

# Versatile High Performance Hierarchies in SQL Server

Page 3 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 2 - High Performance Nested Sets](#)

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

```
SET ANSI_NULLS OFF
GO

CREATE PROCEDURE dbo.UpdateStringSetStack(@ParentEmployeeID int=null, @MaxDepth int=64) AS

DECLARE @StartDepth int, @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

SET @StartDepth = @ParentDepth

DECLARE @CurrentDepth int
SET @CurrentDepth = @ParentDepth + 1

DECLARE @StackTable table(EmployeeID int, Depth int)
DECLARE @ValueTable table(Depth int, ParentStringSet varchar(255), Offset int)

DECLARE @SingleEmployeeID int, @SingleDepth int
DECLARE @NewStringSet varchar(255), @OldStringSet varchar(255)

INSERT INTO @StackTable(EmployeeID, Depth)
SELECT EmployeeID,@CurrentDepth FROM EmployeesSet WHERE BossID=@ParentEmployeeID
```

```

INSERT INTO @ValueTable(Depth, ParentStringSet, Offset)
VALUES(@CurrentDepth,@ParentStringSet,0)

DECLARE @Offset int
WHILE (@CurrentDepth > @StartDepth)
BEGIN
    SELECT @Offset = Offset, @ParentStringSet=ParentStringSet FROM @ValueTable WHERE Depth =
    @CurrentDepth

    SET @SingleEmployeeID=NULL
    SELECT TOP 1 @SingleEmployeeID = EmployeeID FROM @StackTable WHERE Depth=@CurrentDepth

    IF (@SingleEmployeeID IS NOT NULL)
    BEGIN
        UPDATE @ValueTable SET Offset = (@Offset + 1) WHERE Depth=@CurrentDepth
        SELECT @NewStringSet=@ParentStringSet+CodeChar From Code Where CodeKey=@Offset

        UPDATE EmployeesSet SET Depth=@CurrentDepth,@OldStringSet=StringSet,
        StringSet=@NewStringSet
        WHERE EmployeeID=@SingleEmployeeID

        -- Remove this item from the stack, and then add its children records
        DELETE FROM @StackTable WHERE EmployeeID=@SingleEmployeeID

        IF (@NewStringSet <> @OldStringSet)
        BEGIN
            SET @CurrentDepth = @CurrentDepth + 1

            IF ((@CurrentDepth - @StartDepth)>@MaxDepth)
            BEGIN
                RAISERROR ('The maximum recursive levels of %d was exceeded building nested set',16, 1,
                @MaxDepth)
            END

            INSERT INTO @ValueTable(Depth,ParentStringSet,Offset)
            VALUES(@CurrentDepth,@NewStringSet,0)

            -- Compute the new code for this item based upon the parent.
            INSERT INTO @StackTable(EmployeeID, Depth)
            SELECT EmployeeID,@CurrentDepth FROM EmployeesSet WHERE BossID=@SingleEmployeeID
        END
    END ELSE BEGIN
        -- We have exhausted this branch
        DELETE FROM @ValueTable WHERE Depth = @CurrentDepth
        SET @CurrentDepth = @CurrentDepth - 1
    END
END
END
GO

```

---

Figure 11 : *UpdateStringSetStack* Stored Procedure

```

SET ANSI_NULLS OFF
GO

CREATE PROCEDURE dbo.SingleStringSet @EmployeeID int AS

DECLARE @BossID int

SELECT @BossID = BossID FROM EmployeesSet WHERE EmployeeID=@EmployeeID

DECLARE @BossDepth int, @BossStringSet varchar(255), @OldStringSet varchar(255), @NewStringSet
varchar(255), @OldDepth int

SET @BossDepth = 0
SET @BossStringSet = ''

SELECT @BossDepth=Depth, @BossStringSet=StringSet FROM EmployeesSet WHERE
EmployeeID=@BossID

SELECT @OldStringSet=StringSet, @OldDepth=Depth FROM EmployeesSet WHERE
EmployeeID=@EmployeeID

IF (ISNULL(LEFT(@OldStringSet,@BossDepth),'')<>@BossStringSet) OR (@BossDepth = 0) OR (LEN
(@OldStringSet)<>(@BossDepth+1)) BEGIN
    DECLARE @NewCodeChar char(1)
    SELECT TOP 1 @NewCodeChar=CodeChar FROM Code WHERE CodeChar NOT IN
    (SELECT ISNULL(SUBSTRING(StringSet,@BossDepth+1,1),'') FROM EmployeesSet
    WHERE BossID=@BossID AND EmployeeID<>@EmployeeID)

    SET @NewStringSet = @BossStringSet+ISNULL(@NewCodeChar,'Z')

    BEGIN TRAN

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

    IF (@OldStringSet <> @NewStringSet)
    BEGIN
        EXEC UpdateStringSetStack @EmployeeID
    END

    COMMIT TRAN
END
GO

