

[All Topics](#), [MFC / C++ >> Database >> SQL](#)

Improve hierarchy performance using nested sets

 By [James Simpson](#)

This article describes how to use nested sets to improve performance of hierarchies within SQL Server and other relational databases

 C++, SQL
 Windows
 SQL (SQL 2000), Win32, VS
 DB, Dev

Posted 18 May 2003

 [Articles by this author](#)
19,396 views

 Search:
[Help!](#) [Articles](#) [Message Boards](#) [StoreFront](#) [Lounge](#)

Toolbox

-  [Broken Article?](#)
-  [VS.NET 2003 for \\$899](#)
-  [Ultimate Toolbox \\$499](#)
-  [Print version](#)
-  [Send to a friend](#)

Sign in / Sign up

 Email

 Password
☒ Remember me

[Lost your Password?](#)

 14 members have rated this article. Result: 

 Popularity: 5.02. Rating: **4.38** out of 5.

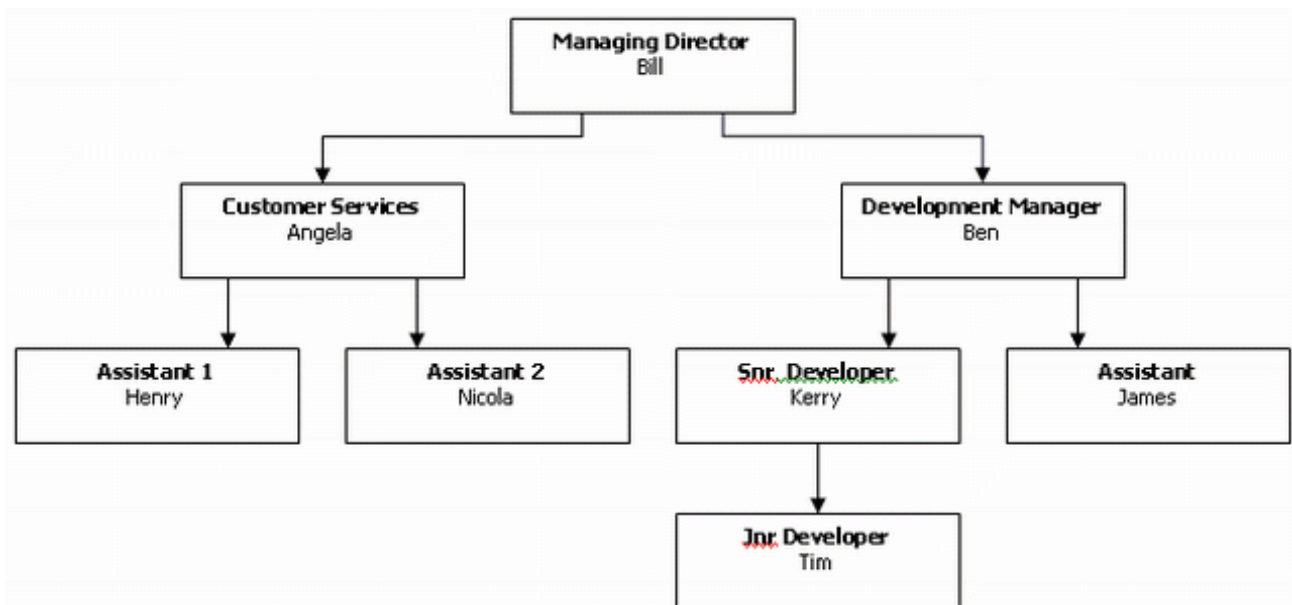
- [Download source files - 2 Kb](#)

Introduction

This article and accompanying code demonstrates the theory and practice behind "Nested Sets". The theory is applicable with any type hierarchy in any database, however the example code was written using SQL Server 2000 and every attempt has been made to make it compatible with most major SQL providers.

