

Versatile High Performance Hierarchies in SQL Server

Page 2 of 3 - by [Dennis W. Forbes](#)

New! <http://www.professionalsqlserver.com>. The premiere source of SQL Server tips, tricks and information

[The #1 Internet Resource For SQL Server Professionals](#)

[Go To Page 1 - Hierarchy Designs](#)

Note: SQL Server 2005 is edging close to completion. Read about how [SQL Server 2005 will change your solutions](#).

Bob L:1 R:18		
Jane L:2 R:13		Joan L:14 R:17
Ken L:3 R:8	Tony L:9 R:12	Paul L:15 R:16
Karen L:4 R:7	Tina L:10 R:11	
Howard L:5 R:6		

Figure 6 : Left and Right Bounded Hierarchies

<i>EmployeeID</i>	<i>Name</i>	<i>BossID</i>	<i>SetString</i>	<i>Depth</i>
1	Bob	Null	A	1
2	Jane	1	AA	2
3	Joan	1	AB	2
4	Ken	2	AAA	3
5	Tony	2	AAB	3
6	Karen	4	AAAA	4
7	Paul	3	ABA	3
8	Howard	6	AAAAA	5
9	Tina	5	AABA	4

Figure 7 : *EmployeeSet* Table with a String Coded Nested Set

```

CREATE TABLE [dbo].[EmployeesSet] (
    [EmployeeID] [int] IDENTITY (1, 1) NOT NULL ,
    [FullName] [varchar] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [Salary] [money] NULL ,
    [BossID] [int] NULL ,
    [StringSet] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [Depth] [int] NULL
) ON [PRIMARY]
GO

CREATE CLUSTERED INDEX [IX_StringSet] ON [dbo].[EmployeesSet]([StringSet]) ON [PRIMARY]
GO

ALTER TABLE [dbo].[EmployeesSet] WITH NOCHECK ADD
    CONSTRAINT [PK_EmployeesSet] PRIMARY KEY NONCLUSTERED
    (
        [EmployeeID]
    ) ON [PRIMARY]
GO

CREATE INDEX [IX_Depth] ON [dbo].[EmployeesSet]([Depth]) ON [PRIMARY]
GO

ALTER TABLE [dbo].[EmployeesSet] ADD
    CONSTRAINT [FK_EmployeesSet_EmployeesSet] FOREIGN KEY
    (
        [BossID]
    ) REFERENCES [dbo].[EmployeesSet] (
        [EmployeeID]
    )
GO

```

Figure 8 : EmployeeSet DDL

CodeKey CodeChar

1	A
2	B
3	C
4	D
...	...
26	0
27	1

28	2
..	..

Figure 9 : Code Table

```
SET ANSI_NULLS off
GO
```

```
CREATE PROCEDURE dbo.UpdateStringSet(@ParentEmployeeID int=null) AS
```

```
DECLARE @ParentDepth int, @ParentStringSet varchar(255)
```

```
SET @ParentDepth = null
```

```
SELECT @ParentDepth=Depth, @ParentStringSet=StringSet FROM EmployeesSet WHERE
EmployeeID=@ParentEmployeeID
```

```
IF (@ParentDepth IS NULL)
```

```
BEGIN
```

```
    SET @ParentDepth=0
```

```
    SET @ParentStringSet = "
```

```
END
```

```
DECLARE @Offset int
```

```
SET @Offset = 0
```

```
DECLARE GetChildrenCursor CURSOR READ_ONLY STATIC LOCAL FOR
```

```
SELECT EmployeeID FROM EmployeesSet WHERE BossID=@ParentEmployeeID ORDER BY StringSet
```

```
OPEN GetChildrenCursor
```

```
DECLARE @EmployeeID int
```

```
DECLARE @NewStringSet varchar(255), @OldStringSet varchar(255)
```

```
FETCH NEXT FROM GetChildrenCursor
```

```
INTO @EmployeeID
```

```
WHILE (@@FETCH_STATUS = 0)
```

```
BEGIN
```

```
    SELECT @NewStringSet=@ParentStringSet+CodeChar From Code Where CodeKey=@Offset
```

```
    UPDATE EmployeesSet SET Depth=@ParentDepth+1,@OldStringSet=StringSet,
StringSet=@NewStringSet WHERE EmployeeID=@EmployeeID
```

```
    SET @Offset = @Offset + 1
```

```
-- Recusively do the same for all children nodes if the string set has changed on this node
```

```

IF (@OldStringSet <> @NewStringSet) BEGIN
    EXEC dbo.UpdateStringSet @EmployeeID
END

FETCH NEXT FROM GetChildrenCursor
INTO @EmployeeID
END

CLOSE GetChildrenCursor
DEALLOCATE GetChildrenCursor
GO

```

Figure 10 : *UpdateStringSet* Stored Procedure

High Performance Hierarchies with Nested Sets

Another powerful technique for coding and querying hierarchical data, a hybrid of an adjacency list and the multi-level relational techniques previously discussed, is the “nested set”: By storing hierarchical hints with each record in the hierarchy, we can pull out branches and nodes using straightforward SQL WHERE conditions. The “nested set” approach offers tremendous advantages over both: It offers the speed and simplicity of the multi-level relational, with the schema independence of the adjacency list-- The best of both worlds.

