# All About CTEs (Common Table Expressions) with Practical Examples

## Table of Contents

1. [What is a CTE?](#what-is-a-cte)
2. [Basic CTE Syntax](#basic-cte-syntax)
3. [Simple CTE Examples](#simple-cte-examples)
4. [Multiple CTEs](#multiple-ctes)
5. [Finding Nth Values with CTE](#finding-nth-values-with-cte)
6. [CTE with Aggregations](#cte-with-aggregations)
7. [Data Transformation with CTE](#data-transformation-with-cte)
8. [Recursive CTE](#recursive-cte)
9. [Running Totals with CTE](#running-totals-with-cte)
10. [Date/Time Analysis](#datetime-analysis)
11. [Data Validation](#data-validation)
12. [Complex Filtering](#complex-filtering)
13. [CTE with CASE Statements](#cte-with-case-statements)
14. [CTE vs Alternatives](#cte-vs-alternatives)

## What is a CTE?

A **Common Table Expression (CTE)** is a temporary named result set that exists only during the execution of a single SQL statement. It’s defined using the WITH clause and provides a way to create more readable and maintainable complex queries.

### Key Benefits:

* **Readability**: Makes complex queries more readable
* **Reusability**: Can reference the same CTE multiple times
* **Organization**: Breaks down complex logic into manageable steps
* **Recursion**: Supports recursive operations
* **Debugging**: Easier to test individual parts of complex queries

## Basic CTE Syntax

WITH cte\_name AS (  
 SELECT columns FROM table WHERE conditions  
)  
SELECT \* FROM cte\_name;

## Simple CTE Examples

### Basic Filtering

-- Find employees with salary > 50000  
WITH high\_earners AS (  
 SELECT name, salary, department   
 FROM employees   
 WHERE salary > 50000  
)  
SELECT \* FROM high\_earners ORDER BY salary DESC;

### CTE with Calculations

-- Calculate bonus (10% of salary) for each employee  
WITH employee\_bonus AS (  
 SELECT   
 name,   
 salary,   
 salary \* 0.10 as bonus,  
 salary + (salary \* 0.10) as total\_compensation  
 FROM employees  
)  
SELECT \* FROM employee\_bonus;

## Multiple CTEs

You can define multiple CTEs in a single query for complex operations:

WITH   
-- CTE 1: Department averages  
dept\_averages AS (  
 SELECT department, AVG(salary) as avg\_salary  
 FROM employees  
 GROUP BY department  
),  
-- CTE 2: Employee performance vs department average  
employee\_performance AS (  
 SELECT   
 e.name,  
 e.salary,  
 e.department,  
 d.avg\_salary,  
 CASE   
 WHEN e.salary > d.avg\_salary THEN 'Above Average'  
 ELSE 'Below Average'  
 END as performance  
 FROM employees e  
 JOIN dept\_averages d ON e.department = d.department  
)  
SELECT \* FROM employee\_performance;

## Finding Nth Values with CTE

### Find 3rd Highest Salary

WITH salary\_ranks AS (  
 SELECT   
 name,   
 salary,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) as rank  
 FROM employees  
)  
SELECT name, salary FROM salary\_ranks WHERE rank = 3;

### Find 2nd Lowest Salary by Department

WITH dept\_salary\_ranks AS (  
 SELECT   
 name,   
 department,   
 salary,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary ASC) as rank  
 FROM employees  
)  
SELECT name, department, salary   
FROM dept\_salary\_ranks   
WHERE rank = 2;

## CTE with Aggregations

### Department-wise Statistics

WITH dept\_stats AS (  
 SELECT   
 department,  
 COUNT(\*) as total\_employees,  
 SUM(salary) as total\_salary,  
 AVG(salary) as avg\_salary,  
 MAX(salary) as max\_salary,  
 MIN(salary) as min\_salary  
 FROM employees  
 GROUP BY department  
)  
SELECT   
 department,  
 total\_employees,  
 total\_salary,  
 ROUND(avg\_salary, 2) as average\_salary,  
 max\_salary - min\_salary as salary\_range  
FROM dept\_stats;

## Data Transformation with CTE

### Clean and Transform Employee Data

WITH clean\_employee\_data AS (  
 SELECT   
 UPPER(TRIM(name)) as clean\_name,  
 LOWER(TRIM(email)) as clean\_email,  
 salary,  
 CASE   
 WHEN department = 'IT' THEN 'Information Technology'  
 WHEN department = 'HR' THEN 'Human Resources'  
 ELSE department  
 END as full\_department\_name  
 FROM employees  
)  
SELECT \* FROM clean\_employee\_data;

## Recursive CTE

### Organizational Hierarchy

-- Find all employees in a management hierarchy  
WITH employee\_hierarchy AS (  
 -- Base case: Top-level managers (no manager)  
 SELECT employee\_id, name, manager\_id, 1 as level  
 FROM employees   
 WHERE manager\_id IS NULL  
   
