Tree Data Representation in Relational Databases

Chih-Jye Wang
Dept. of Computer Science and Software Engineering
Auburn University
Auburn, AL 36849, USA
wangchj@auburn.edu

ABSTRACT

1. INTRODUCTION

Hierarchical structures (or trees) frequently occur in information management that required to be stored and retrieved. Common examples of such structure include organization chart, in which positions of an organization are displayed as a tree, and hierarchical file system where files are stored in levels of directories.

Due to the popularity of relational database management systems (RDBMS), it is often required to store hierarchy data in such databases. The first apparent question is how to store hierarchical data in tables of RDMBS, which are great for storing a list of items without much design foresight for storing hierarchical data?

This paper explores ways of storing hierarchical data in relational databases. The goal is to define a few common operations of hierarchical data, the issues of implementing tree structure in relational model, and optimization for these operations.

2. DEFINITION

2.1 Tree

Hierarchical data structure is also known as a tree, which is defined as a connected acyclic digraph $G = (V, E)$ of a set of vertices V (also called nodes) and a set of edges E that denotes the relationship between two nodes[1]. Figure 1 depicts a typical tree.

Figure 1: Example of a tree.

Each node of the tree can hold any number of data item. In the figure above, only identifiers (A–J) of the nodes are shown for simplicity of illustration. The edges of the tree between two nodes are often named children (and sometimes parent) relationship. The relationship is often non-symmetric (as in digraph); for example, B is a child of A, but the reverse is not true.

In many literatures of computer data structures, the edges are defined as 'children of' a node. For example, node A has children B, C, and D. In the sense of relational database, it does not matter if the edges are defined as 'children of' or 'parent of'; when one is defined, the other is usually inherited, especially when indexes are used on the relational table in which the tree is stored. The relationship, therefore, will be left for the design of database schema.

[1]https://en.wikipedia.org/wiki/Tree_(data_structure)

2.2 Relational Database

Relational data model is proposed by Edgar Codd in 1970 [1]. In this model, a database consists of a set of tables (also known as relations), which each consists of a set of tuples. In colloquial usage, tuples are also called records.

Figure 2 illustrates a relational database with 2 tables, each 3 attributes. The teachers table has 4 records (tuples) and the Courses table has 6.

ID	Name	Office
1	James Madison	A
2	John Adams	B
3	Ronald Reagan	C
4	Franklin Pierce	D

(a) Teachers Table

ID	Name	Teacher
1	Data Structures	1
2	Intro to Algorithms	1
3	Software Construction	2
4	Computer Networks	3
5	Operating Systems	4
6	Compilers	2

(b) Courses Table

Figure 2: A Relational Database

2.3 Common Tree Operations

In this section, we define common operations on trees. We only define data reading operations due to the fact that these operations are the most commonly performed operations in database system after the initial insertion of data. Modification and deletion are less common and are usually trivial cases (compared to a myriad of important reading operations) to correct small mistakes. In addition, before updating or deletion, reading must be performed to locate the correct data to modify. With respect to hierarchical data, once an hierarchy is inserted into a database, the structure is rarely changed without removing an entire subtree.

We define the operations in this section without considering how the trees are implemented in relational model. Section 4 and 3 will cover implementation details and how these operations can be evaluated under a relational designs.

Operations thtat will be used in this paper are enumerated below. All examples use the tree of Figure 1.

1. $Root(n) \rightarrow r$: given a node n, this operation finds the root r of the tree in which n resides: $Member(n, r) \rightarrow true$.

$$Root(A) \rightarrow A$$
$$Root(E) \rightarrow A$$

2. $Parent(n) \rightarrow p$: Given a node n, this operation finds the parent node p, or $null$ if n is a root node.

$$Parent(A) \rightarrow null$$
$$Parent(B) \rightarrow A$$

3. $Children(n) \rightarrow \{c_1, c_2, ..., c_n\}$: Given a node n, this operation finds it's child nodes $\{c_1, c_2, ..., c_n\}$. If n is a leaf node (no children), then the empty set is returned.

$$Children(A) \rightarrow \{B, C, D\}$$
$$Children(E) \rightarrow \{\}$$

4. $Siblings(n) \rightarrow \{n_1, n_2, ..., n_n\}$: This operation finds a list of nodes $\{n_1, n_2, ..., n_n\}$ that have the same depth as node n. If this node has no siblings, the empty set is returned.