The common method of implementing such hierarchical hints is by using left-right record boundary hint, with parent nodes having bounds encapsulating the bounds of all child nodes, and so on. This is the technique advocated by Joe Celko in his excellent book [SQL For Smarties](#), and is oft used as the reference implementation of powerful hierarchical designs. [Figure 6](#) demonstrates such a design, with each record having an L (left) and R (right) value -- By selecting the left and right value for the desired parent record, one can select all child records by simply querying for records containing a left value greater than the selected left value and a right value less than the selected right value. While it offers a very powerful approach, it does have some downsides, primarily the complexity involved when modifying relationships in the hierarchy: It is a complex task to maintain the nested set information when nodes are moved, deleted, and inserted, so instead we’re going to examine an equally powerfully approach utilizing strings.

String-Based "Nested Sets"

SQL Server offers some powerful string comparison features such as *LIKE*. Coupling these with appropriate indexing allows for high performance “nested sets” (where we are bounding sets of data by textual markers) utilizing string keys to encapsulate hierarchical relationships in records. [Figure 7](#) represents the adjacency list from [Figure 4](#) with the addition of the columns SetString, a varchar column with case sensitive, accent sensitive collation (for example Latin1_General_CS_AS), and Depth, an integer column. [Figure 8](#) includes the SQL DDL to create a table and set-up the appropriate indexes for

speedy hierarchical searches. SetString, you will notice, is a string with each record having the value of its parent record, but with an incrementing additional character. Depth indicates the distance from the root (note that root in actuality is always an invisible “null” record, though in usage most users will have a single literal record as their functional root), and will always equal the length of the string held by the SetString field, though we maintain it as a separate field to allow for easy indexing. Such an approach allows for incredibly simplistic querying: To return a set of all of Jane’s underlings, both direct and indirect, one can run a simple query like the one following.

```
SELECT * FROM Employees WHERE SetString LIKE (SELECT SetString+'%' FROM Employees WHERE EmployeeID=2)
```

With an index on the SetString, such a query is highly efficient. This could easily be modified to only include records within a certain depth range if desired. The follow query gets all of Jane’s direct reports, and reports of her direct reports (the two levels below her).

```
DECLARE @SetString nvarchar(255), @DepthTop int, @DepthBottom int
```

```
SELECT @SetString = SetString+'%', @DepthTop=Depth+1, @DepthBottom=Depth+2 FROM Employees WHERE EmployeeID=2
```

```
SELECT * FROM Employees WHERE SetString LIKE @SetString and Depth >= @DepthTop and Depth < = @DepthBottom
```

Of course this could also be accommodated by using the functionality of the LIKE comparator.

```
DECLARE @SetString
```

```
SELECT @SetString = SetString FROM Employees WHERE EmployeeID=2
```

```
SELECT * FROM Employees WHERE SetString LIKE @SetString + '_' OR SetString LIKE @SetString+'__'
```

In such an example only SetStrings with one or two additional characters beyond a matching initial set will be returned, achieving the same results.

Automatic Nested Sets

Generating the SetStrings field values to properly encapsulate the hierarchical relationships is a non-trivial operation. The first step is deciding on the code sequencing that we wish to use, as there may be a desire to use these coding patterns outside of the database (for example if used for employee organizational codes)--i.e. If there is a desire for them to be readable and memorable where necessary (a requirement which excludes us from using carriage returns or other system characters). The number of items that can share a common direct parent is dictated by the breadth of the coding pattern: In this case we’ve used a varchar--varchars use one byte, allowing 256 distinct values per character--because it can be matched and stored efficiently, though if one planned on having hundreds or thousands of elements with the same parent, an nvarchar could be used. So long as you created the StringSet field as a case-sensitive, accent-sensitive column, you can have both an upper and lower-case letter as separate

character codes, as well as letters with varying accents. One could simply have an incrementing number that is converted to a **char** or **nchar** (using the functions *CHAR()* or *NCHAR()*, respectively), however for our sample we've set up a code table (see [Figure 9](#)), unsurprisingly named *code* . Because we've opted against iterating through all available character values, to control the code order and printability, we have limited the maximum number of sibling members (or nodes which share a single direct parent) to 36 in the instance that we use only upper case characters and numbers, though you could significantly increase that by adding lower case characters, punctuation, etc, changing the field to an *nvarchar* and utilizing unicode if the situation warranted it (i.e. very wide hierarchies).

[Figure 10](#) demonstrates a stored procedure which, through recursive iteration, walks from a particular element (specified by the parameter *@ParentEmployeeID*) down the employee tree filling in the *StringSet* and *Depth* values. By calling this stored procedure with a parameter of null we can update the *StringSet* for every record from the root element(s) (if there is more than one element at the root, they are presumed to have a virtual unspecified parents, and to be siblings), and immediately we can do powerful and efficient hierarchical queries. Indeed, such queries could not only return branches of the tree for direct use, but can be used as subqueries on other tables with the *IN* condition to generate sales aggregates, etc. To call this stored procedure on updates or inserts on a nested set enabled table would be a trivial operation.

[Go To Page 3 - Pulling It All Together](#)

© 2003 - [yafla](#)