Common Table Expressions (CTEs)

Advanced Query Techniques: Sequential SQL

May 16, 2025

Contents

1 What Are They? (Meanings & Values)

Core Meaning: The SQL Storytellers Common Table Expressions, or CTEs, are like named chapters in your SQL query story. They are temporary, named result sets that you define within a single SQL statement (like a ‘SELECT‘, ‘INSERT‘, ‘UPDATE‘, or ‘DELETE‘). Think of them as a variable that holds a table, but it only exists for that one query. Once the query finishes its grand performance, the CTE vanishes like a stage prop.
Their main gig? To make complex queries less of a tangled web and more of a clear, step-by-step narrative.

Typical Values: A Temporary Table’s Twin A CTE produces a virtual table—a set of rows and columns—derived from the query defined within it. This resulting structure can be referenced by name, just like a standard table or view, but only within the scope of the statement where the CTE is defined.
Key Characteristics:

- **Temporary:** Exists only for the duration of the query execution. Not stored on disk like a permanent table.
- **Named:** You give it a name, making your SQL more self-documenting. If your query were a recipe, a CTE is like naming a pre-mixed ingredient set, "DrySpicesMix".
- **Non-Recursive (Usually) or Recursive:** Can be simple, one-off definitions, or they can be recursive, allowing them to call themselves to process hierarchical or graph-like data.

Imagine you need to calculate average salaries for departments, then find employees above that average. A CTE can hold the "department averages" table neatly.

2 Relations: How They Play with Others

CTEs don't live in a vacuum; they are team players, interacting with other CTEs and a host of SQL concepts that come before them in your learning journey.

2.1 Intra-CTE Relations: The Family Tree

Within the 'WITH' clause itself, CTEs can have relationships:

- **Basic CTEs ('WITH' clause):** The fundamental building block. Each CTE is defined by a name and a query.
- **Chained/Nested CTEs:** One CTE can reference another CTE defined earlier in the same 'WITH' clause. This allows for building up complex logic step-by-step. It's like saying, "First, figure out A (CTE1), then using A, figure out B (CTE2)."

  Chained CTE Example (Conceptual)

```
WITH
  SalesSummary AS (
    -- Complex query to summarize sales
  ),
  TopProducts AS (
    SELECT * FROM SalesSummary WHERE revenue > 10000
  )
SELECT * FROM TopProducts;
```

- **Recursive CTEs ('WITH RECURSIVE'):** A special type of CTE that can reference itself. This is the star player for hierarchical data (like org charts or parts explosions) and graph traversals. It involves an "anchor" part (the starting point) and a "recursive" part (how to get to the next level). We'll dive deeper into this syntax later.

2.2 Relations with Previous SQL Concepts: The Extended Family

A CTE's definition (the query inside the parentheses 'AS (...)') can use almost any SQL feature you've learned so far. The resulting CTE can then be used by the main query, also leveraging these features.

How CTEs Use Your Existing SQL Toolkit Here's how CTEs party with concepts from Basic, Intermediate, and Complementary SQL, plus prior Advanced SQL bits:
From BASIC SQL:

* **Conditionals: 'WHERE'**: Filter data within a CTE's definition. All your favorite operators ('BETWEEN', 'IN', 'NOT', 'LIKE', '=', '!=', '<>', '%') are invited.

  ```
  WITH HighEarners AS (
  ```

```
  SELECT employeeName, salary
  FROM Employees
  WHERE salary > 70000
)
SELECT * FROM HighEarners;
```

* **Conditionals: ‘ORDER BY‘, ‘LIMIT‘, ‘OFFSET‘**: Can be used in the main query referencing a CTE. Sometimes used in a CTE definition if it’s the final step or combined with ‘LIMIT‘/‘FETCH‘ (though ‘ORDER BY‘ in a CTE without ‘LIMIT‘ is often just a suggestion to the optimizer unless it’s the outermost query part).

From INTERMEDIATE SQL:

* **Aggregators: ‘GROUP BY‘, ‘HAVING‘**, (‘SUM‘, ‘AVG‘, etc.): Perfect for pre-calculating aggregate values. A CTE can define a summary table.

```
WITH DepartmentSalaries AS (
  SELECT departmentId, AVG(salary) AS avgSalary
  FROM Employees
  GROUP BY departmentId
  HAVING COUNT(*) > 5
)
SELECT * FROM DepartmentSalaries;
```

