Common Table Expressions (CTEs) in SQL

# Agenda

• Introduction to CTEs

• Basic Syntax of a CTE

• Types of CTEs

• Non-Recursive CTEs

• Recursive CTEs

• Multiple CTEs in a Single Query

• Advanced CTE Techniques

• Nested CTEs

• Applying Window Functions in CTEs

• Limitations of CTEs

# Introduction to CTEs

A **Common Table Expression (CTE)** is a temporary, named result set in SQL that exists only for the duration of a single query. Defined using the **WITH** keyword, a CTE acts like a virtual table that can be referenced multiple times within the query.

CTEs are particularly useful for:

* Organizing and simplifying complex queries
* Improving readability and maintainability
* Reducing duplication by avoiding repeated subqueries
* Supporting recursive operations for hierarchical data

By breaking queries into smaller, modular parts, CTEs help produce SQL code that is more efficient, easier to debug, and easier to understand.

## Why use CTEs?

- **Improved Readability –** Breaks complex queries into smaller, more understandable parts.

- **Reusability –** Can be referenced multiple times within the same query.

- **Recursive Queries –** Supports recursion for hierarchical data like org charts.

- **Enhanced Modularity –** Allows step-by-step query building.

- **Temporary Scope –** Acts as a temporary view within a query.

# Basic Syntax of a CTE

The structure of a CTE begins with the WITH keyword, followed by the CTE name, AS, and a query inside parentheses.

|  |
| --- |
| WITH cte\_name AS (  -- Define your query here  SELECT column1, column2, ... FROM table\_name  WHERE condition  )  -- Main query that uses the CTE  SELECT columns FROM cte\_name; |

Example: Return employees who earn over $50,000?

|  |
| --- |
| WITH HighEarningEmployees AS (  SELECT EmployeeID, FirstName, LastName, Salary  FROM Employees  WHERE Salary > 50000 ) SELECT EmployeeID, FirstName, LastName FROM HighEarningEmployees; |

# Types of CTEs

There are two main types of CTEs: Non-Recursive and Recursive.

## Non-Recursive CTEs

Non-Recursive CTEs are the simplest form. They structure queries by storing results temporarily without recursion.

**Example: Return list employees with a salary over $50,000?**

|  |
| --- |
| WITH HighEarningEmployees AS (  SELECT EmployeeID, FirstName, LastName, Salary  FROM Employees  WHERE Salary > 50000 ) SELECT EmployeeID, FirstName, LastName FROM HighEarningEmployees; |

## Recursive CTEs

Recursive CTEs reference themselves, making them useful for hierarchical data such as org charts, file systems, or parent-child structures.

**Example: Return all employees in the reporting hierarchy under a specific manager, say, ManagerID = 1?**

|  |
| --- |
| WITH EmployeeHierarchy AS (  SELECT EmployeeID, ManagerID, FirstName, 1 AS Level  FROM Employees  WHERE ManagerID = 1   UNION ALL   SELECT e.EmployeeID, e.ManagerID, e.FirstName, eh.Level + 1  FROM Employees e  INNER JOIN EmployeeHierarchy eh ON e.ManagerID = eh.EmployeeID )  SELECT EmployeeID, FirstName, Level FROM EmployeeHierarchy; |

# Multiple CTEs in a Single Query

Multiple CTEs can be combined to build structured queries in stages.

**Example: Return Avg Salary in each department while identifying employees with salaries above their departmental avg & finding names of employees along with their Manager Names?**

|  |
| --- |
| WITH DeptAvgSalary AS (  SELECT DepartmentID, AVG(Salary) AS AvgSalary  FROM Employees  GROUP BY DepartmentID ), HighEarners AS (  SELECT e.EmployeeID, e.FirstName, e.Salary, e.DepartmentID, e.ManagerID  FROM Employees e  JOIN DeptAvgSalary d ON e.DepartmentID = d.DepartmentID  WHERE e.Salary > d.AvgSalary ), Managers AS (  SELECT h.EmployeeID, h.FirstName AS EmployeeFirstName, h.Salary,  m.FirstName AS ManagerFirstName  FROM HighEarners h  LEFT JOIN Employees m ON h.ManagerID = m.EmployeeID ) SELECT EmployeeFirstName, Salary, ManagerFirstName FROM Managers; |

**Benefits of Using Multiple CTEs**

1. **Readability**
   * Complex queries can be broken into smaller, logical steps.
   * Each CTE focuses on a specific task, making the SQL easier to read and understand.
2. **Reusability**
   * Once defined, CTEs can be referenced by subsequent CTEs or the final query.
   * Eliminates duplication of logic and ensures consistency across the query.
3. **Maintainability**
   * Changes can be made within the relevant CTE without affecting the entire query.
   * Enhances modularity, making long queries easier to update and debug.

# Advanced CTE Techniques

## Nested CTEs (Using CTEs within CTEs)

Nested CTEs allow multi-step transformations inside each other.

**Example: Calculate the average salary per department, then find employees earning above average, and finally determine the count of such high earners per department?**

|  |
| --- |
| WITH DeptAvgSalary AS (  SELECT DepartmentID, AVG(Salary) AS AvgSalary  FROM Employees  GROUP BY DepartmentID ), HighEarners AS (  SELECT e.EmployeeID, e.FirstName, e.DepartmentID, e.Salary  FROM Employees e  JOIN DeptAvgSalary d ON e.DepartmentID = d.DepartmentID  WHERE e.Salary > d.AvgSalary ), HighEarnerCount AS (  SELECT DepartmentID, COUNT(EmployeeID) AS HighEarnerCount  FROM HighEarners  GROUP BY DepartmentID ) SELECT \* FROM HighEarnerCount; |

## Applying Window Functions in CTEs

CTEs can integrate window functions for ranking, running totals, etc.

**Example: Ranking Employees by Salary within Departments?**

|  |
| --- |
| WITH RankedSalaries AS (  SELECT EmployeeID, FirstName, DepartmentID, Salary,  RANK() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS SalaryRank  FROM Employees ) SELECT EmployeeID, FirstName, Salary, SalaryRank FROM RankedSalaries WHERE SalaryRank = 1; |

Here:

* RANK() assigns ranks within each department, with the highest salary ranked as 1.
* The main query filters only the top earners in each department.

# Limitations of CTEs

- **Performance Issues –** May be materialized, causing overhead.

- **Optimization Limitations –** Some engines do not optimize CTEs well.

- **Scope & Lifetime –** Only valid within a single SQL statement.

- **Recursive Limits –** Default recursion depth restrictions (e.g., 100 in SQL Server).

- **Infinite Recursion Risks –** Must have clear termination condition.

- **Readability –** Excessive nesting can make queries harder to read.

- **No Indexing –** CTEs cannot be indexed like views.