 UNION ALL  
   
 -- Recursive case: Find employees under each manager  
 SELECT e.employee\_id, e.name, e.manager\_id, eh.level + 1  
 FROM employees e  
 INNER JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id  
)  
SELECT   
 employee\_id,  
 name,  
 level,  
 REPLICATE(' ', level - 1) + name as indented\_name  
FROM employee\_hierarchy  
ORDER BY level, name;

## Running Totals with CTE

### Calculate Running Total of Salaries

WITH salary\_running\_total AS (  
 SELECT   
 name,  
 salary,  
 SUM(salary) OVER(ORDER BY salary DESC) as running\_total  
 FROM employees  
)  
SELECT   
 name,  
 salary,  
 running\_total,  
 ROUND((running\_total \* 100.0) / (SELECT SUM(salary) FROM employees), 2) as percentage\_of\_total  
FROM salary\_running\_total;

## Date/Time Analysis

### Monthly Sales Analysis

-- Assuming we have a sales table  
WITH monthly\_sales AS (  
 SELECT   
 YEAR(sale\_date) as year,  
 MONTH(sale\_date) as month,  
 SUM(amount) as monthly\_total,  
 COUNT(\*) as transaction\_count  
 FROM sales  
 GROUP BY YEAR(sale\_date), MONTH(sale\_date)  
),  
sales\_with\_growth AS (  
 SELECT   
 year,  
 month,  
 monthly\_total,  
 transaction\_count,  
 LAG(monthly\_total) OVER(ORDER BY year, month) as previous\_month,  
 monthly\_total - LAG(monthly\_total) OVER(ORDER BY year, month) as growth  
 FROM monthly\_sales  
)  
SELECT   
 year,  
 month,  
 monthly\_total,  
 previous\_month,  
 COALESCE(growth, 0) as monthly\_growth,  
 CASE   
 WHEN previous\_month IS NULL THEN 'N/A'  
 WHEN growth > 0 THEN 'Increase'  
 WHEN growth < 0 THEN 'Decrease'  
 ELSE 'No Change'  
 END as trend  
FROM sales\_with\_growth;

## Data Validation

### Find Data Quality Issues

WITH data\_quality\_check AS (  
 SELECT   
 employee\_id,  
 name,  
 email,  
 salary,  
 CASE   
 WHEN name IS NULL OR TRIM(name) = '' THEN 'Missing Name'  
 WHEN email IS NULL OR email NOT LIKE '%@%' THEN 'Invalid Email'  
 WHEN salary <= 0 THEN 'Invalid Salary'  
 ELSE 'Valid'  
 END as data\_status  
 FROM employees  
)  
SELECT \* FROM data\_quality\_check   
WHERE data\_status != 'Valid';

## Complex Filtering

### Advanced Multi-Step Filtering

-- Find employees who earn more than department average   
-- and are in top 3 earners overall  
WITH   
dept\_avg AS (  
 SELECT department, AVG(salary) as avg\_salary  
 FROM employees  
 GROUP BY department  
),  
above\_dept\_avg AS (  
 SELECT e.\*, d.avg\_salary  
 FROM employees e  
 JOIN dept\_avg d ON e.department = d.department  
 WHERE e.salary > d.avg\_salary  
),  
top\_earners AS (  
 SELECT \*, ROW\_NUMBER() OVER(ORDER BY salary DESC) as overall\_rank  
 FROM above\_dept\_avg  
)  
SELECT name, department, salary, avg\_salary, overall\_rank  
FROM top\_earners  
WHERE overall\_rank <= 3;

## CTE with CASE Statements

### Categorize Employees and Calculate Adjusted Salary

WITH employee\_categories AS (  
 SELECT   
 name,  
 salary,  
 department,  
 CASE   
 WHEN salary >= 70000 THEN 'Senior'  
 WHEN salary >= 50000 THEN 'Mid-Level'  
 ELSE 'Junior'  
 END as category,  
 CASE   
 WHEN salary >= 70000 THEN salary \* 1.05 -- 5% bonus  
 WHEN salary >= 50000 THEN salary \* 1.03 -- 3% bonus  
 ELSE salary \* 1.02 -- 2% bonus  
 END as adjusted\_salary  
 FROM employees  
)  
SELECT   
 category,  
 COUNT(\*) as employee\_count,  
 AVG(salary) as avg\_current\_salary,  
 AVG(adjusted\_salary) as avg\_adjusted\_salary,  
 AVG(adjusted\_salary - salary) as avg\_bonus  
FROM employee\_categories  
GROUP BY category;

