**A Recursive Common Table Expression (CTE)**

A Recursive Common Table Expression (CTE) in MySQL is a powerful feature that allows you to perform recursive operations on hierarchical data stored in a table. Recursive CTEs are often used to work with data structures like trees, graphs, and hierarchies, where a record can have a parent-child relationship with other records in the same table. Recursive CTEs make it easier to traverse these structures in a more structured way.

Here's a step-by-step explanation of how to create and use a recursive CTE in MySQL:

**Database Setup:** Ensure you have a database and a table with hierarchical data. For this explanation, let's assume you have a table called **employees** with the following structure:

CREATE TABLE employees (

id INT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

manager\_id INT,

-- Other columns

);

-- Inserting 20 sample rows into the employees table

INSERT INTO employees (id, name, manager\_id)

VALUES

(1, 'John Doe', NULL), -- CEO

(2, 'Jane Smith', 1), -- VP of Marketing

(3, 'Bob Johnson', 1), -- VP of Sales

(4, 'Alice Brown', 2), -- Marketing Manager 1

(5, 'Charlie Davis', 2), -- Marketing Manager 2

(6, 'Eve White', 3), -- Sales Manager 1

(7, 'Frank Green', 3), -- Sales Manager 2

(8, 'Grace Wilson', 4), -- Marketing Staff 1

(9, 'Harry Lee', 4), -- Marketing Staff 2

(10, 'Ivy Clark', 5), -- Marketing Staff 3

(11, 'Jack Turner', 5), -- Marketing Staff 4

(12, 'Kelly Martin', 6), -- Sales Staff 1

(13, 'Liam Hall', 6), -- Sales Staff 2

(14, 'Mia Adams', 7), -- Sales Staff 3

(15, 'Noah White', 7), -- Sales Staff 4

(16, 'Olivia Scott', 8), -- Marketing Intern 1

(17, 'Patrick Allen', 8), -- Marketing Intern 2

(18, 'Quinn Young', 9), -- Marketing Intern 3

(19, 'Rachel Harris', 10), -- Marketing Intern 4

(20, 'Samuel Turner', 12); -- Sales Intern 1

**Write the Recursive CTE:** To create a recursive CTE, you need to use the **WITH RECURSIVE** clause.

Here's a basic example that retrieves the hierarchy of employees starting from a given employee's ID:

WITH RECURSIVE EmployeeHierarchy AS (

SELECT id, name, manager\_id

FROM employees

WHERE id = 1 -- Replace 1 with the starting employee's ID

UNION ALL

SELECT e.id, e.name, e.manager\_id

FROM employees e

JOIN EmployeeHierarchy eh ON e.manager\_id = eh.id

)

SELECT \* FROM EmployeeHierarchy;

1. In this example, we first select the root employee (replace **1** with the desired starting ID).
2. Then, we use a **UNION ALL** clause to recursively join the **employees** table with the **EmployeeHierarchy** CTE using the **manager\_id** relationship.
3. **Execution:** When you execute the above SQL query, it will retrieve the hierarchy of employees starting from the specified employee ID.
4. The result will be a table that includes the selected employee and all of their subordinates in a hierarchical structure.
5. **Customize Output:** You can customize the output as needed. For instance, you can order the results by the employee's level in the hierarchy or format the output differently.

Here's a breakdown of the key components in the recursive CTE query:

* **WITH RECURSIVE**: This clause is used to indicate that you're defining a recursive CTE.
* **EmployeeHierarchy**: This is the name you give to the CTE. You can choose any name you like.
* The first part of the CTE is the base case, which selects the root node.
* The second part of the CTE uses a **UNION ALL** to recursively select child nodes by joining the **employees** table with the CTE itself.

Recursive CTEs in MySQL are a versatile way to work with hierarchical data, and you can adapt them to various use cases like organization charts, bill of materials, comment threads, and more.

