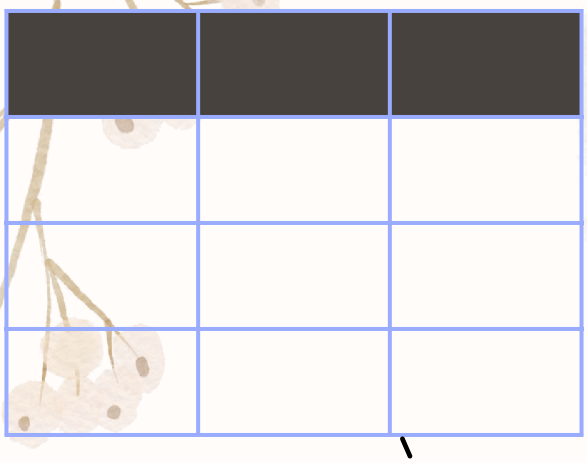
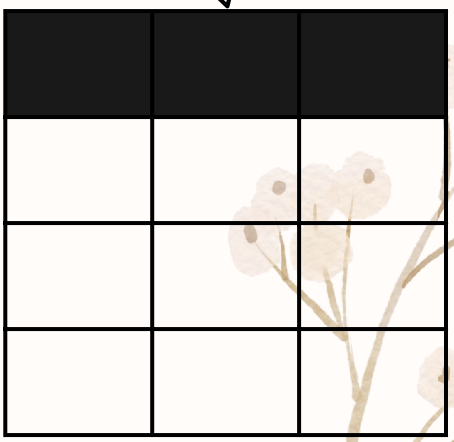
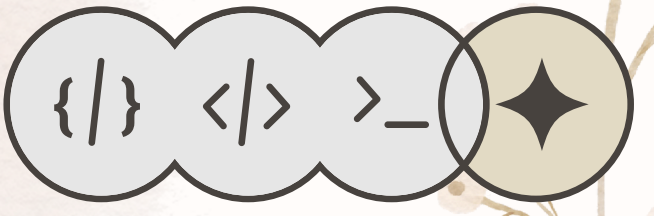


CTES IN SQL





COMMON TABLE EXPRESSION - CTE:

Consider a scenario:

Suppose, you're organizing a birthday party.

- **Step 01**: You first make a guest list of everyone you want to invite.
- **Step 02**: Then, you use that list to plan things like seating arrangements or food orders.

The guest list isn't a permanent document—it's just a temporary tool to help you plan your party efficiently.

In SQL, a CTE is like that temporary guest list. It helps you prepare data that you can use later in the same query.

Formal Definition:

A CTE (Common Table Expression) in SQL is like a temporary table or a shortcut that you can create and use within the same query.
It helps you organize and simplify your SQL code, especially when working with complex queries.

Syntax:

```
WITH CTE_Name AS (
    -- Your query goes here
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition
)
-- Main query referencing the CTE
SELECT *
FROM CTE_Name
WHERE another_condition;
```

- **WITH Clause**: Declares the start of the CTE.
- **CTE_Name**: A user-defined name for the CTE, used to reference it in subsequent queries.
- **Inner Query**: The query inside parentheses defines the temporary result set.
- **Main Query**: The outer query uses the CTE as if it were a table.

Key Points of CTE:

1. **Temporary Scope:** A CTE exists only during the execution of the query in which it is defined.
2. **Defined with WITH Clause:** Starts with the keyword WITH followed by a name and a query inside parentheses.
3. **Reusable Within Query:** You can reference the CTE multiple times in the same query.
4. **Supports Recursion:** Recursive CTEs are used for tasks like traversing hierarchical data (e.g., org charts, file directories).
5. **Cannot Be Indexed:** CTEs are not stored on disk or indexed—they are executed on the fly.
6. **Simplifies Joins and Aggregations:** Makes handling joins, aggregations, and derived data much cleaner.
7. **No Materialization:** Unlike views or temporary tables, CTEs do not persist or store intermediate results; they are virtual and exist only during query execution.
8. **Can Use Multiple CTEs:** You can define and chain multiple CTEs in a single query.
   Better Alternative to Subqueries:

Example 01:

Suppose,

- You have data stored in table '`Employees`'.
- You have to analyze employees who have a salary above the department's average salary.