The Theory

Nested sets rely on a demoralization of parts of the hierarchy, They can be used to efficiently use a hierarchy without relying on the Parent / Child link. How a person usually views or represents a hierarchy is by using parent child relationship, this can be seen below:



The above hierarchy shows a make-believe company organisational structure, this data can be compiled into a database table and represented very easily, the SQL Script below shows how to create the table and populate the data:

```
-- Create the SQL Table
CREATE TABLE Employee
(
    EmployeeID INT IDENTITY(1, 1) NOT NULL,
    ParentID INT NULL,
    JobTitle VARCHAR(50) NOT NULL,
    FirstName VARCHAR(50) NOT NULL,
    PRIMARY KEY(EmployeeID),
    FOREIGN KEY ParentID REFERENCES Employee (EmployeeID)
)

GO
```

And populate the data:

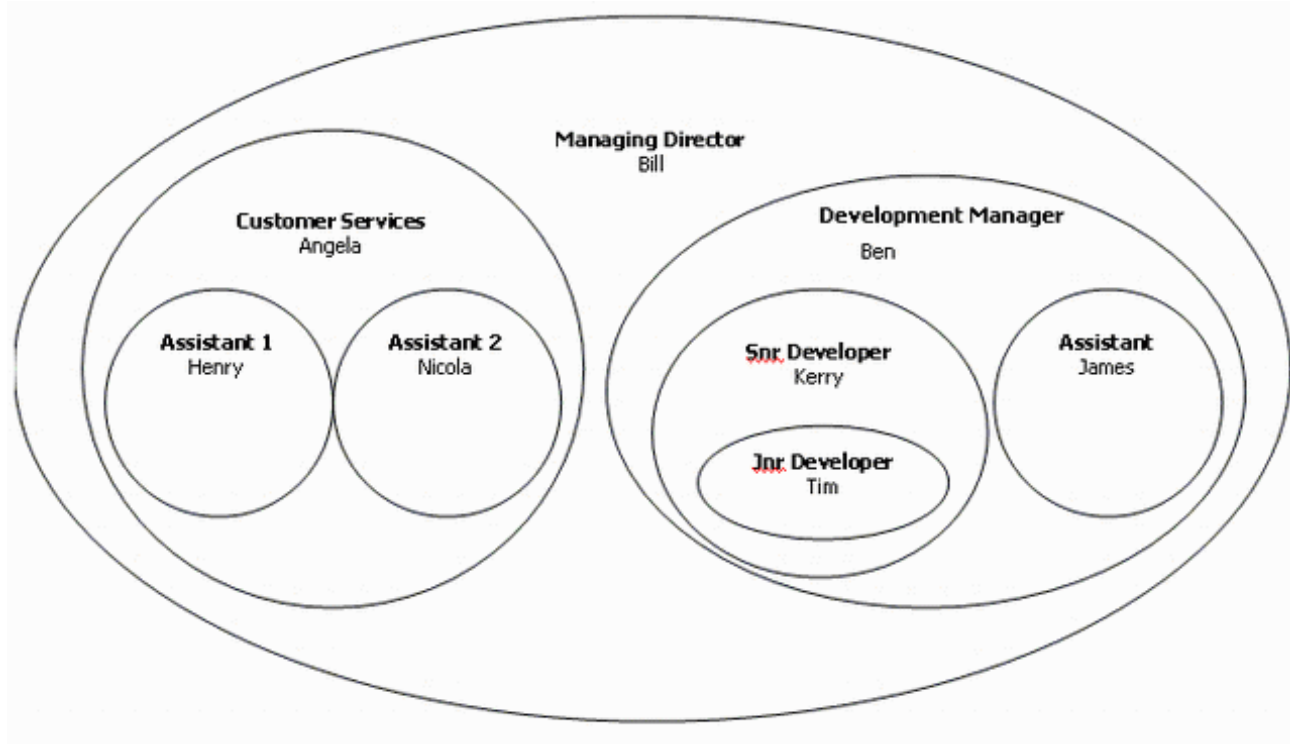
```
-- Set identity insert ON so we can insert the employee ID
SET IDENTITY_INSERT Employee ON
SET NOCOUNT ON

INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (1, NULL, 'Managing Director', 'Bill')
-- Employees under Bill
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (2, 1, 'Customer Services', 'Angela')
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (3, 1, 'Development Manager', 'Ben')
-- Employees under angela
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (4, 2, 'Assistant 1', 'Henry')
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (5, 2, 'Assistant 2', 'Nicola')
-- Employees under ben
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (6, 3, 'Snr Developer', 'Kerry')
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (7, 3, 'Assistant', 'James')
-- Employees under kerry
INSERT INTO Employee (EmployeeID, ParentID, JobTitle, FirstName)
VALUES (8, 6, 'Jrn Developer', 'Tim')
SET NOCOUNT OFF

SET IDENTITY_INSERT Employee OFF
```