* **Conditionals: ‘DISTINCT‘**: Select unique rows within a CTE.
* **Arithmetic & Math Functions**: Create calculated columns inside your CTEs.
* **Date Functions**: Manipulate and filter by dates within CTE definitions.
* **Casters (‘CAST‘, ‘::‘)**: Change data types as needed within a CTE.
* **Null Space (‘COALESCE‘, ‘IS NULL‘)**: Handle ‘NULL‘ values gracefully in your CTE logic.
* **‘CASE‘ Expressions**: Implement conditional logic for columns or filters.
* **Joins (‘INNER‘, ‘LEFT‘, ‘RIGHT‘, ‘FULL OUTER‘)**: CTEs can be built from multiple joined tables. The CTE itself can then be joined to other tables or CTEs.

```
WITH EmployeeDeptInfo AS (
  SELECT e.employeeName, d.departmentName
  FROM Employees e
  INNER JOIN Departments d ON e.departmentId = d.departmentId
)
SELECT * FROM EmployeeDeptInfo;
```

From Complementary SQL:

* **Advanced 'WHERE' Conditions** (Subqueries with 'IN', 'EXISTS', 'ANY', 'ALL', 'IS DISTINCT FROM'): These can be used within a CTE definition, or the main query can use them with a CTE's result.
* **Advanced 'ORDER BY'**: Sort results from CTEs with multiple columns, 'NULLS FIRST/LAST'.
* **More Joins ('CROSS JOIN', 'NATURAL JOIN', 'SELF JOIN', 'USING' clause)**: CTEs can participate in or be defined using these join types. Recursive CTEs inherently perform a kind of self-join.
* **Advanced Aggregators ('COUNT(DISTINCT)', 'FILTER' clause)**: Enhance your aggregations within CTEs.
* **More Date Functions ('Date arithmetic', 'OVERLAPS')**: Complex date logic can live in a CTE.
* **Advanced 'CASE' usage ('CASE in ORDER BY/GROUP BY')**: Apply these within CTE definitions.
* **More Null Handling ('NULLIF')**: Useful for data cleaning or calculations in CTEs.
* **String Functions**: Manipulate text data to create derived columns in CTEs.
* **Set Operations ('UNION', 'UNION ALL', 'INTERSECT', 'EXCEPT')**: Combine results from different queries within a CTE definition, or combine results of multiple CTEs. Recursive CTEs use 'UNION' or 'UNION ALL'.

```
WITH
  Sales2022 AS (SELECT productId, salesAmount FROM Sales WHERE
  year = 2022),
  Sales2023 AS (SELECT productId, salesAmount FROM Sales WHERE
  year = 2023),
  CombinedSales AS (
    SELECT productId, salesAmount FROM Sales2022
    UNION ALL
    SELECT productId, salesAmount FROM Sales2023
  )
SELECT productId, SUM(salesAmount) FROM CombinedSales
GROUP BY productId;
```

* **Subqueries (Scalar, Correlated, in 'FROM', in 'SELECT')**: CTEs are often a more readable alternative to subqueries in the 'FROM' clause (derived tables). CTE definitions themselves can also contain various types of subqueries.

From ADVANCED SQL (Previous Topics):

* **Other Query Clauses (‘FETCH‘, ‘OFFSET‘)**: Used with the main query that selects from a CTE to limit results. Some databases allow them in CTE definitions if ‘ORDER BY‘ is also present.

* **‘LATERAL‘ Joins**: A CTE can be on the "left" side of a ‘LATERAL‘ join, and the subquery on the "right" side can reference columns from this CTE. This is powerful for applying a subquery to each row of a CTE.

```
WITH Departments AS (SELECT departmentId, departmentName
FROM AllDepartments)
SELECT d.departmentName, topEmp.employeeName
FROM Departments d
LEFT JOIN LATERAL (
  SELECT e.employeeName
  FROM Employees e
  WHERE e.departmentId = d.departmentId -- Correlated
  ORDER BY e.salary DESC
  LIMIT 1
) topEmp ON TRUE;
```

In essence, a CTE lets you package up any query you could write using these prior concepts into a neat, named module. This package can then be used as a data source for the subsequent parts of your main query. It’s like preparing ingredients before you start the main cooking.

3 How to Use Them: Structures & Syntax

Let's get down to the nuts and bolts—or rather, the 'WITH's and 'AS'es—of CTEs.

3.1 Categorizing CTEs

We can broadly group CTEs into two main types based on their behavior and syntax:

1. **Basic (Non-Recursive) CTEs:** These are the workhorses for improving query readability and modularity. They are defined once and executed (logically) once per reference.

2. **Recursive CTEs:** These are the specialists for dealing with hierarchical data or iterative processes. They have a unique structure allowing them to reference themselves.