Input Table: Employees

EmployeeID	FirstName	LastName	DepartmentID	Salary
1	Alice	Johnson	101	70000
2	Bob	Smith	101	80000
3	Charlie	Brown	102	90000
4	Diana	Lee	102	75000
5	Edward	Wilson	103	60000
6	Fiona	Carter	103	65000

Query Using CTE:

```
WITH DepartmentAverage AS (
    SELECT DepartmentID,AVG(Salary) AS AvgSalary
    FROM Employees
    GROUP BY DepartmentID
)
SELECT
e.EmployeeID,e.FirstName,e.LastName,e.DepartmentID,e.Salary,d.AvgSalary
FROM Employees e
INNER JOIN DepartmentAverage d
ON e.DepartmentID = d.DepartmentID
WHERE e.Salary > d.AvgSalary;
```

CTE Breakdown:

1. **DepartmentAverage CTE**:
    - Computes the average salary for each department.
    - Result of DepartmentAverage:

DepartmentID	AvgSalary
101	75000
102	82500
103	62500

1. **Main Query:**
    - Joins the Employees table with the DepartmentAverage CTE.
    - Filters employees earning more than their department's average salary.

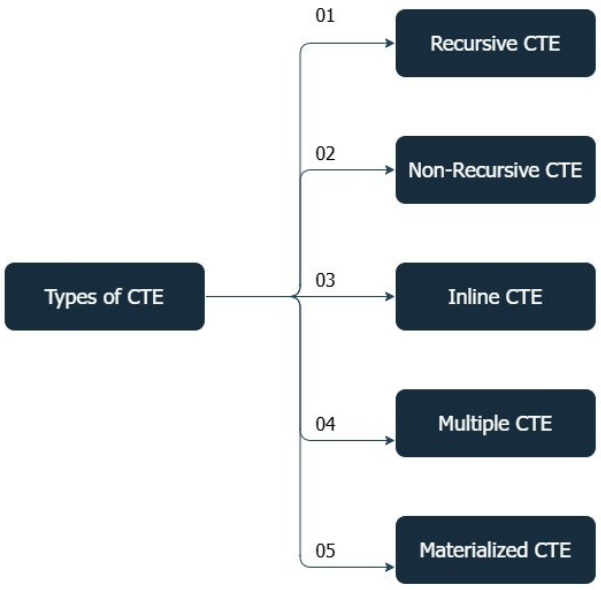
EmployeeID	FirstName	LastName	DepartmentID	Salary	AvgSalary
2	Bob	Smith	101	80000	75000
3	Charlie	Brown	102	90000	82500

Explanation:

- Bob (Department 101) earns 80,000, which is above the average of 75,000 for his department.
- Charlie (Department 102) earns 90,000, which is above the average of 82,500 for his department.

TYPES OF CTEs in SQL:

In SQL, Common Table Expressions (CTEs) can be broadly categorized into the following types based on their usage and characteristics:



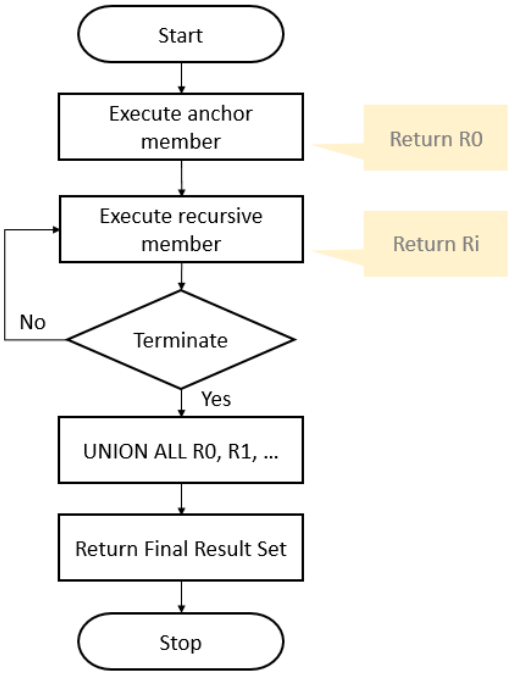
- **Recursive CTE**: For hierarchical or iterative data.
- **Non-Recursive CTE**: Simplifies complex queries.
- **Inline CTE:** Acts as an inline view for modular query design.
- **Multiple CTEs:** Multiple result sets in a single query.
- **Materialized CTE:** Performance optimization (SQL Server 2022+).

01. Recursive CTE:

A Recursive CTE (Common Table Expression) is a powerful SQL feature used to handle hierarchical data or repetitive tasks.