$$Siblings(A) \rightarrow \{\}$$
$$Siblings(B) \rightarrow \{C, D\}$$

5. $Leaves(n) \rightarrow \{n_1, n_2, ..., n_n\}$: This operation returns a set of leaf nodes $\{n_1, n_2, ..., n_n\}$ of the subtree rooted at n

$$Leaves(A) \rightarrow \{E, F, G, H, I, J\}$$
$$Leaves(B) \rightarrow \{E, F\}$$
$$Leaves(E) \rightarrow \{E\}$$

6. $Height(n) \rightarrow \mathbb{N}$: Given a node n, this operation returns the count of node of longest path from n to a leaf node.

$$Height(A) \rightarrow 3$$
$$Height(E) \rightarrow 1$$

7. $Depth(n) \rightarrow \mathbb{N}$: Given a node n, this returns $|Path(n)|$.

$$Depth(A) \rightarrow 1$$
$$Depth(E) \rightarrow 3$$

8. $Path(n) \rightarrow (n_1, n_2, ..., n_n)$: given a node n, this operation retrieves the sequence of nodes $(n_1, n_2, ..., n_n)$ that forms the path from root to node n, where n_1 is the root and n_n is n. This operation is the same as retrieving all *ancestors* of n.

$$Path(A) \rightarrow (A)$$
$$Path(E) \rightarrow (A, B, E)$$

9. $Member(n, t) \rightarrow boolean$: given two nodes n and t, this operation tests if node n is a *descendant* of node t – or, if n is in the subtree rooted at t.

$$Member(B, A) \rightarrow true$$
$$Member(A, B) \rightarrow false$$
$$Member(C, B) \rightarrow false$$

10. $Tree(n) \rightarrow (n_1, n_2, ..., n_n)$: given a node n, this operation returns a sequence of nodes $(n_1, n_2, ..., n_n)$ obtained by breadth-first traversal (BFT) from n, where $n_1 = n$. The original tree should be possible to be reconstructed from this sequence.

$$Tree(A) \rightarrow (A, B, C, D, E, F, G, H, I, J)$$
$$Tree(B) \rightarrow (B, E, F)$$

3. ADJACENCY LIST

One of the simplest way of representing hierarchy in relational data model is a method called adjacency list. In this approach, a table (we will call it Nodes table) is used to store the nodes of one or multiple trees, and each record of the table represent a node.

Each record of the Nodes table consists of field of primary key of the identifier of the node, and a field of foreign key that holds the identifier of the parent node, which references the same table. The root node of a tree would have *Parent = null*. The schema for the adjacency list table is shown in Listing 1.

Listing 1: Schema for adjacency list

```
1 CREATE TABLE Nodes (
2   ID CHAR PRIMAY KEY NULL,
3   Parent CHAR NOT NULL,
4   FOREIGN KEY (Parent) REFERENCES Nodes(ID)
5 )
```

Note: The name 'adjacency list' may cause a bit of confusion. An adjacency list implies a list of child (adjacent) nodes of a node, but the schema for the table above only has a reference from a node its parent node, which should probably more aptly named 'parent-link' approach.

As noted in section 2.1, this distinction is actually not important in relation database. The reason is that with a reference to parent defined, the *children* of a node (adjacency list) can be easily found. For example the child nodes of A can be by found using the query in Listing 2:

Listing 2: SQL for querying child nodes.

```
1 SELECT * FROM Nodes WHERE Parent='A'
```

3.1 Operations

In this section, we present how the operations defined in Section 2.3 can be implemented in the context the adjacency list model. For operations that involve simple queries, we provide only the queries; for operations that requires more complex operations (e.g. loops), we present the algorithm in Java programming language.

3.1.1 Root Operation

If the Nodes table contains only a single hierarchy, the operation of finding the root of an arbitrary node in the tree would be a trivial query of looking for the record whose Parent attribute is null. This is *not* the case for us. We assume that the Nodes table contains records for multiple trees. This complicate the query quite a bit and requires us to use some form of iteration (or recursion) to traverse to the top of the tree (until we reach a node with Parent = null). This procedure makes the time-complexity of operation $O(d_n)$. Listing 3 shows the code for this operation.

Listing 3: Root for Adjacency List

```
1  /** Returns the first node after executing the
       query.*/
2  static Node getOneNode(String query){...}
3
4  /** Returns the root node of node n. */
5  static Node Root(Node n)
6  {
7      if(n.Parent == null)
8          return n;
9      Node parent = getOneNode("SELECT * FROM
           Nodes WHERE ID='" + n.ID + "'");
10     return Root(parent);
11 }
```

3.1.2 Parent, Children, Siblings Operations

In the adjacency list model, these operations are trivial and can be implemented via a short query (Listing 4). The Parent operation (line 6) is trivial because every node contains a foreign key that points to the primary key of the parent node. Since the primary key indicates the exact location of a record in the database, retriving a parent node is $O(1)$ operation.

