


DB Ghost™
Make your life easier and download your FREE evaluation copy

 Microsoft
SQL
Server
articles,
news and
forums

[Home](#) | [Weblogs](#) | [Ask SQLTeam](#) | [Forums](#) | [Search](#) | [FAQ](#) | [Elsewhere](#)
[Submit Link](#) | [RSS](#) | [Write an Article](#) | [Resources](#) | [Newsletter](#) | [Books](#) | [Advertise](#) | [About](#)

Topics

[All Topics](#)
[General](#) (76)
[Database Design](#) (59)
[Queries](#) (105)
[Stored Procs](#) (26)
[Performance](#) (53)
[Administration](#) (113)
[App Design](#) (64)
[OLAP](#) (9)
[Security](#) (37)
[Installation](#) (13)
[Import/Export](#) (25)
[Other](#) (41)
[Site News](#) (27)
[Books](#) (13)
[Resources](#) (59)
[Suggested Books](#) (14)

Article Search

[Advanced Search](#)

Poll

Are your database operations being affected by Sarbanes-Oxley?

- ☐ Yes
☐ No
☐ Outside the US but similar laws
☐ Yak Herders aren't covered by the law

[Results](#) (1096) - 4

[Comments](#)

Newsletter

Your e-mail address:

Write for Us

More Trees & Hierarchies in SQL

 robvolk on 5/1/2002 in [Table Design](#)

Hierarchies are sometimes difficult to store in SQL tables...things like trees, threaded forums, org charts and the like...and it's usually even harder to retrieve the hierarchy once you do store it. Here's a method that's easy to understand and maintain, and gives you the full hierarchy (or any piece of it) very quickly and easily.

While XML handles hierarchical data quite well, relational SQL doesn't. There are several different ways to model a hierarchical structure. The most common and familiar is known as the adjacency model, and it usually works like this:

The table would look like this:

EmployeeID	Name	BossID
1001	Denis Eaton-Hogg	NULL
1002	Bobbi Flekman	1001
1003	Ian Faith	1002
1004	David St. Hubbins	1003
1005	Nigel Tufnel	1003
1006	Derek Smalls	1003

And the org chart/indented list looks like this:

```

Denis Eaton-Hogg
  Bobbi Flekman
    Ian Faith
      David St. Hubbins
        Nigel Tufnel
          Derek Smalls
  
```

It's called the "adjacency" model because the parent (boss) data is stored in the same row as the child (employee) data, in an adjacent column. It's a pretty straightforward design that's easily understood by everyone...no deep relational theory needed. You can find a person's boss easily, and you can find their coworkers by querying the BossID column.

The trouble begins when you want to list several levels of a hierarchy. To find the boss's boss, you would need to join the Employees table to itself, like this:

```

SELECT BigBoss.Name BigBoss, Boss.Name Boss, Employees.Name Employee
FROM Employees
INNER JOIN Employees AS Boss ON Employees.BossID=Boss.EmployeeID
INNER JOIN Employees BigBoss ON Boss.BossID=BigBoss.EmployeeID
  
```

And you'd get the following:

BigBoss	Boss	Employee
Denis Eaton-Hogg	Bobbi Flekman	Ian Faith
Bobbi Flekman	Ian Faith	David St. Hubbins
Bobbi Flekman	Ian Faith	Nigel Tufnel
Bobbi Flekman	Ian Faith	Derek Smalls

For each level, you'd need to join the table to itself...not an attractive option if you have 5

SQLTeam.com is looking for **authors** and **articles** on Microsoft SQL Server. If you're interested, we have some [information](#) for you to look at.

Features

[Ask SQLTeam](#)

[Submit a Link](#)

[Newsletter](#)

[Books & Merchandise](#)

[FAQ](#)

[Advertising](#)

[About](#)

Hosting

Hosted By:
orcsweb

User Info

[Login in the Forums](#)
or [register](#)

or more levels! It would be great if it could join itself as many times as needed. This is called a recursive join, and though some database products support it (Oracle has the CONNECT BY syntax) SQL Server is not one of them.

If you look in Books Online under "expanding hierarchies" you'll find a stored procedure that runs through an adjacency table to expand the hierarchy. While it works, it's a procedural method that requires a stack (using a temp table) and can take a while to run with large hierarchies. It also PRINTs out the indented list, so you'd need to modify it to use ANOTHER temp table if you wanted the results as a table/query.

If you've followed Joe Celko's columns or bought his books, he recommends the [nested set model](#) for representing trees in SQL (he's [posted it](#) on SQL Team [a few times](#)). It's very well detailed in the following articles, [Part I](#), [II](#), [III](#), [IV](#), and also in his book, [SQL For Smarties](#), and I recommend checking it out. It's very efficient and makes it extremely easy to pull out trees/subtrees from the table.

However (you knew this was coming!) one of the issues I have with nested sets is the complexity required to do relatively simple tasks, like adding, deleting, or moving nodes in the tree. Even finding an employee's immediate supervisor or subordinates requires 3 self-joins AND a subquery! Joe admits this shortcoming in his book...and it's interesting that the solution ONLY appears in his book, I've never seen him post it online.

Although there's a very seductive logic to nested sets, and it's easy to do complicated tree operations with them, I find them less intuitive than the adjacency model. It's harder for me to visualize a hierarchy or org chart with them. You may be able to use them more easily than I can, but if you also find them daunting, read on.