It allows a query to refer to itself, enabling you to perform operations like,

- Tree traversals
- Generating sequential data
- Processing hierarchical relationships



How Recursive CTE Works:

A recursive CTE consists of two parts:

- **Base case:** The initial query that forms the starting point of recursion. This query is executed once.
- **Recursive case:** The query that references the CTE itself, repeatedly calling the CTE to fetch results until a termination condition is met.

The termination condition ensures that the recursion stops.

If the termination condition is not defined, it can lead to an infinite loop.

Syntax:

```
WITH CTEName AS (

    -- Base case: first query
    SELECT <columns>
    FROM <table_name>
    WHERE <conditions>

    UNION ALL

    -- Recursive case: query that references the CTE itself
    SELECT <columns>
    FROM <table_name>
    INNER JOIN CTEName ON <conditions>
)
SELECT * FROM CTEName;
```

Example: Finding Employee Hierarchy (Manager-Subordinate Relationship)

Let's consider a scenario where we have a table representing an organizational hierarchy. Each employee has a manager, and we want to find the hierarchy of employees starting from the CEO (who has no manager) and traverse down to all subordinates.

EmployeeID	EmployeeName	ManagerID
1	John	NULL
2	Alice	1
3	Bob	1
4	Charlie	2
5	David	3
6	Eve	3

We want to list the employees in a hierarchical order starting from top-level managers.

Recursive CTE Query:

We'll use a **recursive CTE** to:

1. **Base condition**: Start with the CEO (ManagerID is NULL).
2. **Recursive step**: Find employees who report to the employees in the previous level.
3. **Termination condition**: Automatically stops when no employees are left to process.

```
WITH EmployeeHierarchy AS (

    -- Base case: Select top-level managers (those with no manager)
    SELECT EmployeeID, EmployeeName, ManagerID, 0 AS Level
    FROM Employees
    WHERE ManagerID IS NULL

    UNION ALL

    -- Recursive case: Select employees under managers
    SELECT e.EmployeeID, e.EmployeeName, e.ManagerID, eh.Level + 1 AS Level
    FROM Employees e
    INNER JOIN EmployeeHierarchy eh ON e.ManagerID = eh.EmployeeID
)
SELECT * FROM EmployeeHierarchy
ORDER BY Level, EmployeeID;
```

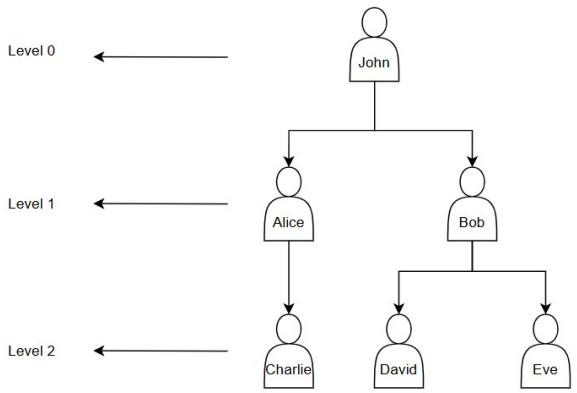
EmployeeID	EmployeeName	ManagerID	Level
1	John	NULL	0
2	Alice	1	1
3	Bob	1	1
4	Charlie	2	2
5	David	3	2
6	Eve	3	2

Explanation:

1. **Base Case**:
    - Starts with the CEO (ManagerID IS NULL).
    - Includes their EmployeeID, Name, ManagerID, and sets Level = 0.
2. **Recursive Step**:
    - For each employee in the current level, finds their subordinates by matching EmployeeID with the ManagerID in the Employees table.
    - Increments the level by 1 for subordinates.
3. **Termination**:
    - Stops automatically when no more employees are found for the next level.

- The base case selects John as the top-level manager (Level 0).
- The recursive case selects employees Alice and Bob (Level 1) under John.
- Further recursion finds employees under Alice and Bob at Level 2.

Level 0: John (CEO)
Level 1: Alice, Bob
Level 2: David, Charlie, Eve



02. Non-Recursive CTE:

A Non-Recursive CTE (Common Table Expression) is a query that does not reference itself. Unlike Recursive CTEs, which are used to handle hierarchical or repetitive data, Non-Recursive CTEs are just temporary result sets that you can use to simplify complex queries, enhance readability, and avoid writing subqueries repeatedly in your SQL statements.

