Hierarchies in SQL - Nested Sets

Graeme Geldenhuys

2009-09-14

Trees and hierarchies often appear in software. They can take on various forms - from visual user interface components like the Lazarus TTreeview, to a hierarchy describing an organisational chart of a company – showing available positions that can be occupied by employees. Often such hierarchies need to be stored in a Relational Database, saving the various items of information, including the relationships between those items. Quickly you will notice that relational database table are not well organised for hierarchies. Database tables are flat lists, compared to a well structured hierarchical XML files. This article will help you overcome the difficulties in representing hierarchical data in a database table. The method I will be using is called Nested Sets.

Introduction to Graphs

Trees are a special kind of directed graph[1]. A graph is a diagram representing a system of connections or interrelations among two or more things. The relationship between these “things” are depicted by a number of distinctive lines. In a more general term, a Tree consist of *nodes* which are normally represented by boxes. These nodes are joined by *lines,* which represent the relationship between two or more nodes.

A tree is normally draw from top to bottom – like an upside down tree from a park. The lines connecting the nodes only touch two edges of a box. For this article we can call those the “in edge” and “out edge”.

A node that has a connecting lines touching the “in edge” of another node, is referred to as the *parent node*. The node on the opposite end of that line is referred to as the *child node*. A node can have zero or

1 http://en.wikipedia.org/wiki/Graph_%28mathematics%29

more child nodes. The topmost node is called the *root node*, and does not have any parent node. It will only have lines touching its “out edge”.

If a node has no children, it is commonly referred to as a *leaf node*. It is also worth mentioning that you can always follow a path from the root node to any other node in the hierarchy.

These are the common terms I will be using in this article. Now that we have the terminology covered, it's time I show you a visual representation of such a graph.

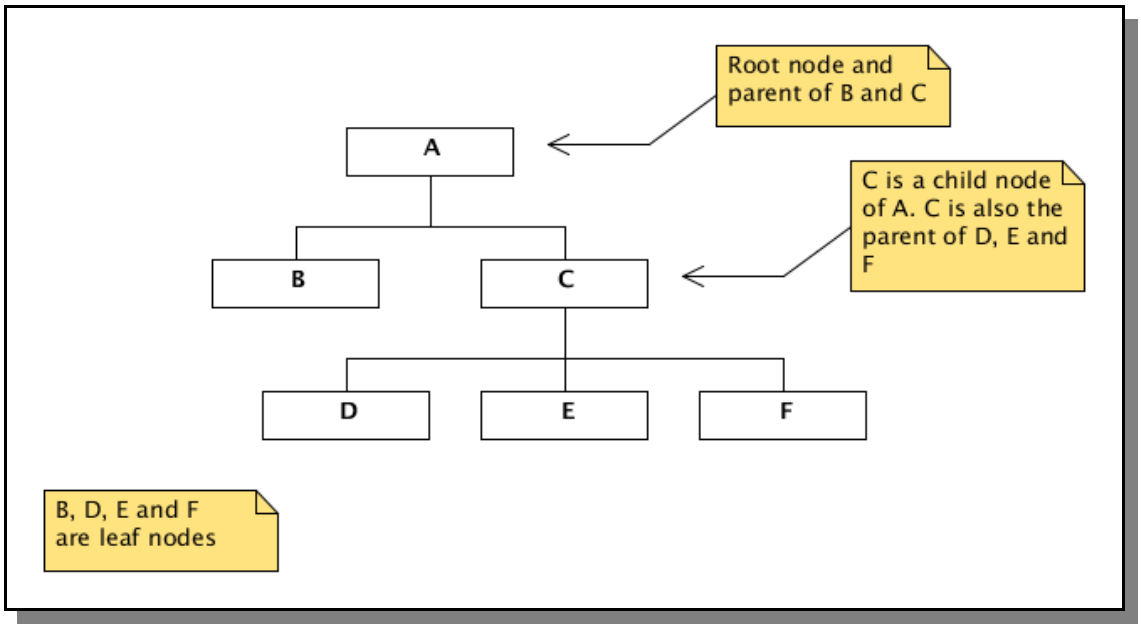

Figure 1: Basic tree hierarchy.