Here's another example of a recursive CTE on the same **employees** table, but this time, we'll find all the subordinates of a specific employee:

Suppose you want to find all the subordinates of employee 'Jane Smith' (ID 2):

WITH RECURSIVE Subordinates AS (

SELECT id, name, manager\_id

FROM employees

WHERE name = 'Jane Smith' -- Replace with the desired employee's name

UNION ALL

SELECT e.id, e.name, e.manager\_id

FROM employees e

JOIN Subordinates s ON e.manager\_id = s.id

)

SELECT \* FROM Subordinates;

Explanation:

1. We define a recursive CTE named **Subordinates** using the **WITH RECURSIVE** clause.
2. In the initial part of the CTE, we select the employee with the name 'Jane Smith' (or any other employee whose subordinates you want to find).
3. In the recursive part, we join the **employees** table with the **Subordinates** CTE using the **manager\_id** relationship. This allows us to find all the employees who report directly or indirectly to 'Jane Smith.'
4. Finally, we select all the rows from the **Subordinates** CTE to retrieve the list of 'Jane Smith's subordinates.

When you run this query, it will return a result set containing all the employees who report directly or indirectly to 'Jane Smith' in a hierarchical structure. You can adjust the starting employee's name or ID to find subordinates for any other employee in the table.

Top of Form

Recursive Common Table Expressions (CTEs) in MySQL are a powerful feature used for querying and manipulating hierarchical data. They enable you to perform complex operations on data structures like trees, graphs, and organizational hierarchies that have a parent-child relationship within the same table. The primary use cases of recursive CTEs in MySQL include:

1. **Hierarchical Data**: Recursive CTEs are most commonly used to work with hierarchical data. This includes organizational charts, file systems, product categories, comment threads, and any data that can be represented as a tree structure.
2. **Traversal and Exploration**: You can use recursive CTEs to traverse and explore hierarchical data, allowing you to find all children, descendants, parents, ancestors, or the complete hierarchy of a specific node in the hierarchy.
3. **Data Reporting**: Recursive CTEs can be employed to generate hierarchical reports. For example, you can create an organizational chart that shows the entire hierarchy of employees in an organization or list all subcategories of a product category.
4. **Path Finding**: You can use recursive CTEs to find the path or route between two nodes in a hierarchy. This is helpful for navigation and determining relationships within the hierarchy.
5. **Data Transformation**: Recursive CTEs can be used to transform hierarchical data into a different format or structure that is more suitable for analysis, reporting, or presentation.
6. **Calculations and Aggregations**: You can perform calculations or aggregations on hierarchical data, such as calculating totals, sums, or averages for a specific branch of the hierarchy or for the entire tree.
7. **Querying Bill of Materials**: In manufacturing or product design, recursive CTEs are used to query bill of materials (BOM) structures to determine the components required to build a product and their subcomponents.
8. **Comment and Discussion Threads**: Recursive CTEs can be used to model and query comment or discussion threads, allowing you to display comments in a threaded or nested format.
9. **Access Control and Permissions**: In systems with role-based access control, recursive CTEs can be used to determine the permissions granted to a user by tracing their role hierarchy.
10. **Social Networks and Friendships**: In social network applications, recursive CTEs can help find connections between users, showing friends of friends or finding the shortest path between two users.

Recursive CTEs provide a structured and efficient way to work with hierarchical data without the need for extensive procedural code. They simplify complex queries and make it easier to retrieve, analyze, and manipulate hierarchical data in MySQL

Let's create a complex recursive CTE to find the entire hierarchy of employees in the **employees** table, including their managers and subordinates. We'll also calculate the depth of each employee in the hierarchy.