How Non-Recursive CTE Works:

1. It is defined with a WITH clause.
2. The CTE contains a single query that is executed once.
3. The result of the query is stored temporarily and can be referred to later in the main query.

Non-Recursive CTEs can be very useful for breaking down complex queries into smaller, manageable pieces.

Syntax:

```
WITH CTE_Name AS (
    SELECT <columns>
    FROM <table_name>
    WHERE <conditions>
)
SELECT <columns>
FROM CTE_Name;
```

Example 1: Simplifying a Query with Non-Recursive CTE

Assume we have a Sales table, and we want to calculate the total sales for a particular year (e.g., 2024), and then filter it based on the highest sales amount.

SaleID	SaleAmount	SaleDate
1	100	2024-01-01
2	200	2024-02-01
3	150	2024-03-01
4	250	2024-01-15
5	300	2024-03-10
6	200	2023-03-10
7	100	2023-02-10
8	350	2023-01-10

Non-Recursive CTE Query:

```
WITH TotalSales AS (
    SELECT SUM(SaleAmount) AS TotalAmount
    FROM Sales
    WHERE YEAR(SaleDate) = 2024
)
SELECT *
FROM TotalSales
WHERE TotalAmount > 500;
```

Explanation:

- The CTE named TotalSales calculates the total sales amount for the year 2024.
- The main query then selects from TotalSales and filters the result to only include totals greater than 500.
- The result would show that the total sales for 2024 exceed 500.

Output:

TotalAmount
1000

03. Inline CTE:

Inline CTEs (Common Table Expressions) are a way to define a temporary result set directly within a SQL query for immediate use. They are referred to as "inline" because they are defined and used directly within the scope of a single query, allowing for concise, readable SQL code without creating a permanent table or repeating subquery logic.

Key Characteristics of Inline CTEs:

Temporary Scope: They only exist within the scope of the query they are part of.

Defined and Used in a Single Query: The WITH clause is defined once, and the CTE is used immediately in the subsequent query.

Improved Readability: Eliminates the need for nested subqueries, making complex queries more readable.

No Stored Results: Results from a CTE are not stored; they are computed on the fly during execution.

Syntax:

```
WITH CTE_Name AS (
    SELECT <columns>
    FROM <table_name>
    WHERE <conditions>
)
SELECT <columns>
FROM CTE_Name
WHERE <additional_conditions>;
```

Example 1: Filtering Data with Inline CTE

Assume we have a Sales table:

SaleID	CustomerID	SaleAmount	SaleDate
1	101	500	2023-12-01
2	102	300	2023-12-05
3	103	700	2023-12-07
4	101	200	2023-12-10
5	102	400	2023-12-12

Query:

Let's say we want to calculate the total sales amount per customer and then filter out customers whose total sales are greater than 600.

```
WITH CustomerSales AS (
    SELECT CustomerID, SUM(SaleAmount) AS TotalSales
    FROM Sales
    GROUP BY CustomerID
)
SELECT CustomerID, TotalSales
FROM CustomerSales
WHERE TotalSales >= 600;
```

Explanation:

1. The **CTE** named CustomerSales calculates the total sales for each customer.
2. The **main query** filters the results to show only customers with total sales >= 600.

CustomerID	TotalSales
101	700
103	700

Inline CTE vs Non-Recursive CTE:

Aspect	Inline CTE	Non-Recursive CTE
Definition	A single-use query for simplification.	Used for multi-step data processing.
Complexity	Generally simple and straightforward.	Can involve multiple layers of processing.
Use Case	Replace subqueries, simplify SQL.	Structure multi-step calculations.
Recursion	Not recursive.	Not recursive but involves layered logic.
Output	Produces a simplified query result.	Can generate intermediate and final results.

04. Multiple CTE:

Multiple CTEs are when you define and use more than one Common Table Expression (CTE) in a single query. They are particularly useful for breaking down complex queries into smaller, manageable pieces, and for reusing intermediate results across different parts of the query.

Key Features of Multiple CTEs:

Sequential Definitions: CTEs are defined one after another, separated by commas.