There are three popular methods of modelling hierarchies. These are *Nested Sets*, *Adjacency Lists* and *Path Enumeration Model*. Before I get to the Nested Sets modelling technique, I will briefly describe the two other alternatives, so that when I work through the Nested Set approach, I can do some comparisons and show where there are pros and cons compared to the two other approaches.

Also it is important to note that when hierarchies are going to be stored in a database, that it is wise to treat the hierarchy and the nodes as separate information. Thus it would be advisable if you stored those pieces of information in separate tables. The nodes would be stored in one table, which represents the data you are working with. The tree would be in another table which represents the relationships between nodes.

Adjacency List Model

This model was first introduced by Dr. Edgar F. Codd[2] after initial criticisms surfaced that the relational model (used in databases) could not model hierarchical data.

This is probably the most popular model used in databases, because it is very easy to implement. It also very closely represents how graphs where handled in programming languages like C, where programmers used pointers to build a hierarchy.

Even though it is a very popular model to use in relational databases, it is definitely not the best method to use. The reason being that the data is not well normalized. It causes a lot of data duplication. It also complicates table updates with its often usage of referential constraints.

```
CREATE TABLE AdjTree (
  child CHAR(2) NOT NULL,
  parent CHAR(2),
  PRIMARY KEY (child, parent)
);
```

Above is very basic SQL, creating a table with two fields. The `parent` field allows NULL simply because the Root Node has no parent.

AdjTree	
child	parent
'A'	NULL
'B'	'A'
'C'	'A'
'D'	'C'
'E'	'C'
'F'	'C'

Here I populated the `AdjTree` table with data representing the tree hierarchy shown in Figure 1. You can clearly see the data duplication of the child data used in the parent field.

Path Enumeration Model

In this model, a string representing the path from the root node, is stored as part of the data in a node. Itzik and Tom go into a lot more detail about this approach in their book entitled “Advanced Transact-

2 http://en.wikipedia.org/wiki/Edgar_F._Codd

SQL for SQL Server 2000”[3].

Looking at a populated dataset, one can immediately see the duplication of data, especially if you have large hierarchies that go many levels deep.

Even though this model looks convenient when you look at the data, from a visual point of few, it isn't. To work with the nodes, you have to resort to string handling functions inside your SQL statements, to correctly retrieve the hierarchical information. Not all relational database management systems are equal in their string handling abilities, so your data and SQL queries might be limited to a specific database vendor, or an external programming language.

What makes this method even worse, is that if you had to change the root node for some reason, it will cause a massive ripple effect. You will end up having to modify every single record related to the hierarchy – the worst case senario.

```
CREATE TABLE PathEnumTree (
  node CHAR(2) NOT NULL PRIMAPRY KEY,
  path VARCHAR(600) NOT NULL
);
```

Shown above, we again create a very basic table in SQL for our path enumerated model. The table contains two field. The `node` field will contain our data, and the `path` field will contain a string representing the path in the tree hierarchy from the root node.

PathEnumTree	
node	path
'A'	'a/'
'B'	'a/b/'
'C'	'a/c/'
'D'	'a/c/d/'
'E'	'a/c/e/'
'F'	'a/c/f/'

Here again I used the tree hierarchy from Figure 1, and populated the `PathEnumTree` table accordingly. From the data in the `path` field you can see the string representing the full path from the root node, for each node. You can also clearly see the duplication of data. And the deeper the tree, the more duplication occurs.

3 Itzik Ben-Gan, Tom Moreau: Advanced Transact-SQL for SQL Server 2000. Apress 2000. ISBN-13: 978-1893115828

Nested Sets

Finally we come to the heart of this article. SQL is a set-oriented language, and because Nested Sets work with sets, it is a much better approach to model a hierarchy with. Strangely enough this approach is often overlooked by most SQL developers.