```

---

Figure 12 : *SingleStringSet* Stored Procedure

```

CREATE TRIGGER trg_UpdateHierarchy ON dbo.EmployeesSet
FOR INSERT, UPDATE
AS

```

```

DECLARE IterateNewRecords CURSOR LOCAL STATIC FOR
SELECT EmployeeID FROM inserted

DECLARE @EmployeeID int

OPEN IterateNewRecords

FETCH NEXT FROM IterateNewRecords
INTO @EmployeeID

WHILE (@@FETCH_STATUS = 0) BEGIN
    EXEC SingleStringSet @EmployeeID

    FETCH NEXT FROM IterateNewRecords
    INTO @EmployeeID
END

CLOSE IterateNewRecords
DEALLOCATE IterateNewRecords

```

---

Figure 13 : *EmployeeSet* Trigger

## Pulling It All Together - Automatically Maintained Nested Sets

While the stored procedure in [Figure 10](#) powerfully enables the on-demand creation of nested sets, it is limited in that it relies upon recursion. Recursion is constrained by SQL Server limitations to a maximum of 32 recursive nested calls. While a 32-level hierarchy is large enough for most hierarchies, one should strive to avoid such arbitrary limitations[\*] where possible, ensuring that crippling constraints aren't encountered on a fateful day in the future (usually on the same day that everything else that could go wrong does). As such a variant of [Figure 10](#) which uses a table "stack" architecture instead of recursive calling is desired. [Figure 11](#) shows just such a stored procedure. With this stored procedure the hierarchy depth is limited only by the space available for the table variables (which generally will facilitate massive hierarchies).

The next step in the journey to high performance, reliable hierarchies is to make it automatic: When records are added or deleted, or relationships changed, the hierarchical information is automatically updated. To accommodate this requirement, we will first create a intermediary stored procedure: A bit of functionality that checks to see if a given record *needs* to have its nested set information (the *StringSet* and *Depth* fields) updated, and only then will it perform the potentially costly task of cascade updating the dependant records. [Figure 12](#) demonstrates just such a procedure, verifying first that a given record no longer correlates hierarchically with its parent, in such an instance calling [UpdateStringSetStack](#) to update the child records.

Now that the plumbing is all in place, all that remains is the addition of a trigger on our *EmployeeSet* table to automatically call [SingleStringSet](#) when necessary. [Figure 13](#) demonstrates such a trigger,

iterating through the *inserted* virtual table and calling our intermediary stored procedure, [SingleStringSet](#), to determine if more action is necessary. Observant readers will note that the trigger only operates on the INSERT and UPDATE events, ignoring the DELETE events - This is by design, as the appropriate self-referential foreign key relationship will ensure that no record can be deleted until the dependant records have been assigned a different parent record, eliminating the need to handle DELETE events as such records could only be leaf nodes.

## Conclusion

While the examples and code in this paper have been specific, in this case to an employee hierarchy, such a design can be easily adapted to any other hierarchy type. By creating automatically maintained nested sets, you acquire the simplicity of versatile relational adjacency lists, with the speed and simplicity of nested sets: A win-win for any database.

\* - Of course, we have imposed just such an arbitrary limitation by utilizing a coding table, limiting us to 36 possible siblings with a single parent if only upper-case characters and numbers are used; However that limit can be easily expanded (by adding elements to our *code* table we can expand out to 256 siblings using a varchar string, or 65536 with a nvarchar string), and is unlikely to become a practical limit in any case. i.e. a 4 level hierarchy (presuming a virtual root) with 36 items per parent offers up some 1.6 million nodes.