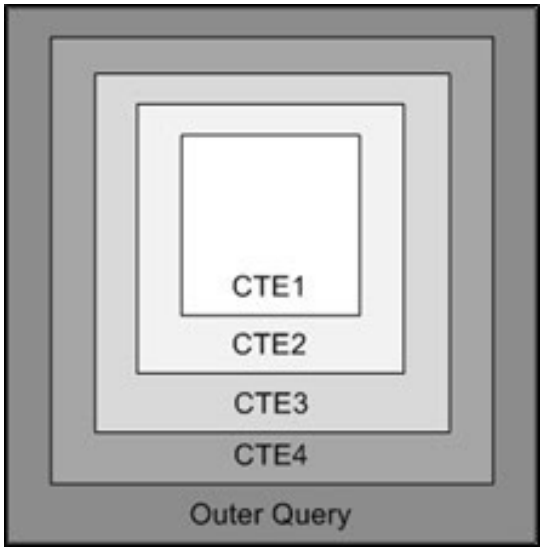
Dependent or Independent: Subsequent CTEs can reference previous ones, or they can operate independently.

Enhanced Readability: They make complex queries more modular and readable.

Single Query Scope: All CTEs exist within the context of a single SQL query.

Syntax for Multiple CTEs:

```
WITH CTE1 AS (
  SELECT <columns>
  FROM <table_name>
  WHERE <conditions>
),
CTE2 AS (
  SELECT <columns>
  FROM CTE1
  WHERE <conditions>
),
CTE3 AS (
  SELECT <columns>
  FROM <table_name>
  WHERE <conditions>
)
SELECT <columns>
FROM CTE2
JOIN CTE3 ON <condition>;
```



Example 1: Aggregating Data Across Multiple CTEs

Table: `Orders`

OrderID	CustomerID	OrderDate	OrderAmount
1	101	2023-01-01	500
2	102	2023-01-02	300
3	103	2023-01-03	700
4	101	2023-01-04	200
5	102	2023-01-05	400

Query:

Let's calculate:

1. Total order amount for each customer (using CTE1).
2. Customers with total order amounts greater than 600 (using CTE2).

```
WITH CTE1 AS (
    SELECT CustomerID, SUM(OrderAmount) AS TotalOrderAmount
    FROM Orders
    GROUP BY CustomerID
),
CTE2 AS (
    SELECT CustomerID, TotalOrderAmount
    FROM CTE1
    WHERE TotalOrderAmount > 600
)
SELECT CustomerID, TotalOrderAmount
FROM CTE2;
```

Result:

CustomerID	TotalOrderAmount
101	700
103	700

05. Materialized CTE:

Materialized CTEs are a concept introduced in some database systems (e.g., Snowflake, PostgreSQL starting with version 12) to improve query performance. When a CTE is materialized, the results of the CTE are computed once and temporarily stored (similar to a temporary table) for reuse in subsequent parts of the query. This avoids recomputation of the CTE when it is referenced multiple times.

Key Characteristics:

Reusability: The result of a materialized CTE is cached, so it is not recalculated every time it's used.

Performance Boost: Materialization can significantly improve performance in complex queries where the same CTE is referenced multiple times.

Explicit Materialization: Some databases allow you to explicitly request materialization, while others decide automatically.

Syntax in PostgreSQL

In PostgreSQL, materialization of a CTE can be enforced using the MATERIALIZED keyword. Conversely, the NOT MATERIALIZED keyword forces inlining (where the CTE is recalculated every time it's used).

```
WITH [MATERIALIZED | NOT MATERIALIZED] CTE_Name AS (
    SELECT <columns>
    FROM <table>
    WHERE <conditions>
)
SELECT *
FROM CTE_Name;
```

Example 1: Materialized CTE

Table: `Sales`

SaleID	ProductID	Quantity	Price
1	101	10	20
2	102	15	30
3	101	20	20
4	103	10	50

Query:

We want to:

- Calculate the total revenue for each product.
- Reuse this calculation to find products with revenue greater than $500 and display their details.

```
WITH MATERIALIZED RevenueCTE AS (
    SELECT ProductID, SUM(Quantity * Price) AS TotalRevenue
    FROM Sales
    GROUP BY ProductID
)
SELECT *
FROM RevenueCTE
WHERE TotalRevenue > 500;
```

Output:

ProductID	TotalRevenue
101	600
103	500

How materialized CTE differ from simple CTE?

Aspect	Simple CTE	Materialized CTE
Evaluation	Recomputed every time it is referenced.	Computed once and stored temporarily.
Performance	May be slower for repeated usage.	Faster when reused multiple times.
Storage	Not stored; purely inline execution.	Stored temporarily in memory or disk.
Syntax	Standard WITH clause.	WITH MATERIALIZED or equivalent (engine-specific).
Support	Supported by most SQL engines.	Limited support (e.g., PostgreSQL, Snowflake).