Here's the SQL query:

WITH RECURSIVE EmployeeHierarchy AS (

SELECT id, name, manager\_id, 0 AS depth

FROM employees

WHERE manager\_id IS NULL -- Start with the CEO (assuming CEO has no manager)

UNION ALL

SELECT e.id, e.name, e.manager\_id, eh.depth + 1

FROM employees e

JOIN EmployeeHierarchy eh ON e.manager\_id = eh.id

)

SELECT

eh.id,

eh.name,

eh.depth,

GROUP\_CONCAT(DISTINCT eh.manager\_id ORDER BY eh.depth DESC) AS path,

LPAD('', eh.depth, ' ') || eh.name AS indented\_name

FROM EmployeeHierarchy eh

GROUP BY eh.id

ORDER BY path;

Explanation:

1. We define a recursive CTE named **EmployeeHierarchy** using the **WITH RECURSIVE** clause. In the initial part, we select the CEO (assuming the CEO has no manager).
2. In the recursive part, we join the **employees** table with the **EmployeeHierarchy** CTE to find all employees reporting to a specific manager. We also calculate the depth of each employee in the hierarchy.
3. In the final **SELECT** statement:
   * We select the **id**, **name**, **depth**, and a concatenated **path** representing the hierarchy path for each employee.
   * We use the **GROUP\_CONCAT** function to concatenate all the manager IDs in the hierarchy path.
   * We create an **indented\_name** column to display employee names with indentation based on their depth in the hierarchy.
   * We group the results by **id** and order them by the **path** to display the hierarchy in a structured manner.

When you execute this query, it will return a result set showing the entire hierarchy of employees, including their names, depths, hierarchical paths, and indented names. This complex recursive CTE provides a detailed view of the organizational structure, making it easier to understand the hierarchy and relationships among employees

If you want to include the hierarchical levels (or generations) of employees in the recursive CTE, you can modify the query as follows:

WITH RECURSIVE EmployeeHierarchy AS (

SELECT id, name, manager\_id, 0 AS level

FROM employees

WHERE manager\_id IS NULL -- Start with the CEO (assuming CEO has no manager)

UNION ALL

SELECT e.id, e.name, e.manager\_id, eh.level + 1

FROM employees e

JOIN EmployeeHierarchy eh ON e.manager\_id = eh.id

)

SELECT

eh.id,

eh.name,

eh.level,

GROUP\_CONCAT(DISTINCT eh.manager\_id ORDER BY eh.level DESC) AS path,

LPAD('', eh.level \* 2, ' ') || eh.name AS indented\_name

FROM EmployeeHierarchy eh

GROUP BY eh.id

ORDER BY path;

In this query:

* We added a **level** column that starts at 0 for the CEO and increments by 1 for each subsequent level in the hierarchy.
* The **LPAD** function is used to create an **indented\_name** column that adds spaces based on the employee's level, making the output visually show the hierarchy levels.

The rest of the query remains the same, including the recursive CTE and the final **SELECT** statement for displaying the hierarchy path and indented names.

This query will return the entire hierarchy of employees with their respective levels, showing the organizational structure with appropriate indentation.

Here's another example of a recursive CTE on the **employees** table that includes levels in the hierarchy:

WITH RECURSIVE EmployeeHierarchy AS (

SELECT id, name, manager\_id, 0 AS level

FROM employees

WHERE manager\_id IS NULL -- Start with the CEO (assuming CEO has no manager)

UNION ALL

SELECT e.id, e.name, e.manager\_id, eh.level + 1

FROM employees e

JOIN EmployeeHierarchy eh ON e.manager\_id = eh.id

)

SELECT

eh.id,

eh.name,

eh.level,

CONCAT(REPEAT('--', eh.level), eh.name) AS indented\_name

FROM EmployeeHierarchy eh

ORDER BY id;

In this query:

* We added a **level** column to track the hierarchical levels, starting with 0 for the CEO and incrementing by 1 for each subsequent level.
* The **CONCAT** function is used to create an **indented\_name** column that visually represents the hierarchy by adding '--' characters based on the employee's level.
* We've removed the **GROUP\_CONCAT** and **path** from the final output to simplify the result set.

When you execute this query, it will return a result set showing the entire hierarchy of employees, including their levels and indented names. The **indented\_name** column visually represents the hierarchical structure, making it easier to understand the levels and relationships among employees.