Nested Sets work in a similar principal to standard XML or HTML. As with XML and HTML, you have tags nested inside other tags. The tags clearly indicate the *beginning* and *end* of a set.

```
<html>
   <head>
      <title>...</title>
   </head>
   <body>
      ...
   </body>
</html>
```

Nested sets use a similar approach. The best way to describe this is to look at a visual representation of a hierarchy, as shown in Figure 2.

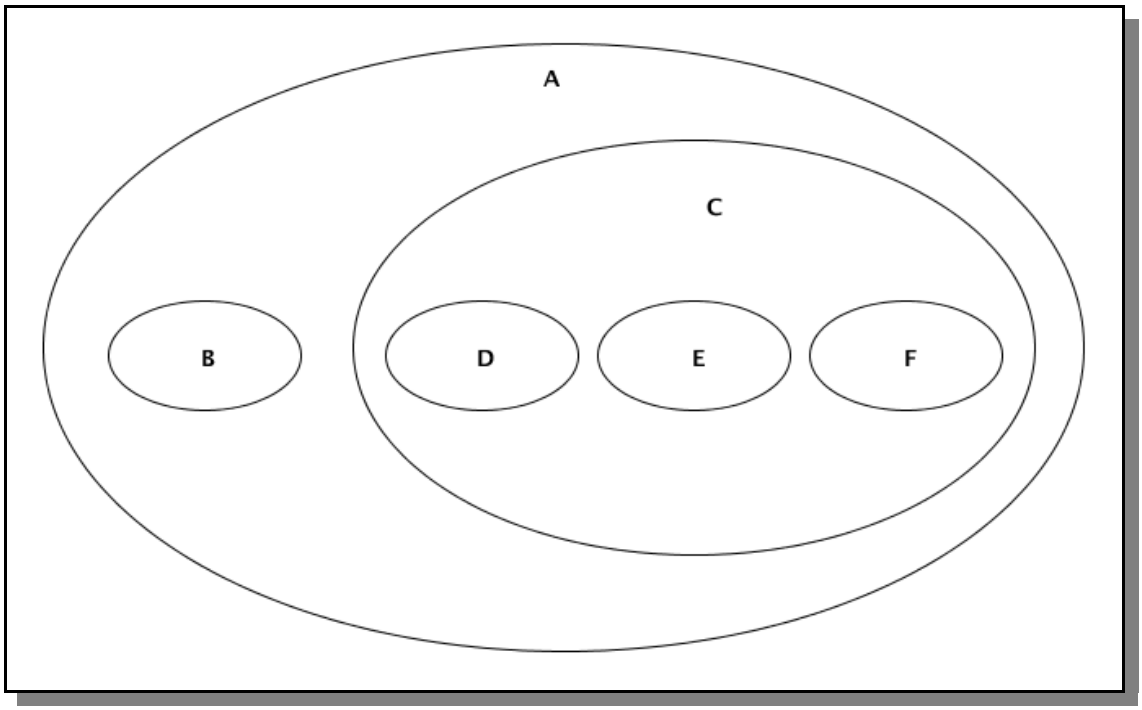

Figure 2: Oval representation of Nested Sets Model.

If we have to flatten the oval nested sets example using a numbered line, we will end up with the following diagram shown in Figure 3. As you can see, A spans the whole range and covers all the other items on the line. You will also notice how C covers the D, E and F nodes.

In both Figure 2 and Figure 3 you will also notice that the lines never overlap each other. Sets are completely enclose inside other sets.

