Technologi Software Developer Test Report

At first glance, the database design solutions that came to my mind were:

1. Store the file system in a Relational Database.
    a. Create a self-referencing table as Node(id, name, parent_id).
    b. Create a table with paths as Node(id, name, path).
    c. Create two tables as Node(id, name) and Relation(parent, child).
2. Store the file system in a Graph Database like Neo4j. Edges in the graph represent the parent-child relation.

While researching about storing a file system structure in a database, I found out that the most important keyword is “hierarchical data”. This keyword has led me to valuable resources.

Possible Solutions

Adjacency List: Each entry knows its immediate parent. Operations like move, delete, rename, insert are easy. Also, it is efficient to query paths, thanks to recursive queries [1].

Path Enumeration (Materialized Path): Each entry stores the whole path to the root. This can be a good solution for searching directories, since only paths are required. However, it is hard to rearrange nodes, and modify the tree. Also, there is no referential integrity.

Nested Sets: Each directory stores two additional numbers that represent the range for subdirectories. Requires only one table. It is easy to find ancestors and descendants by looking at range representations. One drawback is that when we insert an entry into the database, roughly half of the other records need to be updated [2].

Closure Table: Store every path from each node to each of its descendants. This solution requires one additional reference table to keep all relations. Gives the ability to find all children of X, to depth N [3]. Takes up more space (O(n^2)) compared to other options

After inspecting the pros and cons of the solutions above, I decided to use the Adjacency List solution. Because it is a simple, easy to maintain, and efficient solution. If the database didn’t support recursive queries, I would use the Path Enumeration solution which is especially efficient for retrieving paths of directories.

Database Design

I used PostgreSQL as the DBMS, and created a table named Directory where each entry holds id, name, and parent_id (reference to the parent entry). I didn't create separate tables for file and folder, because both have the same attributes and to be able to use recursive query, every node should be in the same table. I also created a view named Path Mapping in order to make the queries look simpler in the application. Actually, the idea comes from the Path Enumeration solution. This view holds the name, id, and full path of each directory. Depending on the frequency of directory updates and directory searches, it might be better to use a materialized view. Definitions of Directory and Path Mapping are saved in the "init.sql" file.

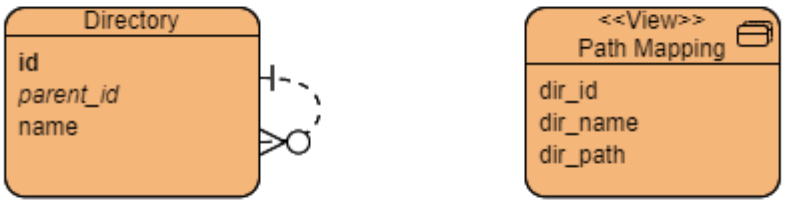

Figure 1 Entity Relationship Diagram

Web Application

It was a bit of a challenge as it was the first web application I have developed without using Spring framework. I created the web application using Java Servlets and JSP with Maven as the build system and Tomcat(9) as the web server. Database connection pool was established via the DataSource library. The process of reading initial data from a txt file is done right after server startup. Stack structure was used to read hierarchical data from the file. Unit tests are written using Junit and JWebUnit.

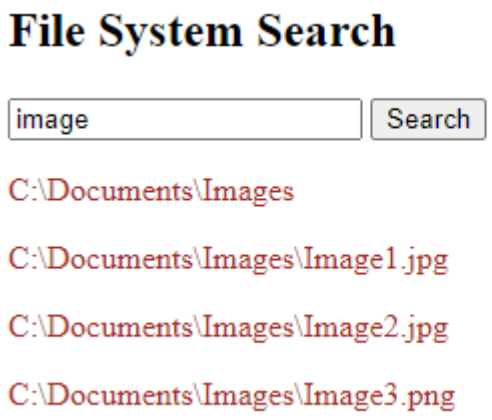

Figure 2 Web Interface

References

[1] "Adjacency List Vs. Nested Sets: Postgresql". 2009. *EXPLAIN EXTENDED*. https://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/.
[2] "Dbazine.Com: Trees In SQL: Nested Sets And Materialized Path". 2022. *Dbazine.Com*. http://www.dbazine.com/oracle/or-articles/tropashko4/.
[3] 2022. *Dirtsimple.Org*. https://dirtsimple.org/2010/11/simplest-way-to-do-tree-based-queries.html.