## Advanced CTE Examples

### CTE for Pivot-like Operations

-- Transform rows to columns using CTE  
WITH department\_salaries AS (  
 SELECT   
 name,  
 CASE WHEN department = 'IT' THEN salary END as IT\_salary,  
 CASE WHEN department = 'Sales' THEN salary END as Sales\_salary,  
 CASE WHEN department = 'HR' THEN salary END as HR\_salary  
 FROM employees  
)  
SELECT   
 SUM(IT\_salary) as total\_IT\_salary,  
 SUM(Sales\_salary) as total\_Sales\_salary,  
 SUM(HR\_salary) as total\_HR\_salary,  
 AVG(IT\_salary) as avg\_IT\_salary,  
 AVG(Sales\_salary) as avg\_Sales\_salary,  
 AVG(HR\_salary) as avg\_HR\_salary  
FROM department\_salaries;

### CTE for Ranking Analysis

-- Complex ranking with multiple criteria  
WITH employee\_rankings AS (  
 SELECT   
 name,  
 department,  
 salary,  
 ROW\_NUMBER() OVER(ORDER BY salary DESC) as overall\_rank,  
 ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) as dept\_rank,  
 DENSE\_RANK() OVER(ORDER BY salary DESC) as salary\_dense\_rank,  
 NTILE(4) OVER(ORDER BY salary DESC) as quartile  
 FROM employees  
)  
SELECT   
 name,  
 department,  
 salary,  
 overall\_rank,  
 dept\_rank,  
 salary\_dense\_rank,  
 CASE quartile  
 WHEN 1 THEN 'Top 25%'  
 WHEN 2 THEN 'Upper Middle 25%'  
 WHEN 3 THEN 'Lower Middle 25%'  
 WHEN 4 THEN 'Bottom 25%'  
 END as salary\_quartile  
FROM employee\_rankings  
ORDER BY overall\_rank;

### CTE for Gap Analysis

-- Find gaps in employee IDs  
WITH id\_gaps AS (  
 SELECT   
 employee\_id,  
 employee\_id - ROW\_NUMBER() OVER(ORDER BY employee\_id) as gap\_group  
 FROM employees  
),  
gap\_analysis AS (  
 SELECT   
 gap\_group,  
 MIN(employee\_id) as start\_id,  
 MAX(employee\_id) as end\_id,  
 COUNT(\*) as consecutive\_count  
 FROM id\_gaps  
 GROUP BY gap\_group  
)  
SELECT   
 start\_id,  
 end\_id,  
 consecutive\_count,  
 CASE   
 WHEN start\_id = end\_id THEN 'Single ID'  
 ELSE 'Range: ' + CAST(start\_id AS VARCHAR) + '-' + CAST(end\_id AS VARCHAR)  
 END as id\_range  
FROM gap\_analysis  
ORDER BY start\_id;

## CTE Best Practices

### 1. Use Meaningful Names

-- Good  
WITH high\_performing\_employees AS (...)  
  
-- Avoid  
WITH cte1 AS (...)

### 2. Break Complex Logic into Steps

-- Instead of one complex query, use multiple CTEs  
WITH   
step1\_filter AS (...),  
step2\_calculations AS (...),  
step3\_final\_format AS (...)  
SELECT \* FROM step3\_final\_format;

### 3. Add Comments for Complex CTEs

WITH   
-- Calculate department averages for performance comparison  
dept\_averages AS (  
 SELECT department, AVG(salary) as avg\_salary  
 FROM employees  
 GROUP BY department  
),  
-- Identify employees above department average  
above\_average\_performers AS (  
 SELECT e.\*, d.avg\_salary  
 FROM employees e  
 JOIN dept\_averages d ON e.department = d.department  
 WHERE e.salary > d.avg\_salary  
)  
SELECT \* FROM above\_average\_performers;

## CTE vs Alternatives

| Feature | CTE | Subquery | Temporary Table | View |
| --- | --- | --- | --- | --- |
| **Scope** | Single statement | Single statement | Session | Permanent |
| **Reusability** | Within query | No | Yes | Yes |
| **Performance** | Good | Variable | Best | Good |
| **Memory Usage** | Low | Low | High | Low |
| **Recursion** | Yes | No | No | No |
| **Readability** | Excellent | Poor for complex | Good | Good |
| **Maintenance** | Easy | Difficult | Moderate | Easy |

### When to Use CTEs:

* **Complex queries** that need to be broken down
* **Recursive operations** (hierarchical data)
* **Multiple references** to the same subquery
* **Improved readability** is priority
* **Temporary calculations** within a single query

### When to Avoid CTEs:

* **Simple queries** where a direct SELECT works
* **Performance critical** operations (consider temp tables)
* **Cross-session** data sharing (use temp tables or views)
* **Very large datasets** (temp tables might be better)

## Summary

CTEs are powerful tools for: - Making complex SQL queries more readable and maintainable - Breaking down complex logic into manageable steps - Handling recursive operations elegantly - Improving code organization and debugging - Creating reusable query components within a single statement