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.

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)

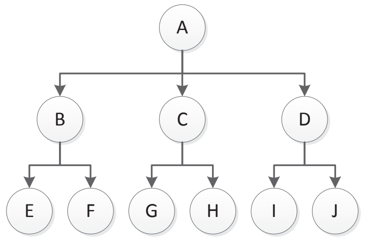


Figure 1: Example of a tree.

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

$$Root(A) \to A$$
$$Root(E) \to A$$

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

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

3. $Children(n) \to \{c_1, c_2, ..., c_n\}$: 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.

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

4. $Siblings(n) \to \{n_1, n_2, ..., n_n\}$: 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.

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

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

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

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

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

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

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

Listing 1: Schema for adjacency list

```sql
CREATE TABLE Nodes (
  ID CHAR PRIMAY KEY NULL,
  Parent CHAR NOT NULL,
  FOREIGN KEY (Parent) REFERENCES Nodes(ID)
)
```

8. $Path(n) \to (n_1, n_2, ..., n_n)$: 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.

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

9. $Member(n, t) \to boolean$: 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.

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

10. $Tree(n) \to (n_1, n_2, ..., n_n)$: 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.

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

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.

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.

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

Listing 3: Root for Adjacency List

```java
/** Returns the first node after executing the
     query.*/
static Node getOneNode(String query){...}

/** Returns the root node of node n. */
static Node Root(Node n)
{
    if(n.Parent == null)
        return n;
    Node parent = getOneNode("SELECT * FROM
        Nodes WHERE ID='" + n.ID + "'");
    return Root(parent);
}
```

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.

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 $O(1)$.

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.

Listing 4: Parent, Children, and Sibling

```sql
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'
```

Listing 5: Leaves

```java
/** Returns an array of leaf nodes under n. */
static Node[] Leaves(Node n)
{
    ArrayList<Node> result = new ArrayList<
        Node>();
    Leaves(n, result);
    return result.toArray();
}

/** Recursive method.*/
static void Leaves(Node n, List<Node> result)
{
    Node[] children = getChildren(n);
    if(children.length == 0)
    {
        result.add(n);
        return;
    }
    for(Node child : children)
        Leaves(child, result);
}
```

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.

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.

3.1.4 Tree Operation

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

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 num-

Listing 6: Member

```java
static boolean Member(Node n, Node t)
{
    while(n != null)
    {
        if(n.equals(t))  //n = t
            return true;
        n = getParent(n);
    }
    return false;
}
```

Listing 7: Tree via Adjacency List

```java
static Node[] Tree(Node n)
{
    Queue<Node> q = new LinkedList<Node>();
    ArrayList<Node> result = new ArrayList<
        Node>();
    q.add(n); //enqueue
    result.add(n);
    while(!q.isEmpty())
    {
        Node head = q.remove();
        Node[] ch = getChildren(head);
        for(Node c : ch)
        {
            result.add(c);
            q.add(c);
        }
    }
    return result.toArray();
}
```

Listing 8: Nodes table for nested set model

```sql
CREATE TABLE Nodes (
    ID CHAR PRIMARY KEY NOT NULL,
    Parent CHAR NULL,
    L INT NOT NULL,
    R INT NOT NULL,
    FOREIGN KEY (Parent) REFERENCES Nodes (ID)
)
```

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

```java
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 }
```

ber 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.

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.

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 $Assign()$ function (line 11–12). The calls traverse the tree (down-trip) from the root the leaf nodes, and at every call $count$ is incremented.

Listing 10: Nested Sets Leaves Operation

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

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 $AssignLR()$.

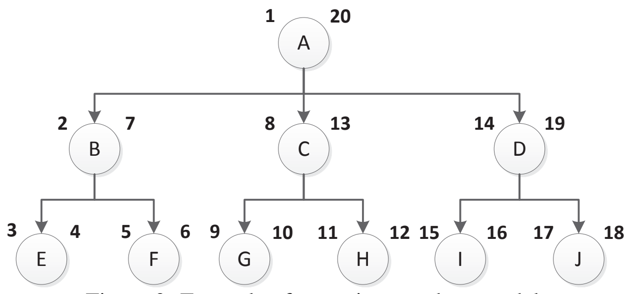


Figure 3: Example of a tree in nested set model.

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 $\{ID : B, L : 2, R : 7\}$.

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.

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

Listing 11: Path, Depth, Member Operations

```sql
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
```

Listing 12: Member Operation

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

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

In addition to the implementation illustrated in Section 3.1.3, we can also evaluate these operations using the nested sets model, which 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).

The implementation presented in Section 3.1.3 (using pure adjacency list) and the implementation presented in this section (using LR-values) both have the time complexity of $O(\log |N|)$. For the pure adjacency list implementation, this is easy to see because the number of nodes to search is simply the depth, which is proportional to $log\ |N|$. As for the implementation in this section, the query evaluator will have to traverse the index tree for the L and R fields to find the qualifying nodes, which also produce log time-complexity.

