# **FILE SYSTEM SIZE CALCULATION PROJECT REPORT**

**Using Dynamic Recursive CTE Approach**

**1. Project Overview**

The goal of this project is to calculate the total size of each folder in a hierarchical file system stored in a SQL database. The solution must:

* Handle nested subfolders at any depth.
* Sum the sizes of all contained files (direct and indirect).
* Return a structured report showing NodeID, NodeName, and TotalSizeBytes.

**Key Challenges**

* Hierarchical Data: Folders can contain subfolders, which may themselves contain more subfolders (recursive structure).
* Dynamic Depth: The solution must work regardless of how deep the folder nesting goes.
* Efficiency: Avoid multiple nested queries for performance.

2. Solution Design: Recursive CTE Approach

We use a Common Table Expression (CTE) with recursion to traverse the folder hierarchy and compute sizes dynamically.

Key Components :

1. Folder Hierarchy CTE – Identifies all folders and their relationships.

2. File Aggregation – Sums file sizes for each folder.

3. Final Calculation – Combines results for folders and files.

**3. Step-by-Step Implementation**

**Step 1: Database Setup**

CREATE TABLE FileSystem (

NodeID INT PRIMARY KEY,

NodeName VARCHAR(50),

ParentID INT,

SizeBytes INT

);

INSERT INTO FileSystem VALUES

(1, 'Documents', NULL, NULL),

(2, 'Pictures', NULL, NULL),

(3, 'File1.txt', 1, 500),

(4, 'Folder1', 1, NULL),

(5, 'Image.jpg', 2, 1200),

(6, 'Subfolder1', 4, NULL),

(7, 'File2.txt', 4, 750),

(8, 'File3.txt', 6, 300),

(9, 'Folder2', 2, NULL),

(10, 'File4.txt', 9, 250);

**Step 2: Recursive CTE Query**

WITH FolderHierarchy AS (

-- Base Case: Top-level folders (ParentID IS NULL)

SELECT

NodeID,

NodeName,

ParentID,

NodeID AS RootFolderID,

0 AS Level

FROM FileSystem

WHERE SizeBytes IS NULL -- Folders have NULL SizeBytes

UNION ALL

-- Recursive Case: Subfolders

SELECT

f.NodeID,

f.NodeName,

f.ParentID,

h.RootFolderID, -- Track the topmost parent

h.Level + 1 -- Track depth

FROM FileSystem f

INNER JOIN FolderHierarchy h ON f.ParentID = h.NodeID

WHERE f.SizeBytes IS NULL -- Only folders

),

FileSizes AS (

-- All files with their sizes

SELECT

NodeID,

NodeName,

ParentID,

SizeBytes

FROM FileSystem

WHERE SizeBytes IS NOT NULL -- Only files

),

FolderAggregates AS (

-- Sum all file sizes per folder hierarchy

SELECT

h.RootFolderID,

SUM(f.SizeBytes) AS TotalSizeBytes

FROM FolderHierarchy h

JOIN FileSizes f ON f.ParentID = h.NodeID

GROUP BY h.RootFolderID

)

**Final Output**

SELECT

fs.NodeID,

fs.NodeName,

CASE

WHEN fs.SizeBytes IS NULL THEN ISNULL(fa.TotalSizeBytes, 0) -- Folder size

ELSE fs.SizeBytes -- File size

END AS TotalSizeBytes

FROM FileSystem fs

LEFT JOIN FolderAggregates fa ON fs.NodeID = fa.RootFolderID

ORDER BY fs.NodeID;

1. How It Works
2. FolderHierarchy CTE

* Base Case: Starts with top-level folders (ParentID IS NULL).
* Recursive Part: Joins child folders to their parents, tracking RootFolderID (topmost parent) and Level (depth).

1. FileSizes CTE

* Extracts all files (non-NULL SizeBytes).

1. FolderAggregates CTE

* For each folder (RootFolderID), sums the sizes of all files in its hierarchy.

1. Final Query

* Combines folders and files.
* For folders, uses the aggregated size.
* For files, uses their direct size.

**Expected Output**

| **NodeID** | **NodeName** | **TotalSizeBytes** |
| --- | --- | --- |
| 1 | Documents | 1550 |
| 2 | Pictures | 1450 |
| 3 | File1.txt | 500 |
| 4 | Folder1 | 1050 |
| 5 | Image.jpg | 1200 |
| 6 | Subfolder1 | 300 |
| 7 | File2.txt | 750 |
| 8 | File3.txt | 300 |
| 9 | Folder2 | 250 |
| 10 | File4.txt | 250 |

**Explanation of Calculations**

* Documents (ID:1) = File1.txt (500) + Folder1 (1050) = 1550
  + Folder1 (ID:4) = File2.txt (750) + Subfolder1 (300) = 1050
    - Subfolder1 (ID:6) = File3.txt (300)
* Pictures (ID:2) = Image.jpg (1200) + Folder2 (250) = 1450
  + Folder2 (ID:9) = File4.txt (250)

**Advantages of This Approach**

* Handles Any Depth – Works for deeply nested folders.
* Dynamic Calculation – No hardcoding of folder levels.
* Efficient Aggregation – Uses GROUP BY for performance.
* Clear Structure – Separates logic into logical CTEs.

**Possible Improvements**

* Add Path Tracking

Modify FolderHierarchy to store the full path (e.g., Documents/Folder1/Subfolder1).

* Handle Circular References

Add a check to prevent infinite loops (e.g., if a folder points to itself).

* Optimize for Large Datasets

Use temporary tables for very large file systems.

**Conclusion**This dynamic recursive CTE approach efficiently computes folder sizes in a hierarchical file system stored in SQL. It is scalable, flexible, and maintainable, making it suitable for real-world applications.