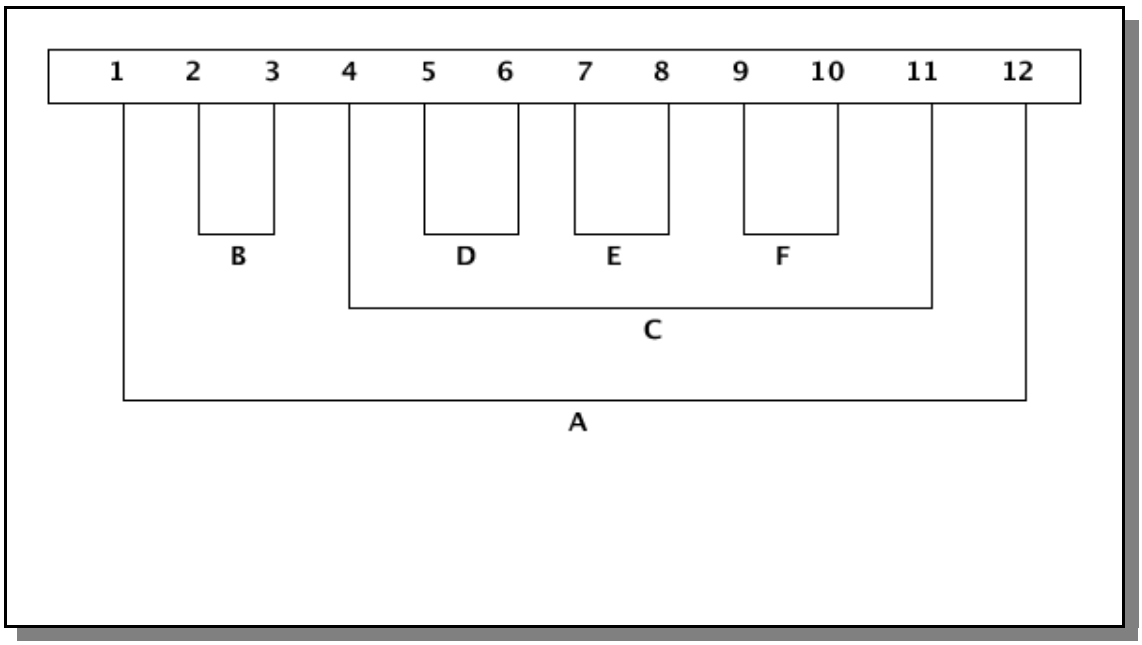

Figure 3: Nested Set flattened as a line of numbers.

Now lets implement this design as a SQL table. Below is the SQL statement to create our sample table. Note the two fields “lft” and “rgt”. These are simply abbreviations for LEFT and RIGHT. We can't actually use the words LEFT and RIGHT as field names in the table, because they are both reserved words in SQL-92. I'll explain the reason for the *lft* and *rgt* fields in short while, lets first look at the table design and a populated table.

```
CREATE TABLE NestedTest (
  node CHAR(2) NOT NULL PRIMARY KEY,
  lft INTEGER NOT NULL CHECK (lft > 0),
  rgt INTEGER NOT NULL CHECK (rgt > 1),
  CONSTRAINT correct_order CHECK (lft < rgt)
);
```

In our table we also added some simple data sanity checks to make sure we can only enter valid data into our *lft* and *rgt* fields. Now lets populate our table using the same information as shown in Figure 1.

NestedTest		
node	lft	rgt
'A'	1	12
'B'	2	3
'C'	4	11
'D'	5	6
'E'	7	8
'F'	9	10

At last, lets explain those *lft* and *rgt* fields. If you take a closer look at the populated `NestedTest` table above, and Figure 3, our flattened nested set, you will see where the numbers come from.

The lines of A in Figure 3 touch the numbers 1 and 12. The lines of B touches the numbers 2 and 3 etc. Another way to explain where the numbers come from, is look at the original tree hierarchy in Figure 1. If we had to draw an outline around our tree, and keeping as close as possible to the nodes, and starting at the root node. We draw the outline in a anti-clockwise direction. If our outline reaches the left edge of a node we increment our counter and add the value to the lft field. If we continue our outline, we will eventually reach the right side of our node. Again keep incrementing our counter and add the value to the rgt field. Do this right around our tree and you will eventually reach the root node again.