Though both produce similar analytical time-complexity, there are differences between the two approaches. The first difference is that the query presented in this section performances the entire evaluation on the database server and sends the final result back to the client; whereas, the logic of the approach of Section 3.1.3 is located on the client, therefore, the server has to send intermediate results back to the client. In other words, the adjacency list implementation requires multiple queries and and results sent between the client and the server. The second difference is that the nested set approach has to traverse the index tree for both R and L fields. The adjacency list approach does not have this complexity.

4.1.3 Member Operation

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

The time complexity of this operation is $O(1)$.

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

Listing 13: Query for Tree for tree A via nested set model

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

Listing 14: Tree reconstruction using Parent

```
Node ReconstructTree(List<Record> records)
{
    Node root, current = null;
    for(Record record : records)
    {
        if(current == null)
        {
            current = MakeNode(record);
            root = current;
        }
        else
        {
            while(current.ID != record["Parent
                "])
                current = current.Parent;

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

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 13 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.

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 13) 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 14 shows the algorithm for the first approach.

Listing 15 shows the algorithm for the second approach.

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 15: Tree reconstruction using Left and Right value

```
//Returns the root node as the tree
Node ReconstructTree(List<Record> records)
{
    Node root, current = null;
    for(Record record : records)
    {
        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;
}
```

Listing 16: 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 indentation
    for(int i = 0 to stack.length - 1)
      Print(" ")
    //Print current record
    Print(Record.ID)
    stack.push(record["right"])
```

5. EVALUATION

This section presents the performance evaluation of the adjacency list and nested sets models presented in section 3 and 4, respectively. Performance are evaluated based on the running time of the operations described in section 2.3, under each data model.

The evaluation is composed of two parts: the server and the client. The server runs an instance of MySQL database and holds hierarchical structure and data. The client connects to the server via local area network, implements the operations, and reports the amount of time to run them.

This section is organized as follows. Section 5.1 and section 5.2 describe the environment and configuration of the server and the client.

5.1 Server Configuration

Table 2 shows the software and hardware configuration of the server machine.

CPU	Intel Processor 64-bit
Memory	4 GB
Operating System	Ubuntu 13.10
Database System	MySQL 5.5 Community Edition

Table 2: Dataset size

In addition, when running the database system, the following settings are made:

Variable	Set Value	Default Value
max_sp_recursion_depth	100	0
maximum_stack memory	100	100
some other memory	100	100

Table 3: Database Server Non-default Settings

Note: briefly describe the reason for above settings.

5.2 Client Configuration

The client machine is responsible for running the experiment code, which is written in Java and connected to the database using MySQL JDBD Connector[3]. Table 4 lists configurations of the client environment.

CPU	Intel Processor 64-bit
Memory	12 GB
Operating System	Ubuntu 13.10 Desktop
Programming Language	Java 7 (OpenJDK 7)
Database Connector	JDBC mysql-connector-java-5.1.29

Table 4: Dataset size

At early stages of coding the experiments, the author repeatedly getting Out Of Memory Error from the Java Virtual Machine. The code for the experiments have been rewritten several times to reduce memory consumption. The error can also be mitigated by expanding the allocated memory to the program. The following are the command sets the minimum and maximum heap memory allocated to the experiment to 8 and 9 gigabytes.

5.3 Data Format and Generation

The experimental datasets are synthetically generated relational hierarchies, each containing the fields ID, Parent, L and R as described for the Nodes table in Listing 8. In addition, each record (an entry in data table) also contains additional 90 ASCII characters, divided into two fields (45 characters each), to simulate data payload in typical database record. However, these additional data is currently not being used in experiments, due to the limitation of amount of memory in machines that run the experiments. The structure of the experimental data shown in Table 5.

The datasets are created by our data generation SQL script (see appendix) and stored on in the database on the server. In addition to being a full tree[4], each dataset is characterized by the following properties:

1. Order: the number of children per node. This is also known as the fan-out or out-degree of a tree.

[3]http://www.oracle.com/technetwork/java/javase/jdbc/index.html
[4]http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees

Listing 17: Nodes table for nested set model

```
java -cp lib/mysql-connector-java-5.1.29-bin.
    jar:. -Xms8g -Xmx9g Experiment
```

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
id	int(11)	NO	PRI	NULL	auto_increment
parent	int(11)	YES	MUL	NULL	
l	int(11)	NO	MUL	NULL	
r	int(11)	NO	MUL	NULL	
data1	varchar(45)	NO		NULL	
data2	varchar(45)	NO		NULL	

Table 5: Experimental data relational format

2. Height: the number of nodes of the longest path from root to a leaf node.

Each of the properties ranges from 2 to 10, with the exception that higher-order datasets have lower max height. For each combination of order and height, a dataset is generated and stored in a database table. Table 6 shows the size (number of records) of each data table.