For the children (line 9) and siblings (line 12) operations, if the 'parent' field is indexed using B-Tree [2], then the time-complexity of this query is $O(\log |N|)$, where $|N|$ is the total number of rows in the Nodes table. If the field is indexed using a hash index, the time-complexity for these becomes $O(1)$.

Listing 4: Parent, Children, and Sibling

```
1  --
2  -- Given a node $n$ and n.ID='E'
3  --
4
5  -- Get the parent of node E
6  SELECT * FROM Nodes WHERE ID=(SELECT Parent
       FROM Nodes WHERE ID='E')
7
8  -- Get the children of node E
9  SELECT * FROM Nodes WHERE Parent='E'
10
11 -- Get the siblings of node E
12 SELECT * FROM Nodes WHERE Parent=(SELECT
       Parent FROM Nodes WHERE ID='E') AND ID !=
       'E'
```

3.1.3 Leaves, Height, Depth, Path, Member Operations

The algorithms for finding leaves and height of a tree will be similar. Given a node, both algorithms iteratively or recursively perform the same operation on the children of the node until the iteration reaches a leaf node. For the Leaves operation, when the recursion reaches a leaf node, the node is added to the result and recursion ends. For the Height operation, when a leaf node is reached, the algorithm returns 1. The parent call returns $1 + \max(\text{recursive calls})$.

Note that for every node in the tree, a call to Leaves() is made. This made the time complexity of these operations $O(|N|)$. This is very inefficient. Considering a tree with 10,000 nodes, which will result in 10,000 queries being made to the database.

We only show the algorithm for Leaves in Listing 5.

Listing 5: Leaves

```
1  /** Returns an array of leaf nodes under n. */
2  static Node[] Leaves(Node n)
3  {
4      ArrayList<Node> result = new ArrayList<
           Node>();
5      Leaves(n, result);
6      return result.toArray();
7  }
8
9  /** Recursive method.*/
10 static void Leaves(Node n, List<Node> result)
11 {
12     Node[] children = getChildren(n);
13     if(children.length == 0)
14     {
15         result.add(n);
16         return;
17     }
18     for(Node child : children)
19         Leaves(child, result);
20 }
```

In addition, the algorithm for Depth, Path, and Member are very similar to the algorithm for finding root. Each algorithm recursively (or iteratively) reaches for the root of the tree while keeping state. For example, the algorithm for depth has to keep track of node count during the iterations. The complexity is the same as finding the root, which is $O(d_n)$ which is proportional to $O(\log |N|)$.

For the algorithm for Memeber, the iteration can stop once the root node of the subtree t is reached. The time-complexity of the implementation is $O(d_n)$. The algorithm is given in Listing 6.

Listing 6: Member

```
1  static boolean Member(Node n, Node t)
2  {
3      while(n != null)
4      {
5          if(n.equals(t))  //n = t
6              return true;
7          n = getParent(n);
8      }
9      return false;
10 }
```

3.1.4 Tree Operation

Listing 7 shows the algorithm to get the BST of tree in this relational model.

Listing 7: Tree via Adjacency List

```
1  static Node[] Tree(Node n)
2  {
3      Queue<Node> q = new LinkedList<Node>();
4      ArrayList<Node> result = new ArrayList<
           Node>();
5      q.add(n); //enqueue
6      result.add(n);
7      while(!q.isEmpty())
8      {
```

```
9           Node head = q.remove();
10          Node[] ch = getChildren(head);
11          for(Node c : ch)
12          {
13              result.add(c);
14              q.add(c);
15          }
16      }
17      return result.toArray();
18  }
```

After examining these implementations, we come to the conclusion that the operations are trivial in the adjacency list model are: *1)* parent, *2)* children, *3)* siblings.

All other operations require iterations (or recursions): *1)* root, *2)* leaves, *3)* height, *4)* depth, *5)* path, *6)* member, *7)* tree. The number of queries of these operations usually depend on the depth of the node in question. Each query submitted to the database incurs communication cost between the processing program and the database; therefore, the cost of computing these operations increases as the depth of the tree increase. The time-complexity of this implementation is $O(|N|)$.

4. NESTED SETS

Another way of representing hierarchical data in relational database is the nested set model [3]. This model is an extension of the adjacency list model. The goal of this model is to eliminate the need to issue multiple queries to the database (and the use of recursion) by placing ordering on the nodes. Listing 8 shows the relational schema for the nested set model.

