Mastering Common Table Expressions (CTEs) in MS SQL (Beginner → Advanced)

Common Table Expressions behave like short-lived views that live only for the duration of a single query. They let you write cleaner, modular SQL, break complex logic into parts, and enable powerful patterns such as recursion.

This guide walks through CTEs from fundamentals to advanced data-engineering patterns using AdventureWorks-friendly examples.

---

1. What is a CTE?

A CTE is a temporary named result set you define before your main query. It makes SQL easier to read and reuse.

Basic shape:

```
WITH cte_name AS (
    SELECT ...
)
SELECT * FROM cte_name;
```

You can think of a CTE as telling SQL: “Treat this query as a virtual table for a moment.”

---

2. Beginner Level

2.1 Simple CTE

This extracts a subset of data then queries it.

```
WITH HighPriceProducts AS (
    SELECT ProductID, Name, ListPrice
    FROM Production.Product
    WHERE ListPrice > 1000
)
SELECT *
FROM HighPriceProducts;
```

2.2 Using multiple CTEs

CTEs can be chained.

```
WITH p AS (
    SELECT ProductID, ListPrice FROM Production.Product
),
q AS (
    SELECT ProductID, ListPrice, ListPrice * 0.1 AS Tax FROM p
)
SELECT * FROM q;
```

2.3 Filtering then joining

```
WITH RecentOrders AS (
    SELECT SalesOrderID, CustomerID, OrderDate
    FROM Sales.SalesOrderHeader
    WHERE OrderDate >= '2014-01-01'
)
SELECT r.*, c.PersonID
FROM RecentOrders r
JOIN Sales.Customer c
    ON r.CustomerID = c.CustomerID;
```

---

3. Intermediate Level

3.1 CTE for windowing and reuse

```
WITH OrderStats AS (
    SELECT
        SalesOrderID,
        CustomerID,
        TotalDue,
        ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY TotalDue DESC) AS rn
    FROM Sales.SalesOrderHeader
)
SELECT * FROM OrderStats WHERE rn = 1;
```

3.2 CTEs to simplify long calculations

```sql
WITH ProductCosts AS (
    SELECT ProductID, StandardCost FROM Production.Product
),
SalesInfo AS (
    SELECT ProductID, SUM(LineTotal) AS Revenue
    FROM Sales.SalesOrderDetail
    GROUP BY ProductID
)
SELECT
    s.ProductID,
    s.Revenue,
    p.StandardCost,
    s.Revenue - p.StandardCost AS Profit
FROM SalesInfo s
LEFT JOIN ProductCosts p
    ON s.ProductID = p.ProductID;
```

3.3 Using CTEs instead of subqueries

This makes code more readable.

```sql
WITH Ranked AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY TotalDue DESC) AS rn
    FROM Sales.SalesOrderHeader
)
SELECT * FROM Ranked WHERE rn <= 10;
```

---

4. Recursive CTEs (Advanced)

A recursive CTE repeatedly refers to itself to generate hierarchical or sequential data.

Structure:

```sql
WITH cte_name AS (
    -- Anchor member
    SELECT ...
    UNION ALL
    -- Recursive member
    SELECT ... FROM cte_name
)
SELECT * FROM cte_name;
```

4.1 Generate a numeric sequence

```
WITH Numbers AS (
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM Numbers WHERE n < 10
)
SELECT * FROM Numbers;
```

4.2 Explore an organizational hierarchy

```
WITH EmpCTE AS (
    -- Anchor: top-level employees
    SELECT BusinessEntityID, ManagerID, 0 AS Level
    FROM HumanResources.Employee
    WHERE ManagerID IS NULL

    UNION ALL

    -- Recursive: employees reporting to previous level
    SELECT e.BusinessEntityID, e.ManagerID, Level + 1
    FROM HumanResources.Employee e
    JOIN EmpCTE c
        ON e.ManagerID = c.BusinessEntityID
)
SELECT * FROM EmpCTE;
```

4.3 Traversing Bill of Materials (BOM)

AdventureWorks includes a BOM table perfect for hierarchical recursion.

```
WITH BOM AS (
    SELECT ProductAssemblyID, ComponentID, PerAssemblyQty, 1 AS Level
    FROM Production.BillOfMaterials
    WHERE ProductAssemblyID = 800  -- Example product

    UNION ALL

    SELECT b.ProductAssemblyID, b.ComponentID, b.PerAssemblyQty, Level + 1
    FROM Production.BillOfMaterials b
    JOIN BOM c
        ON b.ProductAssemblyID = c.ComponentID
)
SELECT * FROM BOM ORDER BY Level;
```

5. Advanced Patterns

5.1 Deduplication using ranked CTE

```
WITH Ranked AS (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY EmailAddress ORDER BY ModifiedDate DESC) AS
rn
    FROM Person.EmailAddress
)
DELETE FROM Ranked WHERE rn > 1;
```

5.2 Break complex pipelines into stages

CTEs can behave like a step-by-step data-engineering pipeline.

```
WITH Raw AS (
    SELECT * FROM Sales.SalesOrderDetail
),
Clean AS (
    SELECT ProductID, LineTotal FROM Raw WHERE LineTotal > 0
),
Aggregated AS (
    SELECT ProductID, SUM(LineTotal) AS Revenue FROM Clean GROUP BY ProductID
)
SELECT * FROM Aggregated;
```

5.3 Combine CTEs with window functions

```
WITH x AS (
    SELECT ProductID, ListPrice,
           PERCENT_RANK() OVER (ORDER BY ListPrice) AS p
    FROM Production.Product
)
SELECT * FROM x WHERE p > 0.9;
```

5.4 Cycle detection (guard against infinite recursion)

SQL Server requires `OPTION (MAXRECURSION n)` to control runaway recursion.

```sql
WITH Numbers AS (
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM Numbers
)
SELECT * FROM Numbers
OPTION (MAXRECURSION 100);
```

---

6. Performance Notes

1. CTEs do not materialize by default; they are query optimizations, not stored results.
2. Recursive CTEs may perform slower on deep hierarchies—consider a table of hierarchy paths.
3. Use MAXRECURSION to prevent infinite loops.
4. CTEs improve readability, which often leads indirectly to better maintainability.

---

7. Real Data Engineering Scenarios

7.1 Sessionization

Use row-number gaps inside a CTE to segment user sessions.

7.2 Financial period rollups

Break complex transformations (clean → enrich → aggregate) into readable CTE layers.

7.3 Hierarchical transformations

BOM expansion, org charts, product lineage, and workflow dependency chains work elegantly with recursion.

---

8. Summary

CTEs serve as SQL's way of letting you narrate your logic instead of compressing logic into tangled subqueries. They help you write clean, modular analytics and tackle recursive hierarchies with surprising elegance.

Further progress often comes from combining CTEs with window functions, building chained transformations, and modeling recursive business structures.