		Height						
		2	**3**	**4**	**5**	**6**	**7**	**8**
Order	**2**	3	7	15	31	63	127	255
	3	4	13	40	121	364	1,093	3,280
	4	5	21	85	341	1,365	5,461	21,845
	5	6	31	156	781	3,906	19,531	97,656
	6	7	43	259	1,555	9,331	55,987	335,923
	7	8	57	400	2,801	19,608	137,257	960,800
	8	9	73	585	4,681	37,449	299,593	2,396,745
	9	10	91	820	7,381	66,430	597,871	5,380,840
	10	11	111	1,111	11,111	111,111	1,111,111	11,111,111

Table 6: Test data size with respect to Order and Height

With the server configuration described earlier in this section, an estimate of the amount of time to generate the data sets is shown in Table 7.

5.4 Experimental Result

The operations defined in Section 2.3 are divided into three categories based on their running time complexity which are $O(logN)$, $O(N)$, and $O(1)$. Members in a category have the similar running time; therefore we have to show the performance for one operation per group. The membership of each category shows as follows:

1. $O(logN)$: Root, Depth, Path, Member
2. $O(N)$: Heaves, Height, Tree
3. $O(1)$: Parent, Children, Siblings

In this section, we report the performance of category 1 and 2. Category 3 is not reported since the input size does not affect the outcome and that the implementation are the same for both adjacency-list and nested-sets model.

5.4.1 *Category 1 - O(log N)*

Figure 4 depicts the running time in milliseconds of the finding root operation for the adjacency list model. From this result, we can rough infer that the time complexity of this algorithm is linear

		Height					
		2	3	4	5	6	
Order	2	200 ms	470 ms	470 ms	2.07 sec	4.2 sec	8.
	3	270 ms	870 ms	2.67 sec	8.07 sec	24.27 sec	1.2
	4	330 ms	1.4 sec	5.67 sec	22.73 sec	1.52 min	6.0
	5	400 ms	2.07 sec	10.4 sec	52.07 sec	4.34 min	21
	6	470 ms	2.86 sec	17.27 sec	1.73 min	10.37 min	1.
	7	530 ms	3.8 sec	26.67 sec	3.11 min	21.79 min	2.
	8	600 ms	4.87 sec	39 sec	5.20 min	41.61 min	5.
	9	670 ms	6.07 sec	54.67 sec	8.20 min	1.23 hr	11
	10	730 ms	7.4 sec	1.23 min	12.35 min	2.06 hr	20

Table 7: Experimental data generation time estimate

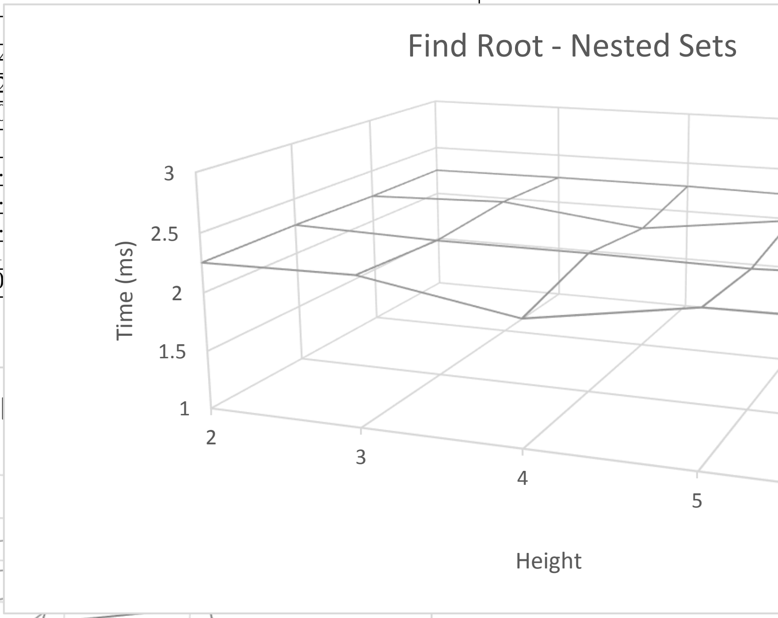


Figure 5: Running time using Nested Sets Model.

