WITH customers\_in\_usa AS (

SELECT

customerName, state

FROM

customers

WHERE

country = 'USA'

) SELECT

customerName

FROM

customers\_in\_usa

WHERE

state = 'CA'

ORDER BY customerName;

How it works.

* First, define a CTE with the name customers\_in\_usa that stores the customer name and state of customers in the USA. The defining query retrieves data from the customers table.
* Second, select the customers located in California from the CTE.

A **Common Table Expression** (CTE), introduced with the WITH keyword in SQL, is a temporary result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs help make complex queries more readable, maintainable, and reusable. Let's explore **why** and **when** CTEs are used, along with their key advantages:

**Reasons to Use a CTE:**

1. **Improves Readability**:
   * Complex queries, especially those with multiple joins, subqueries, or aggregations, can become hard to read. CTEs allow you to break a large, complex query into smaller, logical pieces. This improves clarity, as each CTE can represent a specific step in the query process.
   * For example, in your previous query, the CTE topsales2003 isolates the logic for calculating top sales in 2003, making the final query that fetches employee details cleaner.
2. **Reusability of Result Sets**:
   * CTEs allow you to define a result set once and reuse it multiple times within the same query. This avoids the need to repeat complex subqueries, which simplifies the overall query structure.
   * Example: If you need to reference the same set of complex aggregated data multiple times, using a CTE prevents you from repeating the logic, which enhances both clarity and performance.
3. **Self-Referencing/Recursive Queries**:
   * CTEs are essential for writing **recursive queries**. Recursion allows you to solve hierarchical problems like organizational charts, family trees, or pathfinding (e.g., finding all the direct and indirect reports of an employee).
   * Without CTEs, writing recursive queries would be significantly harder, requiring stored procedures or looping constructs in application code.
4. **Organized Data Processing**:
   * With a CTE, you can stage and process data in steps. This is especially helpful for complex data transformations, such as cleaning, filtering, or calculating derived metrics.
   * It’s like defining intermediate results that lead to the final output. For instance, you might compute sales totals in one CTE, filter high-performing employees in another, and join them together in the main query.
5. **Temporary Data Representation**:
   * CTEs act like temporary tables that exist only for the duration of the query execution. You don’t need to create and manage physical temporary tables in the database, making CTEs a lightweight and flexible alternative for managing intermediate results.
6. **Better Query Organization Compared to Subqueries**:
   * Although subqueries can achieve similar functionality, they can be deeply nested, making the query hard to read. CTEs provide a more structured approach by letting you define intermediate result sets with meaningful names.

**Key Features of CTEs:**

* **Scope**: CTEs are temporary and only exist during the execution of the query in which they are defined. Once the query is finished, the CTE is no longer available.
* **Modular**: You can use multiple CTEs in a query, building a step-by-step flow of data manipulation and making the query easier to debug.
* **Recursive CTEs**: CTEs support recursion, allowing you to work with hierarchical data, such as finding all employees under a certain manager or traversing an organizational tree.

Create the employees table:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

manager\_id INT,

FOREIGN KEY (manager\_id) REFERENCES employees(employee\_id)

);

**employee\_id**: Unique ID for each employee.

**first\_name** and **last\_name**: Employee's name.

**manager\_id**: Refers to the employee's manager (points to another employee's employee\_id).

Insert sample data into the employees table:

INSERT INTO employees (employee\_id, first\_name, last\_name, manager\_id) VALUES

(1, 'John', 'Doe', NULL), -- CEO (Top of hierarchy)

(2, 'Jane', 'Smith', 1), -- Reports to John

(3, 'Alice', 'Johnson', 2), -- Reports to Jane

(4, 'Bob', 'Lee', 2), -- Reports to Jane

(5, 'Chris', 'Evans', 3), -- Reports to Alice

(6, 'Emily', 'Blunt', 3), -- Reports to Alice

(7, 'Michael', 'Clark', 4), -- Reports to Bob

(8, 'David', 'Harris', 4), -- Reports to Bob

(9, 'Sara', 'Parker', 1), -- Reports to John

(10, 'Will', 'Smith', 9); -- Reports to Sara

**Hierarchy:**

* **John Doe** (CEO) – Employee ID 1, has no manager (manager\_id = NULL).
* **Jane Smith** reports to John Doe.
* **Alice Johnson** and **Bob Lee** report to Jane Smith.
* **Chris Evans** and **Emily Blunt** report to Alice Johnson.
* **Michael Clark** and **David Harris** report to Bob Lee.
* **Sara Parker** reports to John Doe.
* **Will Smith** reports to Sara Parker.