Listing 8: Nodes table for nested set model

```
1  CREATE TABLE Nodes (
2      ID CHAR PRIMARY KEY NOT NULL,
3      Parent CHAR NULL,
4      L INT NOT NULL,
5      R INT NOT NULL,
6      FOREIGN KEY (Parent) REFERENCES Nodes (ID)
7  )
```

In this model, there are, for each node, two additional fields: L and R. These two fields stand for left and right. These two values provides an ordering of nodes in a tree. Algorithm for assigning L and R value for nodes is shown in Listing 9.

Listing 9: Algorithm for Left-Right value assignment for nested set model

```
1  int AssignLR(Node node)
2  {
3      return AssignLR(node, 1);
4  }
5
6  /** Recursively assign LR for each node */
7  int AssignLR(Node node, int count)
8  {
9    node.L = count;
10   Node[] children = getChildren(node);
11   for(Node child : children)
12     count = AssignLR(child, count + 1)
13   node.R = ++count;
14   return count;
15 }
```

The heart of the algorithm in Listing 9 is the recursive version (line 6–15) of $AssignLR()$ function. This function takes 2 parameters: $node$, which is the root node of a tree, and $count$ which is a variable to keep track of the current LR value. The $count$ is initially set to 1, so that L of the root is set to 1.

The key points the algorithm is given below:

1. Initially, the function sets L of the current node as the initial value of $count$ that is passed in (line 9). For example, if $node$ is the root node, and initial $count = 1$, then the left value of the root will be 1.

2. For each child of $node$, perform depth-first traversal (DFT) by recursively call $Asssign()$ function (line 11–12). The calls traverse the tree (down-trip) from the root the leaf nodes, and at every call $count$ is incremented.

3. On the up-trip (line 13), R of $node$ is assigned $count + 1$. If node is leaf, then R is assigned, $L + 1$. Else, R is assigned R of last child + 1.

Figure 3 shows the left and right values for the tree in the Figure 1 after executing $AssignLR()$.

Figure 3: Example of a tree in nested set model.

The left and right (LR) values puts an order on the nodes in the database and establish the following property.

PROPERTY 4.1. *Given two nodes a and d, d is a descendant of a if $(a.L < d.L)$ and $(a.R > d.R)$.*

4.1 Operations

In this section we present how the operations defined can be implemented in the nested sets model. Due to the fact that the nested sets model is an extension of the adjacency list model, implementation of all operations still work in this model; however, the performance (in terms of time of query) of some operations can be improved taking advantage of the left and right values provided by this model.

The simple operations in the previous model cannot be improved further. These operations are Root, Parent, Children and Siblings. We use the same code as provided earlier.

Other operations we can improve efficiency using this model. In previous section, we use iterations (loops or recursions) to realize some of the operations, where in each iteration, one query is submitted to the database. Using nested sets model, in most cases however, we can reduce iterations used in the adjacency model to a single query. The operations we can improve include Leaves, Height, Depth, Path, Member, and Tree.

4.1.1 Leaves Operation

The Leaves operation no longer require traversal to the bottom of the tree and more importantly required to visit every node in the tree. With the structural information provided by the LR values, we can find the leaf nodes by looking for nodes with the correct LR boundaries that do not have child node. Listing 10 shows the query to accomplish, given an sample input node of $\{ID : B, L : 2, R : 7\}$.

Listing 10: Nested Sets Leaves Operation

```
1  SELECT * from Nodes N WHERE L > 2 AND R < 7
       AND NOT EXIST (SELECT * FROM Nodes WHERE
       Parent=N.ID)
```

The illustrated query is a correlated query[2]. The outer query (select) simply limits the selected nodes to be the descendants of input node. For each selected node in the outer query, the subquery checks if there is any child node.

The outer query can be accomplished through two index scans, with the time-complexity of $O(2\log|N|)$. The inner query is the same as the children operation discussed in section 3.1.2, which in worst case is also $O(\log|N|)$. With the total complexity of $O(2\log^2|N|)$, which has a growth rate similar to $O(\log|N|)$.

4.1.2 Path, Depth, Member Operations

These operations can be directly obtained from Property 4.1. Path is simply a list of ancestors, which we can easily obtain, and the query node itself. Similarly, Depth is the count of the number of records returned from Path Given an input node of $\{ID : E, L : 3, R : 4\}$, Listing 11 shows the query for Path(line 1-2), and Depth(line 4-5).