So how to represent a hierarchy, using adjacency, and avoiding recursion wherever possible? It's pretty easy really...you build it and store it in the table! (I've posted this method in [this thread](#) a while back, and I'm elaborating on it here)

Here's the table definition for the Tree:

```
CREATE TABLE Tree (
  Node int NOT NULL IDENTITY(100, 1),
  ParentNode int,
  EmployeeID int NOT NULL,
  Depth tinyint,
  Lineage varchar(255) )
```

I'm keeping the Tree table separate for a few good reasons I'll discuss later, but you could simply add the Depth and Lineage columns to the Employees table above, and substitute BossID for ParentNode. (I also didn't really WANT to use an identity column, but most people will anyway) The terms "node" and "lineage" might seem unfamiliar, but I wanted to generalize them a little more than "child", "parent" and "hierarchy".

Based on the Employees table, here's how the Tree will be filled:

Node	ParentNode	EmployeeID	Depth	Lineage
100	NULL	1001	NULL	NULL
101	NULL	1002	NULL	NULL
102	NULL	1003	NULL	NULL
103	NULL	1004	NULL	NULL
104	NULL	1005	NULL	NULL
105	NULL	1006	NULL	NULL

The first thing to do is to populate the parent nodes, which is unnecessary if you use a single

table, but it's easy to do in any case:

```
UPDATE T SET T.ParentNode=P.Node
FROM Tree T
INNER JOIN Employees E ON T.EmployeeID=E.EmployeeID
INNER JOIN Employees B ON E.BossID=B.EmployeeID
INNER JOIN Tree P ON B.EmployeeID=P.EmployeeID
```

And you'll get this:

Node	ParentNode	EmployeeID	Depth	Lineage
100	NULL	1001	NULL	NULL
101	100	1002	NULL	NULL
102	101	1003	NULL	NULL
103	102	1004	NULL	NULL
104	102	1005	NULL	NULL
105	102	1006	NULL	NULL

This will only need to be done once, and afterwards you won't need to maintain the BossID column in the Employees table. The next part is to find the root node of the tree, also known as the top-level, or big boss man, etc. in an org chart. That's the node that has no parent (Null), so we will start there and set the Lineage column as the root:

```
UPDATE Tree SET Lineage='/', Depth=0 WHERE ParentNode Is Null
```

Once that's done, we can then update the rows who are immediate children of the root node:

```
UPDATE T SET T.depth = P.Depth + 1,
T.Lineage = P.Lineage + Ltrim(Str(T.ParentNode,6,0)) + '/'
FROM Tree AS T
INNER JOIN Tree AS P ON (T.ParentNode=P.Node)
WHERE P.Depth>=0
AND P.Lineage Is Not Null
AND T.Depth Is Null
```

In fact, we can just put a loop on this to run through all of the children/grandchildren etc. of the tree:

```
WHILE EXISTS (SELECT * FROM Tree WHERE Depth Is Null)
UPDATE T SET T.depth = P.Depth + 1,
T.Lineage = P.Lineage + Ltrim(Str(T.ParentNode,6,0)) + '/'
FROM Tree AS T
INNER JOIN Tree AS P ON (T.ParentNode=P.Node)
WHERE P.Depth>=0
AND P.Lineage Is Not Null
AND T.Depth Is Null
```

Don't worry about the loop, it runs once for each level in the hierarchy...10 loops for 10 levels or generations. For a corporation, 10 layers of management is pretty deep; for a family tree, you could trace an American family back to the Revolutionary War! And under normal circumstances, you'd also only have to run this procedure once. The final result is:

Node	ParentNode	EmployeeID	Depth	Lineage
100	NULL	1001	0	/
101	100	1002	1	/100/

102	101	1003	2	/100/101/
103	102	1004	3	/100/101/102/
104	102	1005	3	/100/101/102/
105	102	1006	3	/100/101/102/

You'll notice that for each node, the entire lineage back to the root is stored. This means that finding someone's boss, or their boss' boss, doesn't require any self-joins or recursion to create an indented list. In fact, it can be accomplished with a single SELECT!

```
SELECT Space(T.Depth*2) + E.Name AS Name
FROM Employees E
INNER JOIN Tree T ON E.EmployeeID=T.EmployeeID
ORDER BY T.Lineage + Ltrim(Str(T.Node,6,0))
```

If you kept everything in one table you would not even need the JOIN! The Depth column comes in handy for performing the indent by using the Space() function. Using ORDER BY Lineage...etc. will sort the org chart properly, with each subordinate nesting underneath their parent. Sort order is maintained by Node values, and can be changed simply by updating the node value. Inserting or deleting a new node does not affect the rest of the tree, unlike the nested set model. The lineage column can be maintained automatically using triggers, so moving or promoting a node is a no-brainer.

In Part 2 of this article I'll cover all the necessary tree operations that are needed (add, delete, move, promote) and various queries (all superiors, all descendants, paths, org levels, etc.) and some other neat tricks you can do (multiple independent trees in the same table, genealogy trees)

Related Articles:

[The Physical Implementation](#) on 9/5/2000 

[Process Modeling](#) on 8/31/2000 

[Normalizing an Existing Database](#) on 3/8/2001 

[The 7 Normal Forms - Definitions](#) on 12/11/2001 

[Denormalize for Performance](#) on 1/10/2001 rated **5.0** (1)

[Comments](#) (89) | **65240** Page Views | [Print It](#) | [Email It](#) | [\(login to rate articles\)](#)



[Home](#) | [Weblogs](#) | [Ask SQLTeam](#) | [Forums](#) | [Search](#) | [FAQ](#) | [Elsewhere](#)
 Copyright SQLTeam.com 2004. All Right Reserved. [Privacy](#)