgSalaries ON dept.DepartmentID = AvgSalaries.DepartmentID;*

* **Explanation**: The CTE AvgSalaries calculates average salaries per department, and then the main query joins this result with the Departments table.

**Recursive CTEs**

* CTEs can be recursive, which means they can refer to themselves to produce iterative results, often used for hierarchical data like organizational structures or tree traversal.

**Example**:

*WITH RECURSIVE EmployeeHierarchy AS (*

*SELECT EmployeeID, ManagerID, 1 AS Level*

*FROM Employees*

*WHERE ManagerID IS NULL*

*UNION ALL*

*SELECT e.EmployeeID, e.ManagerID, eh.Level + 1*

*FROM Employees e*

*JOIN EmployeeHierarchy eh ON e.ManagerID = eh.EmployeeID*

*)*

*SELECT \* FROM EmployeeHierarchy;*

* **Explanation**: This query builds a hierarchical list of employees and their levels in the organization.

**Summary**

* **Subqueries**: Nested queries that can be used to filter, aggregate, or compare data in various clauses of a SQL statement. Useful for breaking down complex queries into simpler parts.
* **CTEs**: Named temporary result sets that improve readability and manageability of complex queries. CTEs can be recursive, which is useful for working with hierarchical data.

Both techniques help in organizing and structuring SQL queries to make them easier to read, write, and debug.

# SQL UNION Operator

The UNION operator in SQL is used to combine the result sets of two or more SELECT queries into a single result set. It removes duplicate records by default, returning only distinct values.

**Syntax**

*SELECT column1, column2, ...*

*FROM table1*

*UNION*

*SELECT column1, column2, ...*

*FROM table2;*

**Example Usage**

**Example**:

*SELECT FirstName, LastName FROM Employees*

*UNION*

*SELECT FirstName, LastName FROM Customers;*

* **Explanation**: This query combines the first and last names from the Employees and Customers tables into a single result set, removing duplicates.

**UNION ALL**

* **Purpose**: The UNION ALL operator is similar to UNION, but it does not remove duplicate rows. It returns all rows from the combined result set, including duplicates.

**Example**:

*SELECT FirstName, LastName FROM Employees*

*UNION ALL*

*SELECT FirstName, LastName FROM Customers;*

* **Explanation**: This query combines the first and last names from both tables and includes all duplicates.

**ORDER BY with UNION**

* **Usage**: To sort the combined result set, you can use the ORDER BY clause at the end of the last SELECT statement.

**Example**:

*SELECT FirstName, LastName FROM Employees*

*UNION*

*SELECT FirstName, LastName FROM Customers*

*ORDER BY LastName, FirstName;*

* **Explanation**: This query combines the results and sorts them by last name and then by first name.

**Summary**

* **UNION** combines results from multiple SELECT statements into one, removing duplicates.
* **UNION ALL** does the same but keeps all duplicate records.
* Ensure that the number of columns and data types align in each query used with UNION.
* Use ORDER BY at the end to sort the final combined result set.