**CTE - (COMMON TABLE EXPRESSION)**

**Common Table Expressions (CTE)** in SQL are a powerful feature that allows you to create temporary result sets which can be referred to within a SELECT, INSERT, UPDATE, DELETE, or MERGE statement. CTEs are particularly useful for simplifying complex queries by breaking them down into simpler, more readable parts.

**Syntax and Usage of CTEs**

A CTE is defined using the WITH clause followed by the CTE's name and an optional list of column names. It includes a query, often a SELECT statement, that defines the CTE's result set. Here's the basic syntax for a CTE:

WITH cte\_name (column\_name [, ...]) AS (

SELECT

)

The CTE can then be used in the main query as if it were a regular table or view. For example:

WITH cteExample AS (

SELECT column1, column2

FROM table1

WHERE condition

)

SELECT \*

FROM cteExample

**Recursive CTEs:**

Recursive CTEs are a special type of CTE that can reference themselves, making them ideal for dealing with hierarchical data structures like organisational charts or category trees. They consist of two parts: an anchor member that initialises the recursion, and a recursive member that references the CTE itself. The recursive process continues until no more rows are returned.

**EXAMPLE:**

*WITH RECURSIVE factorial(n, fact) AS (*

*SELECT 1, 1 -- Initial value (n=1, fact=1)*

*UNION ALL*

*SELECT n + 1, fact \* (n + 1)*

*FROM factorial*

*WHERE n < 5 -- Calculate factorial up to 5*

*)*

*SELECT \* FROM factorial;*

**OUTPUT:**

| *n* | *fact\*(n+1)* |
| --- | --- |
| *1* | *1* |
| *2* | *2* |
| *3* | *6* |
| *4* | *24* |
| *5* | *120* |

**UNION:**

In SQL, the UNION operator is used to combine the result sets of two or more SELECT statements. It is used when you want to retrieve results from multiple queries into a single output while removing duplicate rows.

**Syntax:**

*SELECT column1, column2, …*

*FROM table1*

*UNION*

*SELECT column1, column2, …*

*FROM table2;*

**Example**

Consider two tables, Employees and Contractors, with the following structure:

Table: Employees

| id | name |
| --- | --- |
| 1 | john |
| 2 | sarah |

Table: Contractors

| id | name |
| --- | --- |
| 3 | mike |
| 2 | sarah |

To create a list of unique names from both tables:

*SELECT name FROM Employees*

*UNION*

*SELECT name FROM Contractors;*

Result:

| name |
| --- |
| john |
| sarah |
| mike |

Note that even though "Sarah" appears in both tables, it is listed only once in the output because UNION removes duplicates.

**UNION ALL:**

If you want to include all duplicates, you can use UNION ALL.

*SELECT name FROM Employees*

*UNION ALL*

*SELECT name FROM Contractors;*

Result:

| name |
| --- |
| john |
| sarah |
| mike |
| sarah |

**FLOW OF EXECUTION IN SQL:**

While SQL queries are generally written in a human-readable order (SELECT, FROM, WHERE, etc.), the actual order of execution is different.

It affects how data is filtered, grouped, and ultimately returned.

FROM Clause

|

WHERE Clause

|

GROUP BY Clause

|

HAVING Clause

|

SELECT Clause

|

ORDER BY Clause

|

LIMIT/ OFFSET Clause

**FROM Clause (Including JOINS):**

The first step is the FROM clause, where the SQL engine identifies the source table(s) or view(s) from which data will be retrieved. If there are multiple tables, joins are evaluated during this step to create a base data set.

**WHERE Clause:**

The WHERE clause is used to filter rows based on specified conditions. This clause narrows down the data set by eliminating rows that do not meet the criteria.

**GROUP BY Clause:**

After filtering, the GROUP BY clause is applied. It groups the resulting rows based on one or more columns. This is often used in conjunction with aggregate functions like `COUNT()`, `SUM()`, `AVG()`, etc.

**HAVING Clause:**

The HAVING clause is similar to the WHERE clause but is applied after grouping. It filters out groups that do not meet the specified conditions.

**SELECT Clause:**

The SELECT clause specifies the columns to be retrieved from the final filtered and grouped result set. This step can include calculations, transformations, or the use of alias names for columns.

**ORDER BY Clause:**

The ORDER BY clause sorts the final result set based on one or more columns. The sorting can be done in ascending (ASC) or descending (DESC) order.

**LIMIT / OFFSET Clause:**

The LIMIT clause (or TOP in some databases) is used to restrict the number of rows returned by the query, and OFFSET is used to skip a specified number of rows.

**EXAMPLE:**

*SELECT department, AVG(salary)*

*FROM employees*

*WHERE age > 30*

*GROUP BY department*

*HAVING AVG(salary) > 50000*

*ORDER BY AVG(salary) DESC;*

The flow of execution is as follows:

1. FROM: Identify employees as the source table.

2. WHERE: Filter rows where age > 30.

3. GROUP BY: Group rows by department.

4. HAVING: Filter groups where the average salary is greater than 50,000.

5. SELECT: Select the department and the calculated AVG(salary).

6. ORDER BY: Sort the result based on AVG(salary) in descending order.