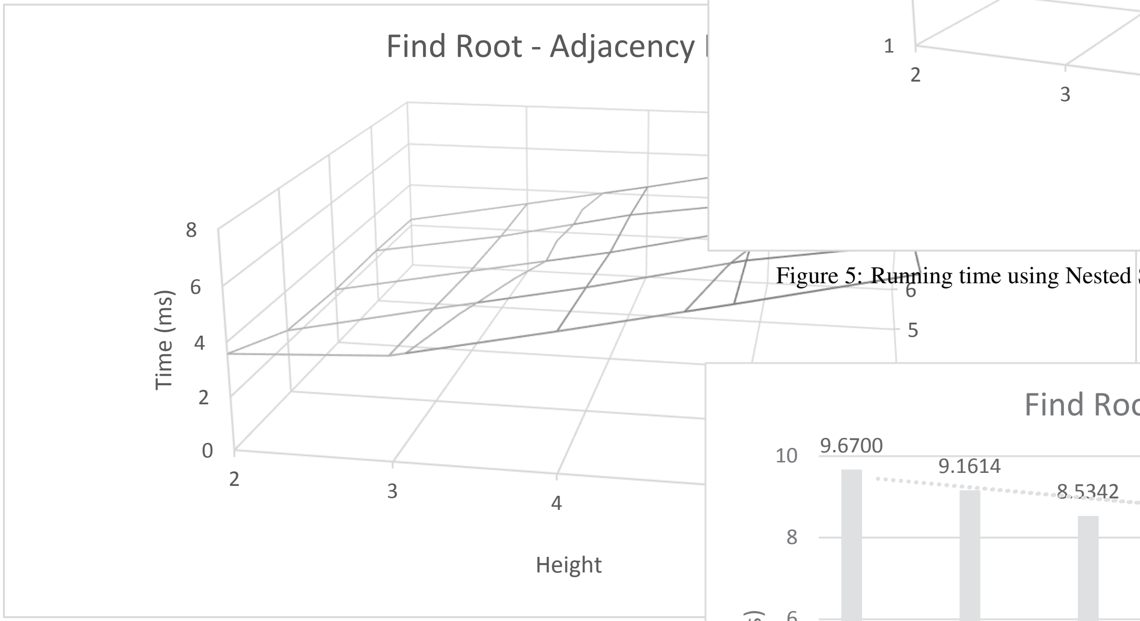


Figure 4: Running time using Adjacency List Model.

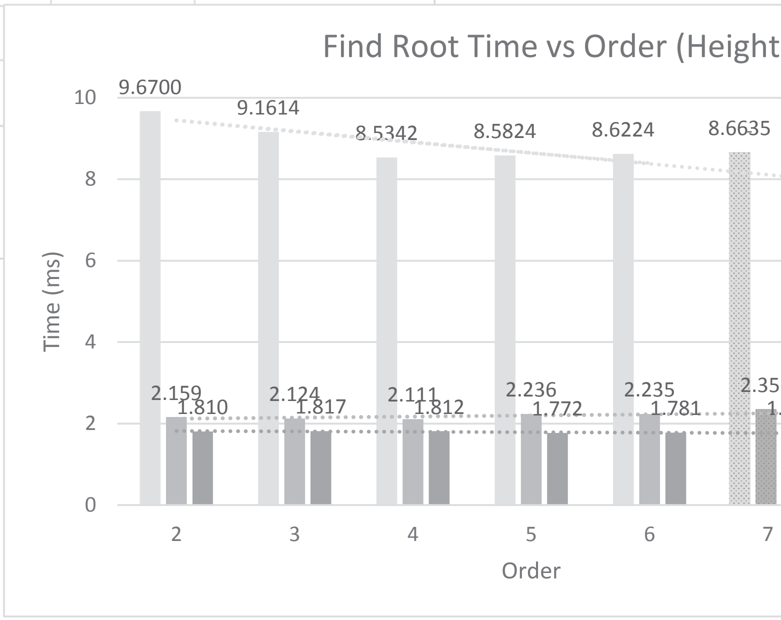


Figure 6: Running time vs node out-degree (order) at height = 8.

with respect to the depth, and constant with respect to the width (order) of the tree.

Figure 5 depicts the running time in milliseconds of the finding root operation for the nested sets model. One can easily see that it is an improvement over the adjacency list model. In this result, the time complexity with increasing depth of the tree has reduced to constant. The time complexity with increasing width stayed constant, with slight constant improvement.

If we take a slice of the previous two figures down where height = 8, we can see the growth of the algorithm with respect to order as shown in Figure 6. The figures shows, as expected, that the growth of the order of the tree has little effect on the running time of the operation.

For the adjacency list model, the trend appears to be decreasing. However if we look at the running time from order 4 to 8, the trend is flat. I believe that the initial decreasing trend is caused by database warm-up. As more data are cached, the database system is able to reach stable-state.

In addition to the two described models, the figure shows the growth trend for adjacency list model implemented using MySQL stored procedure (labeled Stored). The purpose of the stored implementation is to see if stored procedures facilitates performance. In this case, the store version is better than both adjacency list and nested sets models.

In the figure, the trend line is draw using linear regression from the data from order 1 to 6. The measurement of order 7 and 8 are draw to verify the correctness of the growth model.

In the same manner, if we take a slice of Figure 4 and 5 down where order = 8, we can see the growth of the algorithm with respect to height as shown in Figure 7.