Listing 11: Path, Depth, Member Operations

```
1  -- Path Query
2  SELECT * FROM Nodes WHERE L <= 3 AND R >= 4
       ORDER BY L
3
4  -- Depth Query
5  SELECT COUNT(*) FROM Nodes WHERE L <= 3 AND R
       >= 4
```

4.1.3 Member Operation

Member match exactly the definition of Property 4.1. Listing 12 shows a trivial code for this operation.

Listing 12: Member Operation

```
1  static boolean Member(Node n, Node t)
2  {
3      return (t.L < n.L) && (t.R > n.R);
4  }
```

With nested set model, finding ancestors of a node becomes trivial. Because the left-value of a node is assigned before any child-node is traversed and the right-value after, any *ancestor*, denoted by a, of a node n has the following properties:

1. $a.left < n.left$
2. $n.right < a.right$

Note: "left" and "right" are SQL keywords, consider changing left to another name.

With the above properties, the queries for ancestors no longer require multiple queries (or recursion). Listing 13 shows the query to get ancestors of node E in Figure 3.

Listing 13: Query for ancestors for node E via nested set model

```
1  SELECT * FROM Nested_Set WHERE Left < 3 AND
       Right > 4
```

4.1.4 Tree

Getting the entire hierarchy of a tree is the same as getting all the descendants of the root node (including the root). If the database contains only one tree, then this operation involves retrieving all entries in the nodes table. However, we assume that the database contains multiple disconnected trees.

Listing 14 shows this query for the tree in Figure 3, in which the left and right value for the root node A is 1 and 20, respectively.

Listing 14: Query for Tree for tree A via nested set model

```
1  SELECT * FROM Nested_Set WHERE Left > 1 AND
       Right < 20 ORDER BY left ASC
```

To reconstruct the hierarchical structure from the flat list returned from the query above, we look at the result of the query (of the tree in Figure 14) in the Table 1 below.

ID	Left	Right	Parent
A	1	20	null
B	2	7	A
E	3	4	B
F	5	6	B
C	8	13	A
G	9	10	C
H	11	12	C
D	14	19	A
I	15	16	D
J	17	18	D

Table 1: Query result of Tree operation

There are two ways to reconstruct the tree, since there are redundant information in the result table to reconstruct the tree. The first way is to only use the 'Parent' column. The second way is to use only the 'Left' and 'Right' columns (not needing the 'Parent' column). Listing 15 shows the algorithm for the first approach.

Listing 15: Tree reconstruction using Parent

```
1  Node ReconstructTree(List<Record> records)
2  {
3      Node root, current = null;
4      for(Record record : records)
5      {
6          if(current == null)
7          {
8              current = MakeNode(record);
9              root = current;
10         }
11         else
12         {
13             while(current.ID != record["Parent
                   "])
14                 current = current.Parent;
15
16             Node node = MakeNode(record);
17             current.Children.Add(node);
18             current = node;
19         }
20     }
21   return root;
22 }
```

Listing 16 shows the algorithm for the second approach.

Listing 16: Tree reconstruction using Left and Right value

```
1  //Returns the root node as the tree
2  Node ReconstructTree(List<Record> records)
3  {
4      Node root, current = null;
5      for(Record record : records)
6      {
```

[2]http://en.wikipedia.org/wiki/Correlated_subquery

```
        if(current == null) //This should only
             be run once
        {
            current = MakeNode(record);
            root = current;
        }
        else
    {
            while(!(current.Left < record["
                Left"] and current.Right >
                record["Right"]))
                current = current.Parent;

            Node node = MakeNode(record);
            current.Children.Add(node);
            current = node;
        }
    }
    return root;
}
```

4.1.5 *Print Tree*

The tree structure can be easily outputted by a program after the hierarchy has been reconstructed with the algorithms in the previous section. However, sometimes a tree has large number of nodes and we may only need to output the tree structure without needing to reconstruct the tree in memory.

This section shows an algorithm to print the ID of the nodes with increasing indention to show the hierarchy of the tree.

Listing 17: Tree reconstruction using Left and Right value

```
void PrintTree(List<Record> records)
  //Make new stack to hold right values;
  //length = 0
  Stack stack;
  foreach(Record record in records)
    while(stack.top < record["right"] and !
        stack.isEmpty)
      stack.pop()
    //Print identation
    for(int i = 0 to stack.length - 1)
      Print(" ")
    //Print current record
    Print(Record.ID)
    stack.push(record["right"])
```

5. REFERENCES

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[2] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[3] M. Kamfonas. Recursive hierarchies: The relational taboo! *The*