3.2 Syntax for Basic (Non-Recursive) CTEs

The 'WITH' Clause Blueprint The basic structure for defining one or more non-recursive CTEs is:

Basic CTE Syntax

```
WITH CteName1 [(columnAlias1, columnAlias2, ...)] AS (
    -- SQL query defining CteName1
),
CteName2 [(anotherAlias1, ...)] AS (
    -- SQL query defining CteName2.
    -- This query can reference CteName1.
)
-- Main SQL query (SELECT, INSERT, UPDATE, DELETE)
SELECT ...
FROM CteName1
JOIN CteName2 ON ...
WHERE ...;
```

Key Components:

→ **'WITH' keyword:** Signals the start of CTE definitions.

→ **'CteName':** Your chosen name for the CTE. Make it descriptive! "If your CTE were a superhero, what would its name be?"

→ '[(columnAlias1, ...)]' (Optional): You can explicitly name the columns of your CTE. If omitted, the column names are derived from the 'SELECT' list of the CTE's query definition. It's good practice if the derived names are ambiguous or you want cleaner names.

→ **'AS (...)':** Contains the actual SQL query that defines the CTE's result set. This query can be as simple or as complex as needed, using concepts mentioned in Section 2.

→ **Commas ‘,‘:** Separate multiple CTE definitions within a single ‘WITH‘ clause.

→ **Main Query:** After all CTEs are defined, the main SQL statement (e.g., ‘SELECT‘) follows, which can reference any of the defined CTEs.

Usage Examples: Basic CTEs

Let’s assume we have tables:

- ‘Employees (employeeId INT, employeeName VARCHAR, departmentId INT, salary DECIMAL)‘
- ‘Departments (departmentId INT, departmentName VARCHAR)‘

1. **Single Basic CTE for Filtering and Selecting Specific Columns:**

Example: High Earning Tech Employees

```
WITH TechEmployees AS (
    SELECT employeeId, employeeName, salary
    FROM Employees
    WHERE departmentId = (
        SELECT departmentId
        FROM Departments
        WHERE departmentName = 'Technology'
    )
)
SELECT employeeName, salary
FROM TechEmployees
WHERE salary > 80000;
```

This CTE first isolates technology employees, then the main query filters them by salary.

2. **Multiple (Chained) CTEs for Step-by-Step Aggregation:**

Example: Department Average Salary vs Employee Salary

```
WITH DepartmentAvgSalaries AS (
    SELECT
        d.departmentName,
        AVG(e.salary) AS averageSalaryForDepartment
    FROM Employees e
    JOIN Departments d ON e.departmentId = d.departmentId
    GROUP BY d.departmentName
),
EmployeesWithDeptAvg AS (
    SELECT
        e.employeeName,
        e.salary,
```

```
        das.departmentName,
        das.averageSalaryForDepartment
    FROM Employees e
    JOIN Departments d ON e.departmentId = d.departmentId
    JOIN DepartmentAvgSalaries das
        ON d.departmentName = das.departmentName
)
SELECT
    employeeName,
    salary,
    departmentName,
    averageSalaryForDepartment
FROM EmployeesWithDeptAvg
WHERE salary > averageSalaryForDepartment;
```

Here, 'DepartmentAvgSalaries' calculates averages. Then 'EmployeesWithDeptAvg' uses those averages to compare individual salaries.

3.3 Syntax for Recursive CTEs

Recursive CTEs are a bit more magical. They need the ‘RECURSIVE‘ keyword.

The ‘WITH RECURSIVE‘ Blueprint The structure for a recursive CTE is:

Recursive CTE Syntax

```
WITH RECURSIVE RecursiveCteName [(columnAlias1, ...)] AS (
    -- Anchor Member:
    -- This is a non-recursive SELECT statement. It provides the
    -- intial set of rows for the recursion, the "seed" of your
    -- data tree.
    SELECT ...
    FROM ...
    WHERE ...

    UNION ALL -- Or UNION (UNION ALL is usually preferred for
    performance)

    -- Recursive Member:
    -- This SELECT statement references RecursiveCteName itself.
    -- It defines how to get from one level of recursion to the
    next.
    -- It MUST have a condition that eventually stops the recur-
    sion.
    SELECT ...
    FROM ...
    JOIN RecursiveCteName rcte ON ... -- Join condition with itself
    WHERE ... -- Crucial: Termination condition for recursion
)
-- Main query that uses the recursive CTE
SELECT * FROM RecursiveCteName;
```

Key Components:

→ **‘WITH RECURSIVE‘ keyword:** Essential for defining a recursive CTE.

→ **Anchor Member:** The base query. It runs once and provides the starting point(s) for the recursion. It does not reference the CTE itself.