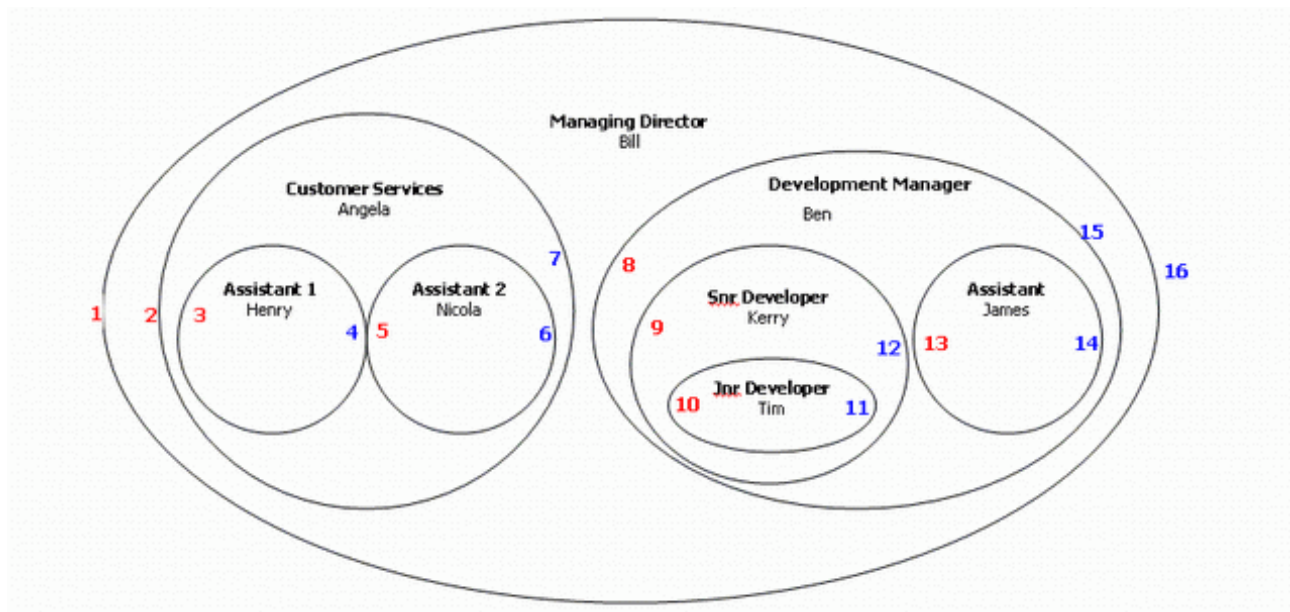
The above SQL code will create the table needed to represent the above hierarchy. Most hierarchy are developed using the above **ParentID** method, and usually it is enough to get by with especially if the hierarchy has a fixed amount of levels. However if a large part of an application is based around the hierarchy and the hierarchy is dynamic, then the above method becomes less efficient and very costly to use. Various common hierarchy operations become difficult. For instance finding the leaf node of the hierarchy using the **ParentID** becomes very difficult and can not easily be retrieved with a simple **SELECT** statement. To find any parent of a child at any level of the hierarchy you must use the **ParentID** of each node, this is also difficult and inefficient to retrieve with a simple select statement.

This is where **nested sets** become infinitely useful. If we were to represent the above hierarchy slightly differently we can gain a lot of advantages. Below is the same hierarchy represented as nested sets:



Although at first this is slightly more confusing to read visually, to a computer system this is a cinch. The above diagram is indicating that each node is aware of all its descendents, and vice versa. For example, we can see easily enough from the diagram that **Ben** and **Angela** are children of **Bill** because they are contained within **Bill**. We can also see that **Tim** And **James** are children of **Ben** because they are contained within **Ben**. This ultimately gives us a link between each member of the hierarchy.