The figure shows that for adjacency list model, the running time grows linearly with respect to height. In contract, the running time for the nested sets model shows big improvement over adjacency list, with almost perfect flat trend line model, and indicates constant time-complexity. Once again, the data for the stored model shows

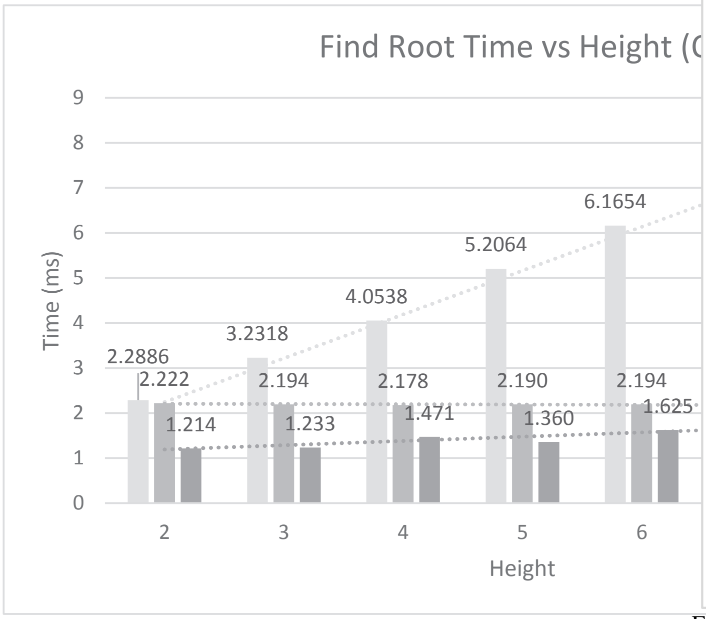


Figure 7: Running time vs tree height at order = 8.

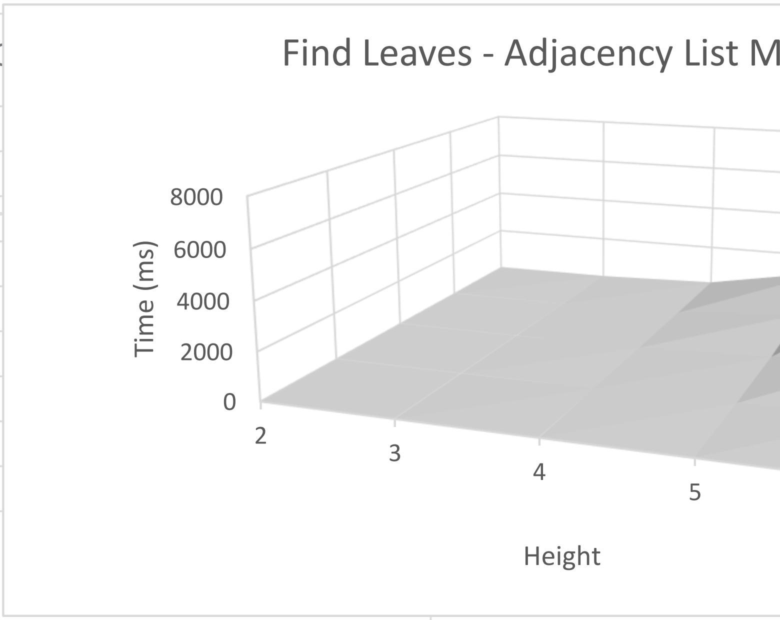


Figure 8: Running time of Finding Leaves using Adjacency List Model.

that the running time are better than both other models; however, the trend shows a linear growth rate, which is not not better than nested sets model larger data set.

5.4.2 Category 2 - O(N)

Figure 8 depicts the running time, in milliseconds, of the finding root operation using the Adjacency List Model. The running time stayed relative flat, below 100 ms for height and order, below 4.

Above 4, the figure shows that the running time appears to be growing exponentially with respect to both dimensions. The exponential growth with respect to height is expected because the operation has $O(N)$ complexity and each increase of height increase N by a factor of w, where N is the number of nodes and w is the order (or width) of the hierarchy.

However, the exponential growth with respect to the order w in the figure is not expected. Increase in the width of the hierarchy only linearly increase the number of node N. This is likely caused by additional processing (such as index tree traversal) required by the database system that makes the cost of all nodes higher than $O(N)$.

Figure 9 depicts the running time, in milliseconds, of the operation using Nested Sets Model. Careful reader will find that the figure looks almost the same as Figure 8. Although the two figures have the same shape, the running time for the Nested Sets Model is much lower. Taking height = 6 and order = 6 for example, the running time Adjacency List Model is around 7,000 ms, but Nested Sets Model around 9 ms.

Figure 9 depicts the running time, in milliseconds, of the operation implemented in MySQL stored procedure. The figure has the same exponential trend as the previous two models; however, the running time using this technique is unreasonably high. For example, with height = 4 and order = 2 (N = 15), the running time is 1831 ms (almost 2 seconds) comparing with 13 ms and 1.15 ms of other two models, for very small data size. For this reason, we will no longer compare the running time of this technique to the other two.

It is interesting to reflect on this a bit. As a reminder, this technique is the Adjacency List version of the algorithm implemented in MySQL stored procedure. That means the algorithm is recursive, and we implemented a stored recursive stored procedure that is called for every node in the hierarchy. The take away here is recursive stored procedure using MySQL is possible, but it is extremely inefficient, to which we attribute this performance.