→ **‘UNION ALL‘ (or ‘UNION‘):** Combines the results of the anchor member with the results of the recursive member. ‘UNION ALL‘ is generally more efficient as it doesn’t remove duplicates between iterations (duplicates within a single iteration’s result are still subject to ‘UNION‘ vs ‘UNION ALL‘ logic if the recursive member itself produces them).

→ **Recursive Member:** This query references the CTE by its own name (‘RecursiveCteName‘). It’s joined with the results from the previous iteration. It’s

crucial that this part has logic that will eventually cause it to return no new rows, thus terminating the recursion (e.g., reaching ‘NULL‘ in a ‘managerId‘ field, or a depth limit). Without a termination condition, you get an infinite loop! "A recursive CTE without an end is like a story that never lands, my friend."

Usage Examples: Recursive CTEs

Let’s enhance our ‘Employees‘ table:

- ‘Employees (employeeId INT, employeeName VARCHAR, managerId INT, salary DECIMAL)‘ (managerId refers to employeeId of the manager)

1. **Simple Hierarchy: Employee Reporting Structure**

Example: Employee Hierarchy for Employee 5

```
WITH RECURSIVE EmployeeHierarchy (employeeId, employeeName, managerId,
level) AS (
    -- Anchor Member: Select the starting employee
    SELECT employeeId, employeeName, managerId, 0 AS level
    FROM Employees
    WHERE employeeId = 5 -- Start with employeeId 5

    UNION ALL

    -- Recursive Member: Find managers of the current set of employees
    SELECT e.employeeId, e.employeeName, e.managerId, eh.level + 1
    FROM Employees e
    JOIN EmployeeHierarchy eh ON e.employeeId = eh.managerId
    -- The recursion stops naturally when eh.managerId is NULL
    (top of hierarchy)
    -- or no matching e.employeeId for eh.managerId is found.
)
SELECT employeeId, employeeName, managerId, level
FROM EmployeeHierarchy
ORDER BY level DESC;
```

This CTE starts with employee 5 and recursively finds their manager, then that manager’s manager, and so on, until it reaches the top (where managerId is NULL or not found).

2. **Generating a Series of Numbers (Illustrative Example)**

Example: Generating Numbers from 1 to 5

```
WITH RECURSIVE NumberSeries (n) AS (
    -- Anchor Member
    SELECT 1

    UNION ALL
```

```
    -- Recursive Member
    SELECT n + 1
    FROM NumberSeries
    WHERE n < 5 -- Termination condition
)
SELECT n FROM NumberSeries;
```

While many databases have 'generate_series' (a concept studied later), this shows recursion's power. It starts with 1 and adds 1 until 'n' reaches 5.

A Note on Database Support and pgAdmin4 The syntax and behavior described are standard SQL. Most modern relational databases like PostgreSQL, SQL Server, Oracle, MySQL (version 8+), and SQLite support CTEs, including recursive ones. You can practice these queries directly in tools like pgAdmin4 for PostgreSQL. While the core CTE concept is standard, some specific behaviors or performance characteristics might vary slightly between database systems. For example, some databases (like PostgreSQL and SQL Server) allow CTEs to be used in 'INSERT', 'UPDATE', and 'DELETE' statements directly, which can be very powerful.

```
-- Example: Update salaries using a CTE (PostgreSQL syntax)
WITH AverageSalaries AS (
  SELECT departmentId, AVG(salary) as avgDeptSalary
  FROM Employees GROUP BY departmentId
)
UPDATE Employees e
SET salary = salary * 1.10
FROM AverageSalaries avgs
WHERE e.departmentId = avgs.departmentId AND e.salary <
avgs.avgDeptSalary;
```

Always check your specific database's documentation for any nuances.

4 Why Use Them? (Advantages)

CTEs aren't just syntactic sugar; they bring tangible benefits to your SQL development. Why bother with 'WITH'? Let's count the ways!

✓ **Improved Readability & Maintainability:** This is the star advantage! CTEs break down long, complex queries into smaller, logical, named steps. Each CTE can focus on a specific part of the problem. It's like reading a well-structured essay with clear paragraphs instead of one massive block of text. Future you (and your colleagues) will thank you. "A CTE a day keeps the query chaos away!"

Alternative without CTEs: Often involves deeply nested subqueries or multiple derived tables, which can become a "query salad" – hard to digest and even harder to modify.

✓ **Modularity and Reusability (within a single query):** Define a complex calculation or filtered dataset once in a CTE, then reference it multiple times in the subsequent parts of your main query or in other CTEs within the same 'WITH' clause. This avoids repeating the same subquery logic, reducing redundancy and the chance of errors if that logic needs to change.