For us to represent this in data terms we need to store two more values per node, these are the extent values of the set. These are shown below on the same diagram in blue and red.



Lets call these **Left (Red Values)** and **Right (Blue Values)**. These values should be stored on the **Employee** table along with the **ParentID** and this is all we need to fully represent a nested sets hierarchy.

```
-- Alter the employee table to include the left and right extent values
```

```
ALTER TABLE Employee ADD [LeftExtent] INT NULL
ALTER TABLE Employee ADD [RightExtent] INT NULL

GO
```

By using the Left and Right values of the database you can pick out some simple rules and simple queries:

```
-- If Left is between parent left and parent right then the node is a child
SELECT * FROM Employee
    CROSS JOIN (
        SELECT [LeftExtent], [RightExtent]
        FROM Employee WHERE Firstname = 'Ben'
    ) AS Parent
WHERE Employee.[LeftExtent]
BETWEEN Parent.[LeftExtent] and Parent.[RightExtent]

-- All Leaf nodes (an item with any children) can be
-- identified by Left = Right - 1
SELECT * FROM Employee WHERE [LeftExtent] = [RightExtent] - 1
```

More advanced queries can be used, but before these are explained the **Left** and **Right** values must be populated. Below is the SQL code used to to populate the Left and Right values, the explanation for how it work is contained in the comments:

```
-- Create the stack table
DECLARE @tmpStack TABLE
(
    ID INT IDENTITY(1, 1),
    EmployeeID INT NOT NULL,
    LeftExtent INT NULL,
    RightExtent INT NULL
)

-- what we do is start from a parent, in this instance
-- we want to start from the very top which is the NULL parent
DECLARE @parentid AS INTEGER
SET @parentid = NULL

-- our counter is the variable used to set the left and right extent
DECLARE @counter AS INTEGER
SET @counter = 1

-- what we do is go check the parentid for children, if it
-- has children we push
-- it into the stack and move on to the next child
DECLARE @id AS INTEGER
DECLARE @oldid AS INTEGER

SET NOCOUNT ON

WHILE 1 = 1
BEGIN
    SET @id = NULL

    -- get the first row which is a child of @parentid, which has not already
    -- been pushed onto the stack
    INSERT INTO @tmpStack
    (
        EmployeeID,
        LeftExtent
    )
    SELECT TOP 1 EmployeeID, @counter
    FROM Employee
    WHERE ISNULL(ParentID, 0) = ISNULL(@parentid, 0)
    AND EmployeeID NOT IN (SELECT EmployeeID FROM @tmpStack)
```

```

-- just check we are getting the right identity value
-- this value does not get reset once its reas so we
-- need to see if its the same as the older one
SET @id = SCOPE_IDENTITY()
IF ISNULL(@id, 0) = ISNULL(@oldid, 0)
    SET @id = NULL
ELSE
BEGIN
    SET @oldid = @id
    SET @counter = @counter + 1
END

-- if we have children (i.e if this operation inserted a row,
-- then we carry on
IF @id IS NULL
BEGIN
    -- no rows left, which means we want to pop the last item
    SELECT TOP 1 @id = ID FROM @tmpStack
    WHERE RightExtent IS NULL ORDER BY ID DESC

    -- there are no items left to pop, so exit the procedure
    IF @id IS NULL
        BREAK

    UPDATE @tmpStack
    SET RightExtent = @counter
    WHERE ID = @id

    SET @counter = @counter + 1

    SELECT @parentid = ParentID
    FROM Employee WHERE EmployeeID =
        (SELECT EmployeeID FROM @tmpStack WHERE ID = @id)
END
ELSE
BEGIN
    -- this is easy enough, move on to the next level
    -- we want the parent id of the item just inserted
    SELECT @parentid = EmployeeID FROM @tmpStack WHERE ID = @id
END
END

-- And update all the Items in Hier_Dept
-- in the original hierarchy table
UPDATE e
SET LeftExtent = ts.LeftExtent,
    RightExtent = ts.RightExtent
FROM Employee e
INNER JOIN @tmpStack ts
ON ts.EmployeeID = e.EmployeeID

GO

