Advanced CTE

What is CTE?

Is a temporary, named result set in SQL that allows you to simplify complex queries, making them easier to read and maintain. CTEs are commonly used when working with multiple subqueries. You might recognize them because they are created with the distinctive `WITH` keyword and, like I mentioned, they can be used in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.

How to create a CTE?

```
WITH cte_name (column1, column2, ...)

AS (

    -- Query that defines the CTE

    SELECT ...

    FROM ...

    WHERE ...

)

-- Main query

SELECT ...

FROM cte_name;
```

- `WITH`: Initiates the CTE definition, indicating that the following name represents a temporary result set.
- `cte_name`: The name is assigned to the CTE to reference it in the main query.
- Optional column list (column1, column2, ...): Specifies column names for the CTE's result set. This is useful when column names need to be adjusted.
- Query that defines the CTE: The inner query that selects data and shapes the temporary result set.
- Main query: References the CTE by its name, using it like a table.

Advantages of CTE

Simplify complex queries

CTEs break down complex SQL statements into smaller, more manageable parts, making the code easier to read, write, and maintain.

Code reusability

CTEs help avoid duplication by allowing the same result set to be reused across different parts of a query. If multiple calculations or operations are based on the same dataset, you can define it once in a CTE and refer to it as needed.

- **Query Organization and Readability:** CTEs improve SQL code **readability** by dividing queries into **logical**, **sequential** steps. Each step in the query process can be represented by its own CTE, making the entire query easier to follow.
- **Hierarchical Data Traversal:** CTEs can help navigate **hierarchical** relationships, such as **organizational structures**, **parent-child relationships**, or any data model that involves **nested levels.** Recursive CTEs are useful for querying hierarchical data because they allow you to **traverse levels iteratively**.
- **Multi-Level Aggregations:** CTEs can help perform aggregations at multiple levels, such as calculating sales figures at different **granularities** (e.g., by month, quarter, and year). Using CTEs to separate these aggregation steps ensures that each level is calculated independently and logically.
- **Combining Data from Multiple Tables**: Multiple CTEs can be used to **combine data** from different tables, making the final combination step more structured. This approach simplifies complex joins and ensures the source data is organized logically for improved readability.

Multiple CTEs in a single query

```
WITH ProductSales AS (
    -- Step 1: Calculate total sales for each product
    SELECT ProductID, SUM(SalesAmount) AS TotalSales
    FROM Sales
    GROUP BY ProductID
),
AverageSales AS (
    -- Step 2: Calculate the average total sales across all products
    SELECT AVG(TotalSales) AS AverageTotalSales
    FROM ProductSales
),
HighSalesProducts AS (
    -- Step 3: Filter products with above-average total sales
    SELECT ProductID, TotalSales
    FROM ProductSales
    WHERE TotalSales > (SELECT AverageTotalSales FROM AverageSales)
)

-- Step 4: Rank the high-sales products

SELECT ProductID, TotalSales, RANK() OVER (ORDER BY TotalSales DESC) AS SalesRank

FROM HighSalesProducts;
```

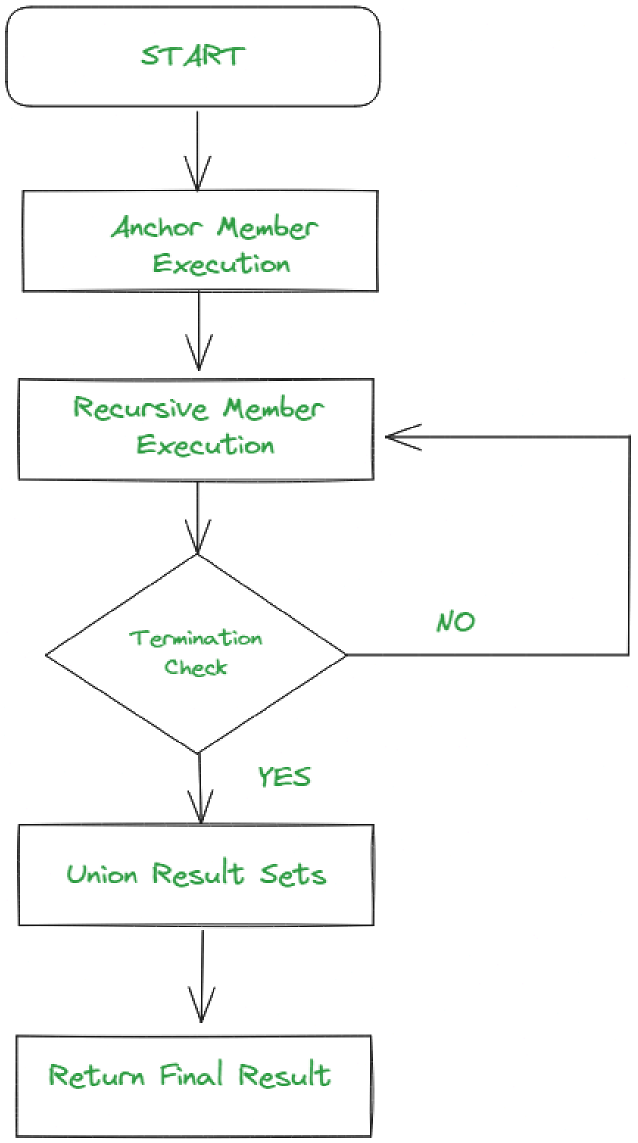
Recursive Common Table Expressions (CTEs)