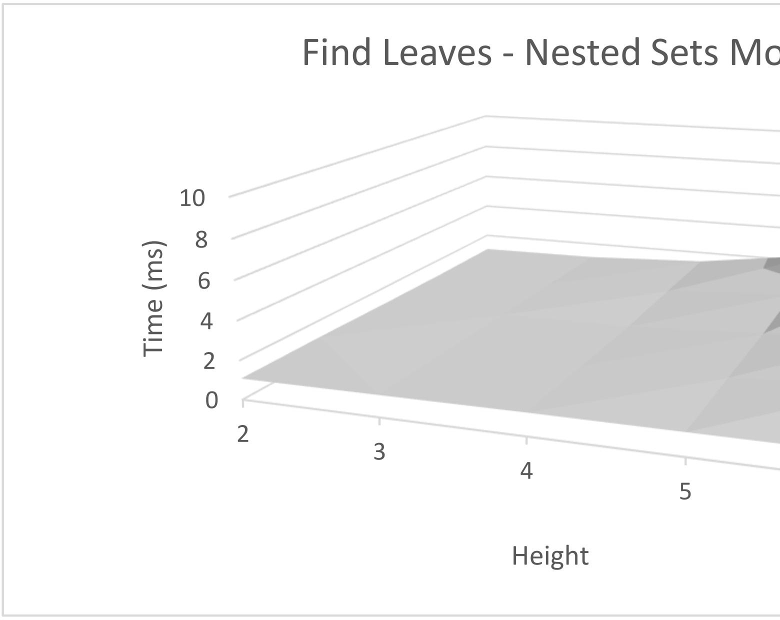


Figure 9: Running time of Finding Leaves using Nested Sets Model.

If we take a slice of Figure 8 and Figure 9 where order = 8, we can see the growth of the algorithms with respect to height, as shown in Figure 11. One thing to note is that the numbers in this figure for Nested Sets is scaled by a factor, due to the fact

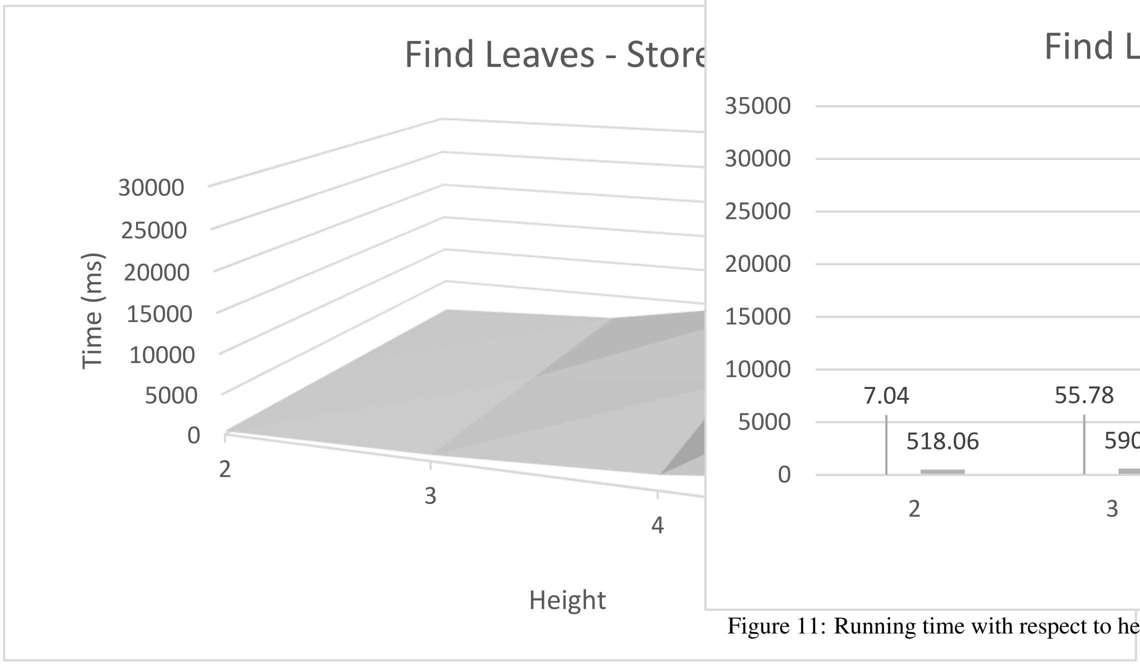


Figure 10: Running time of Finding Leaves using Stored Procedure.