These numbers play and important part in the nested set model, and yield some predictable results, which greatly simplifies our SQL statements. Some of these predictable results are as follows:

- The Root node will always contain a 1 in the lft field.
- The amount of nodes in our hierarchy is the rgt value of our Root node divided by 2. In this case 12 / 2 = 6. And as you can see from Figure 1 we have a total of 6 nodes in our tree.
- A leaf node (a node that doesn't have children) is always a node that when you subtract the lft value from the rgt value, the result must be 1.

  Lets look at node B as an example: 3 – 2 = 1

  Now lets look at another leaf node, node E: 8 – 7 = 1
- If we wanted to know how many child nodes are under a specific parent node we can use the following formula.

  ((parent.rgt – 1) – parent.lft) / 2

  Lets use node C as an example: ((11 – 1) – 4) / 2 = 3 nodes

Now lets convert these predictable results into SQL statements. We start off with finding the Root node:

```
SELECT * FROM NestedTest WHERE lft = 1;
```

As we continue you will see how much we reference the lft and rgt fields. Now if we added indexes on those fields, it will greatly improve our SQL performance as well.

Now lets look for all leaf nodes in our table:

```
SELECT * FROM NestedTest WHERE lft = (rgt - 1);
```

I could have written the WHERE clause as `(rgt - lft) = 1`, but then I would not be taking advantage of our indexes on those fields. So the way I did it, gives us a slight performance boost.

Now lets count the amount of child nodes under the parent node C:

```
SELECT 'Node C has ', ((rgt-1) - lft) / 2, ' children'
  FROM NestedTest
  WHERE node = 'C';
```

As you can see the SQL statements are all pretty easy so far. These statements would already have been a lot more complicated in the Adjacency List Model or the Path Enumeration Model.

Now lets continue this trend and pull some more valuable information from our NestedTest table. Say our tree was representing a corporate hierarchy and we wanted to know who is the boss of who, we would write a query as follows:

```
SELECT p.node, 'is the boss of ', c.node
  FROM NestedTest as p, NESTEDTEST as c
  WHERE (c.lft BETWEEN p.lft AND p.rgt)
    AND (c.lft <> p.lft);
```

In the query above we used a self-join and table aliases, where p represents “parent” and c represents “child”. We will following this style in the queries to come.

Now lets say we where interested in a list of child nodes for a specific parent. We would put together a query as show next. In this example, we want to find out all the child nodes for the parent node C.

```
SELECT c.node, ' is a child of C'
  FROM NestedTest as p, NestedTest as c
  WHERE c.lft BETWEEN p.lft AND p.rgt
    AND c.lft <> p.lft
    AND p.node = 'C';
```

Something that is always handy to find out about a tree structure, is to know on which level each node sits. In the following SQL query, we define our Root node as being at level 1 and start the count from there. This query returns each nodes name, and on what level that node resides.

```
SELECT c.node, (COUNT(p.node)) as treelevel
  FROM NestedTest as p, NestedTest as c
  WHERE (c.lft BETWEEN p.lft AND p.rgt)
  GROUP BY c.node;
```

So far we have only retrieved information from a populated table, but what happens if the records in the table get modified. Lets see what happens with a delete action. I would also like to remind you at this time, that the Nested Set Model works in a left to right order for any nodes below the Root node – remember, that's how we got to the values for the lft and rgt fields.

Now if we had to delete node C, we can follow one of two routes. Promote the left most child node, D in this case, up one level into the gap where C used to be. If we had to put this in terms of a family tree it would go something like this: The father passed away, so the eldest son takes over the family business. The other alternative is to promote all the child nodes up one level. To put this again in the terms of a family. The parents passed away, so the grandparents adopt the kids. This is the most popular option used in Nested Sets, and what I will demonstrate.