Recursive CTEs are a special type of CTE that references itself within its definition, allowing the query to perform repeated operations. This makes them ideal for working with hierarchical or tree-structured data, such as organizational charts, directory structures, or product assemblies. The recursive CTE iteratively processes data, returning results step by step until a termination condition is met.

```
WITH RECURSIVE

cte_name [(col1, col2, ...)]

AS ( subquery )

Select col1, col2, .. from cte_name;
```

A recursive CTE consists of two main parts:

- **Anchor Member:** The part that defines the base query that starts the recursion.
- **Recursive Member:** The part that references the CTE itself, allowing it to perform the "recursive" operations.


START
Anchor Member Execution
Recursive Member Execution
Termination Check
NO
YES
Union Result Sets
Return Final Result

1. **WITH EmployeeHierarchy AS (...):** This defines a CTE named `EmployeeHierarchy` to organize employees based on their reporting structure.
2. **Anchor member:**
   - `SELECT EmployeeID, EmployeeName, ManagerID, 1 AS Level FROM Employees WHERE EmployeeID = 1`: This part selects the top-level manager (assumed to have `EmployeeID = 1`) and sets their hierarchy level to 1.
3. **UNION ALL:**
   - Combines the anchor member with the recursive member to build the hierarchy.
4. **Recursive member:**
   - `SELECT e.EmployeeID, e.EmployeeName, e.ManagerID, eh.Level + 1 FROM Employees e INNER JOIN EmployeeHierarchy eh ON e.ManagerID = eh.EmployeeID`: This part recursively finds employees who report to the current managers, incrementing the hierarchy level by 1 for each level down.
5. **Final SELECT:**
   - `SELECT EmployeeID, EmployeeName, Level FROM EmployeeHierarchy`: This retrieves the hierarchical structure of employees, showing each employee's ID, name, and their level in the hierarchy.

The code aims to create a hierarchical view of employees starting from the top-level manager, showing how each employee fits into the organizational structure.

```
WITH Recursive RecursiveOrganizationCTE AS

(

   SELECT EmployeeID, FirstName, LastName, Department,ManagerID

   FROM Organization

   WHERE ManagerID IS NULL

   UNION ALL



   SELECT e.EmployeeID, e.FirstName, e.LastName, e.department,e.ManagerID

   FROM Organization e

   JOIN RecursiveOrganizationCTE r ON e.ManagerID = r.EmployeeID

)

--Show the records stored inside  the CTE we created above

SELECT *

FROM RecursiveOrganizationCTE;
```

Although CTEs are useful for simplifying complex queries, there are some common pitfalls you should be aware of. They include the following:

- **Infinite Loops in Recursive CTEs:** If the termination condition for a recursive CTE is not met, it can result in an infinite loop, causing the query to run indefinitely.
- **Performance Considerations:** Recursive CTEs can become resource-intensive if the recursion depth is high or large datasets are being processed. To optimize the performance, limit the data processed in each iteration and ensure appropriate filtering to avoid excessive recursion levels.

When to Use CTEs vs. Other Techniques

While CTEs are appropriate for **simplifying queries** involving r**epeated tasks, derived tables, views,** and **temp tables** also serve similar purposes. The following table highlights the advantages and disadvantages of each method and when to use each.

Technique	Advantages	Disadvantages	Suitable Use Case
CTEs	Temporary scope within a single query. No storage or maintenance required Improves readability by modularizing code	Limited to the query in which they are defined	Organizing complex queries, temporary transformations, and breaking down multi-step operations