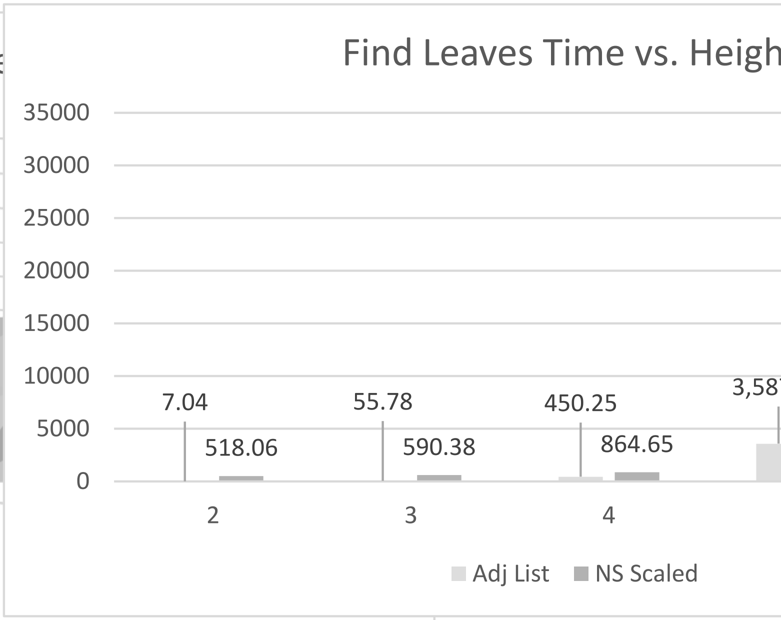


Figure 11: Running time with respect to height at order = 8.

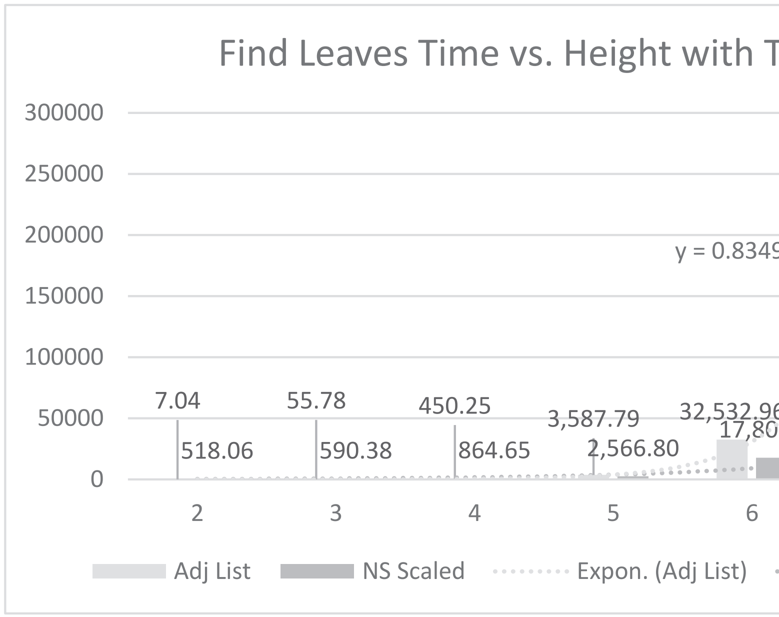


Figure 12: Running time with respect to height model at order = 8.

that the running time for the Nested Sets Model is so much lower than that of using Adjacency List Model. We already know that the algorithm using Nested Sets is generally better; therefore the purpose of this figure is not to compare the two approach, rather is to create a model of running time for each using the data provided.

Based on the data for height = 2 to 6 as shown in previous figure, a running time growth model was build using exponential regression as shown in Figure 12, where projections are shown in dotted lines. In addition, we use the data for height = 7 to validate our model. As shown in the figure, the projection from the growth model for the Adjacency List implementation fits the validation well. However, the model for the Nested Sets implementation grossly underestimate the value for the validation data. In this case, the running time jumped significantly since height = 7. This could be caused by that the experiment has not reached steady state at height = 6 and we need more data points to build a better model. Another reason could be, the database becomes extremely inefficient in processing the request for height higher than 6.

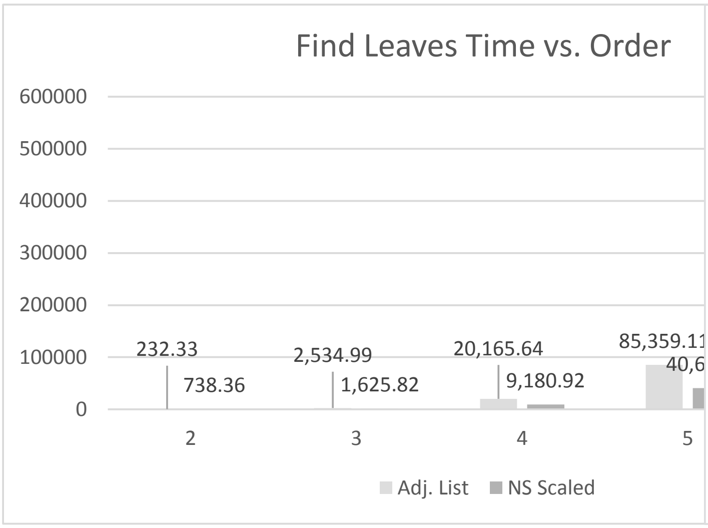
We take the same approach to analyze the running time growth rate of the operation with respect to order and the result is shown in Figure 13 and Figure 14. Once again, the number for Nested Sets Model in these figures are scaled to fit to combine the graph of both approaches in the same figure.