**Example Recursive Query to Get the Organizational Hierarchy:**

* Now, you can use the following **recursive CTE** to get the hierarchy of all employees under John Doe (the CEO with employee\_id = 1):

WITH RECURSIVE EmployeeHierarchy AS (

-- Anchor member: Start with the top-level manager (John Doe)

SELECT employee\_id, first\_name, last\_name, manager\_id, 1 AS level

FROM employees

WHERE employee\_id = 1 -- Starting point (John Doe, the CEO)

UNION ALL

-- Recursive member: Find all employees who report to the current level of managers

SELECT e.employee\_id, e.first\_name, e.last\_name, e.manager\_id, eh.level + 1

FROM employees e

INNER JOIN EmployeeHierarchy eh ON e.manager\_id = eh.employee\_id

)

-- Final query: Retrieve all employees in the hierarchy under John Doe

SELECT \* FROM EmployeeHierarchy;

**UNION ALL**: Combines the results of the anchor member and recursive member. UNION ALL is often used instead of UNION to avoid eliminating duplicate rows.

When using a **CTE** (Common Table Expression) in a query, whether or not the CTE result is stored in memory and reused for the second or subsequent references depends on the **database system's execution strategy**. In MySQL 8.0 and above, CTEs are **not necessarily materialized** (stored in memory) unless explicitly needed, and their results might be recalculated depending on how the query is structured.

**MySQL CTE Execution Strategy:**

* **Non-Recursive CTEs**: In **MySQL 8.0+**, a non-recursive CTE is usually **inlined**, meaning the SQL optimizer replaces the CTE reference with the actual query logic when it's used in the SELECT statement. If a non-recursive CTE is used multiple times in a query, the database engine might **re-execute** the CTE query each time unless it decides to materialize (store) the result temporarily for optimization. By default, MySQL doesn't materialize CTE results unless necessary.
* **Recursive CTEs**: For **recursive CTEs**, MySQL does materialize the intermediate results, as recursion requires storing intermediate rows at each recursion step.

**Materialization and Reuse of CTEs:**

1. **Single Use of CTE:** When a CTE is referenced only once in the query, MySQL treats it similarly to a derived table. In this case, the optimizer might **inline** the CTE and treat it as part of the query plan without storing the result in memory.

WITH DeptSalaries AS (

SELECT department\_id, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department\_id

)

SELECT department\_id, avg\_salary FROM DeptSalaries;

In this case, the query in the DeptSalaries CTE will be executed once, and there's no need for materialization.

**Multiple Use of CTE (Potential for Materialization):** When the CTE is referenced multiple times within the same query, MySQL may or may not materialize the result, depending on its optimizer's decision. If MySQL decides not to materialize, it will **re-execute the CTE each time** it is referenced.

Example of a CTE used twice:

WITH DeptSalaries AS (

SELECT department\_id, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department\_id

)

-- First usage

SELECT department\_id, avg\_salary FROM DeptSalaries;

-- Second usage

WITH DeptSalaries AS (

SELECT department\_id, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department\_id

)

SELECT COUNT(department\_id) FROM DeptSalaries;

* **Without Materialization**: MySQL may execute the DeptSalaries query twice, once for each usage.
* **With Materialization**: MySQL could store the result in a temporary structure after the first execution and reuse it for the second part, avoiding recalculation.
* **Forcing Materialization (Temporary Table):** In some database systems like **SQL Server** or **PostgreSQL**, you can **force materialization** to store the CTE result temporarily. In MySQL, there is no direct way to force materialization for a non-recursive CTE, but materialization might still happen automatically based on the query complexity.
* **Example of Materialization Use Case:** If a CTE performs an expensive or complex calculation and is used multiple times in a query, materialization can lead to performance gains. Without materialization, the database engine will need to re-execute the CTE each time it is referenced.

In the context of a **MySQL database**, **materialization** refers to the process where intermediate results of a subquery, a **CTE (Common Table Expression)**, or a **derived table** are **stored temporarily**—either in memory or on disk—so that they can be reused without recalculating them multiple times.

**Key Points about Materialization in MySQL:**

1. **Definition**:
   * Materialization means the result of a subquery or CTE is evaluated once, stored in a temporary result set (often in memory or as a temporary table), and then reused as needed within the query.
   * This helps avoid re-execution of the subquery or CTE each time it is referenced in the query.
2. **When Materialization Occurs**:
   * **CTEs**: In **MySQL 8.0 and above**, non-recursive CTEs are typically not materialized by default; they are inlined and re-executed if referenced multiple times. However, for **recursive CTEs**, materialization is necessary since the results need to be stored at each step of the recursion.
   * **Derived Tables (Subqueries in FROM)**: MySQL materializes derived tables by default. This means the result of the subquery in the FROM clause is computed once and stored, then reused for further operations in the main query.
3. **Advantages of Materialization**:
   * **Reusability**: If a CTE or subquery is referenced multiple times in a query, materialization ensures it is executed once and the result is reused, improving performance by avoiding recomputation.
   * **Efficient Query Execution**: For complex queries involving large joins, aggregations, or filtering, materialization can prevent reprocessing of data.
   * **Handling Expensive Operations**: For queries that involve expensive operations (like aggregations or window functions), materialization allows those operations to be done once and reused, which is especially useful in large datasets.