```

The logic to the above code is simple enough, if you imagine the hierarchy as its original structure, the left and right values need to be numbered as if you were tracing a line around the outside of the tree e.g.:

Below are some more examples or using the left and right values to query the data:

```
-- Which level of the hierarchy are the employees on
SELECT COUNT(P2.EmployeeID) AS level,
       P1.FirstName,
       P1.LeftExtent ,
       P1.RightExtent
FROM Employee AS P1
INNER JOIN Employee AS P2
  ON (P1.LeftExtent BETWEEN P2.LeftExtent AND P2.RightExtent)
GROUP BY P1.FirstName, P1.LeftExtent, P1.RightExtent
ORDER BY Level ASC

-- counting the employees of a manager (children of a node)
SELECT p1.FirstName,
       COUNT(p2.EmployeeID) Employees
FROM Employee p1
INNER JOIN Employee p2
  ON (p2.LeftExtent BETWEEN p1.LeftExtent AND p1.RightExtent
      AND p2.LeftExtent <> p1.LeftExtent)
GROUP BY p1.FirstName
ORDER BY Employees DESC
```

That's about it. Nested sets in a nut-shell. I hope you have found the code useful and interesting. If you need more detailed information about the theory there are plenty of SQL websites around which will have some information on this subject. Any corrections please don't hesitate to email me!

James Simpson

Click [here](#) to view James Simpson's online profile.

Other popular articles:

- [Exposing tabular data from your COM object - Part 2](#)
The ATL OLE DB Provider templates appear to rely on the fact that your data is kept in a simple array, but that's not really the case at all!
- [A set of ADOX Classes](#)
Simple database catalog access using a set of ADOX classes
- [ADO Connection Strings](#)
A list of ODBC DSN Connection Strings.
- [Hands on how to configure the Microsoft MSDE](#)
An article on how to setup and configure the MSFT MSDE 2000.

[\[Top\]](#)

[Sign in](#) to vote for this article: *Poor* ○ ○ ○ ○ ○ *Excellent*

[Vote](#)

Premium Sponsor



Free customer support for the life of the software. Never pay for customer support again!

Memory Validator
Software Verification
Software solutions for software problems

Hint: For a faster board use IE 4+ or Mozilla, choose 'Message View' from the View dropdown and hit 'Set Options'.



[FAQ](#)

Noise level

Medium



[Search comments](#)

[Set Options](#)

View

Normal (slow)

Per page

25



[New thread](#)

Msgs 1 to 6 of 6 (Total: 6) ([Refresh](#))

[\[First\]](#) [\[Prev\]](#) [\[Next\]](#) [\[\]](#)

Subject	Author	Date
How to insert new records into set's?	Ishigava	5:15 6 Aug '03
Re: How to insert new records into set's?	Hugo Hallman	9:15 2 Jan '04
Re: How to insert new records into set's?	Mike Ošky	18:49 11 Jan '04
Re: How to insert new records into set's?	Keith Bogart	14:57 27 Aug '04
Good Joe's idea :-)	oleg_cherkasenko	7:27 24 May '03
Would be shameful if this wasn't well-known.	mrluc	13:13 26 Oct '04

Last Visit: 0:15 Wednesday 23rd March, 2005

[\[First\]](#) [\[Prev\]](#) [\[Next\]](#) [\[\]](#)

[All Topics](#), [MFC / C++](#) >> [Database](#) >> [SQL](#)

Updated: 18 May 2003

Article content copyright James Simpson, 2003
everything else Copyright © [CodeProject](#), 1999-2005.
[Advertise on The Code Project](#) | [Privacy](#)

[MSDN Communities](#) | [ASP Alliance](#) • [Developer Fusion](#) • [Developersdex](#) • [DevGuru](#) • [Programmers Heaven](#) • [Planet Source Code](#) • [Tek-Tips Forums](#) •