Intuitively, the running time growth rate should grow linearly with respect to the order (or width) of the tree; however, based on the initial observation of the data, it appears to be exponential, so we build a growth rate model also using exponential regression as shown in Figure 14 and using data for order = 7 as validation for our model. As one can see from the figure, the model overestimates the running time for the Adjacency List implementation. This is an indication to confirm our intuition that the growth rate is linear instead of exponential. However, the model fits the validation well for the Nested Sets implementation. This is a bit of a surprise since, this implementation usually have better performance.

6. CONCLUSION

In this paper, we conducted performance evaluation of three implementations of hierarchy in relational database. The first is the Adjacency List Model, which is realized by storing all nodes of the hierarchy in a single Nodes table, and each node, represented by a record in the table, contains a self-referencing foreign key that references the parent node.

The second is the Nested Sets Model, which is an attempt to improve the performance of data retrieving operations over the Adjacency List Model. This model is an extension of the first in that each node (or record) contain two additional *left* and *right* fields



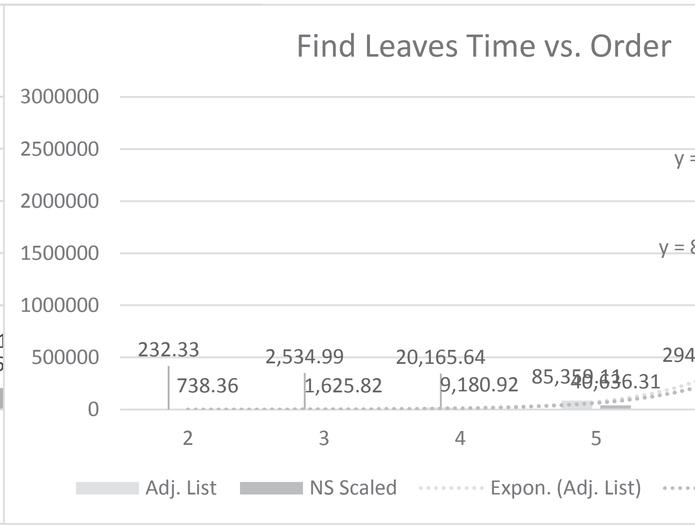


Figure 13: Running time with respect to order at height = 8.

Figure 14: Running time with respect to order model at height = 8.

that defines the subset relationship. Left and right fields are then indexed using B-Tree for fast access. With this extension, one can easily check if a node is in the hierarchy of another node.

The third implementation is the Stored Adjacency List, in which the same algorithm used in Adjacency List Model implemented in MySQL Stored Procedure. This technique is used to verify if stored procedure is a viable way for performance tuning for hierarchy operations.

Performance evaluation were conducted using the operations defined in Section 2.3. Operations are divided into 3 performance groups: O(log N), O(N), and O(1). Members of the same group have similar performance characteristic; therefore, one operation from O(log N), and O(N) is chosen for performance evaluation as representative for the group. The evaluation for group O(1) was conducted, but not reported in this paper, because the run time is negligible and not influenced by input size. The operation to find the root of a node was chosen to represent O(log N) group, and the operation to find all leaf nodes of a node was chosen to represent O(N) group. The performance of the models are measure in running time of the operations, in which the lower the running time the better.

For group O(log N), the evaluation reported in Section 5.4.1 shows the expected running time performance. For the Adjacency List Model, the growth rate of the algorithm is linear with respect to the height and constant with respect to the width (order) of the hierarchy. The Nested Sets Model is a huge improvement over the first in this evaluation. Based on the evaluation, the running time growth rate is constant for both dimensions. For the stored implementation, the running time is better than both other approaches. However, the project model is slow-growing linear with respect to height, and constant with respect to order.

In Group O(N), the Nested Sets Model still significantly outperformed Adjacency List Model; however the running time of the stored approach is unreasonable high for very small data sets. For the Adjacency List Model, the running time, as expected, have exponential running time growth with respect to the height of the tree. From the evaluation result, the growth rate with respect to width also appears to be exponential; however, our validation does not fit the exponential model, therefore we believe the growth is a steep linear model.

For the Nested Sets model, the growth rate is also exponential. Although the running times are much lower that of the other approach, the growth is faster. Frankly, I did not expect an exponential growth rate due to the optimizations that has been done. I believe this could be caused by my own implementation. As for the time complexity against the width, it is also exponential. This is also unexpected. Again this is due to that the implementation requires table joins, which degrades the performance very quickly as the data size increases.

7. 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 Relational Journal*.