Alternative without CTEs: You might have to copy-paste the same subquery multiple times, making the query longer and updates more error-prone. "Define it once, use it thrice (or more), that's what CTEs are for!"

✓ **Recursive Queries Made Possible (and Elegant):** For querying hierarchical data (like organizational charts, bill of materials, social network connections) or performing graph traversals, recursive CTEs are the standard, elegant solution. Trying to do this with traditional SQL joins and subqueries is often extremely complex, inefficient, or simply not possible for arbitrary depths.

Alternative without CTEs for recursion: Often involves procedural code (loops in application layer) or very complex, database-specific extensions, if available at all for the same level of declarative power.

✓ **Better Organization and Step-by-Step Logic:** CTEs allow you to build your query in a more procedural-like fashion, even though SQL is declarative. You can define intermediate results step by step, making the overall logic flow clearer. This is especially helpful for multi-stage calculations or transformations.

✓ **Simplifying Complex Joins and Aggregations:** You can perform complex joins or aggregations within a CTE, and then join this simplified, pre-processed result set to other tables or CTEs. This can make the main query much cleaner.

✓ **Potential for Easier Debugging:** When developing a complex query, you can often test each CTE individually (by selecting all from it and commenting out later parts) to ensure it produces the expected intermediate results. This helps isolate problems more quickly.

The Bottom Line CTEs help you write SQL that is not just correct, but also clear, understandable, and easier to manage over time. They promote a "divide and conquer" strategy for query complexity.

5 Watch Out! (Disadvantages)

While CTEs are incredibly useful, they aren't a silver bullet. Like any tool, they have potential pitfalls or situations where they might not be the optimal choice. Forewarned is forearmed!

(!) **Potential Performance Surprises (Optimization Fence):** In some database systems or for certain complex CTEs, the optimizer might treat a CTE as an "optimization fence." This means it might materialize the CTE's results (i.e., compute and store them temporarily) and then use that materialized result, rather than inlining the CTE's logic into the main query. This can sometimes prevent the optimizer from applying further optimizations like pushing predicates down into the CTE or reordering joins across the CTE boundary as effectively as it might with a derived table or direct joins. *However, modern optimizers (like in PostgreSQL) are increasingly sophisticated and often do a great job of inlining CTEs. It's not a guaranteed disadvantage, but something to be aware of and test if performance is critical. Sometimes a CTE is faster, sometimes a subquery is. No need to fret, just test and let the 'EXPLAIN' plan be your friend!*

(!) **Scope Limitation: One-Query Wonders** A CTE is defined for and exists only within the scope of a single SQL statement. You cannot define a CTE and then reference it in a separate, subsequent SQL statement. If you need a temporary result set that persists across multiple statements within a session or transaction, you'd need to use a temporary table. "A CTE's fame is fleeting, for the next query, it's retreating."

(!) **No Direct Indexing on CTE Results:** You cannot create an explicit index on the intermediate result set of a CTE. If a CTE produces a very large number of rows and is referenced multiple times within the main query (and the optimizer chooses to materialize it), each reference might involve a full scan of that unindexed materialized data. In such scenarios, for procedural workflows, creating a temporary table and explicitly indexing it might offer better performance.

(!) **Verbosity for Trivial Cases:** For a very simple, non-repeated subquery, wrapping it in a CTE might add a bit of verbosity without a significant gain in readability. It's about finding the right balance. "Using a 'WITH' for a tiny 'SELECT' might be correct, but perhaps a bit over-decked."

(!) **Recursive CTE Pitfalls:**

- **Infinite Loops:** The most common issue. If the recursive member doesn't have a proper termination condition (a 'WHERE' clause that eventually stops rows from being generated), the query can run indefinitely, consuming resources until it's killed or hits a recursion limit. "The query that never ends, a recursive journey that transcends... all reasonable time limits."
- **Performance with Deep/Wide Recursion:** Recursive queries can be resource-intensive, especially if the recursion depth is large or if each step generates many new rows (wide fan-out). Careful design and indexing of the underlying tables are crucial.

(!) **Readability of ‘UPDATE‘/‘DELETE‘ with CTEs (in some DBMS):** While powerful, using CTEs with ‘UPDATE‘ or ‘DELETE‘ statements (where supported, e.g., PostgreSQL) can sometimes make the statement’s target less immediately obvious compared to simpler forms, especially for those unfamiliar with the pattern. The CTE defines what to act upon, but the action is on the main table.

A Balanced View Most of these "disadvantages" are more like "things to be aware of" or "trade-offs." The benefits of CTEs, particularly for readability and handling recursion, often far outweigh these considerations for many use cases. Always analyze your specific query and database system if performance is a major concern.