The other thing we would have to do after moving the child nodes up one level, is to resynchronise the lft and rgt fields of all the nodes affected by this move. This is not a requirement in Nested Sets – having all lft and rgt fields in order without gaps. But if gaps exist, we loose some of the benefits of Nested Sets as I described earlier.

To help simplify our resynchronisation query, I am going to create a new helper View in our database. We will reference that View in our query. This View acts as a temporary table – nothing too complicated.

```
CREATE VIEW LftRgt (seq_number)
AS
  SELECT lft from NestedTest
  UNION
  SELECT rgt FROM NestedTest;
```

```
UPDATE NestedTest
  set lft = (SELECT COUNT(*) FROM LftRgt
             WHERE seq_number <= lft),
      rgt = (SELECT COUNT(*) FROM LftRgt
             WHERE seq_number <= rgt);
```

I bet you were surprised to see how simple this query actually is. Again one of the benefits in using the Nested Sets Model! So as soon as we deleted a node somewhere in our tree, we simply need to run the query show above, and our lft and rgt fields will be in sequential order without any gaps. All our benefits or Nested Sets are still intact.

After deleting the C node, our tree will look like the one shown in figure 4.

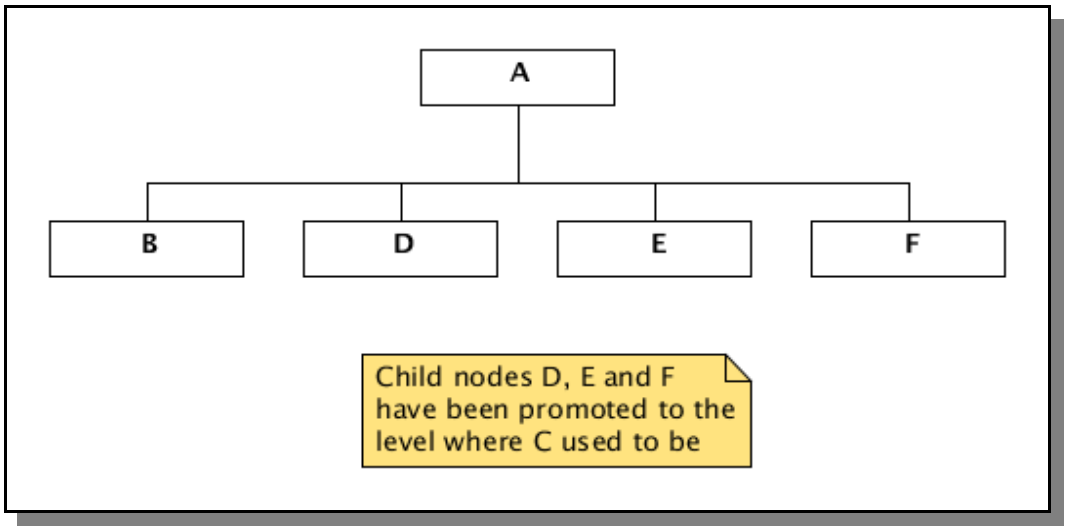

Figure 4: Tree after node C was deleted.

Conclusion

There are many other queries one can implement to extract information from a Nested Sets Model. What I described here is hopefully enough to get you interested in this model, or any of the alternative models used for working with hierarchies in a relational database. I only listed three models in this article, but there are many more specialised models out there. The key being *specialised*.

It is important to note that no one model will fit all types of data. The model you choose might also depend on how you work with your data. Do you frequently add or delete nodes, or do you simply use the tree to extract various statistics from the hierarchy. Various models have various performance gains, or ease of working with the hier-

archy.

I hope you found this article useful, and please search on the internet for more information about hierarchy usage in relation database. It's a fast topic, but a very interesting one. A good book I can recommend covering this topic in great detail, is a book entitled “Trees and Hierarchies in SQL for Smarties”[4]. A book worth reading.

4 Joe Celko: Trees and Hierarchies in SQL for Smarties. Morgan Kaufmann 2004. ISBN-13: 978-1558609204