How to create fast database queries

**[Adjacency list vs. nested sets: SQL Server](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

FROM: <http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/>

Continuing the series:

What is better to store hierarchical data: **nested sets** model or **adjacency list** (parent-child) model?

For detailed explanations of the terms, see the first article in the series:

* **[Adjacency list vs. nested sets: PostgreSQL](http://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/)**

Now, let's see what's better for **SQL Server**.

We will create a single table that holds both adjacency list data and nested sets data, with **8** levels of nesting, **5** children of each parent node and **2,441,405** records:  
  
**[Table creation details](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.CREATE SCHEMA [20090925\_nested]

02.CREATE TABLE [20090925\_nested].t\_hierarchy (

03.        id INT NOT NULL PRIMARY KEY,

04.        parent INT NOT NULL,

05.        lft INT NOT NULL,

06.        rgt INT NOT NULL,

07.        data VARCHAR(100) NOT NULL,

08.        stuffing VARCHAR(100) NOT NULL

09.)

10.GO

11.WITH    nums AS

12.        (

13.        SELECT  1 AS num

14.        UNION ALL

15.        SELECT  num + 1

16.        FROM    nums

17.        WHERE   num < 5

18.        ),

19.        q AS

20.        (

21.        SELECT  num AS id, 0 AS parent,

22.                1 + 585937 \* (num - 1) AS lft,

23.                1 + 585937 \* num - 1 AS rgt,

24.                FLOOR(585937 / 5) AS width

25.        FROM    nums

26.        UNION ALL

27.        SELECT  id \* 5 + num, id,

28.                1 + lft + width \* (num - 1),

29.                1 + lft + width \* num - 1,

30.                FLOOR(width / 5)

31.        FROM    q

32.        CROSS JOIN

33.                nums

34.        WHERE   width > 1

35.        )

36.INSERT

37.INTO    [20090925\_nested].t\_hierarchy

38.SELECT  id, parent, lft, rgt, 'Value ' + CAST(id AS VARCHAR), REPLICATE('\*', 100)

39.FROM    q

40.

41.CREATE INDEX IX\_hierarchy\_parent ON [20090925\_nested].t\_hierarchy (parent)

42.CREATE INDEX IX\_hierarchy\_lft ON [20090925\_nested].t\_hierarchy (lft)

43.CREATE INDEX IX\_hierarchy\_rgt ON [20090925\_nested].t\_hierarchy (rgt)

We will test the performance of **3** most common operations:

* Find all descendants of a given node
* Find all ancestors of a given node
* Find all descendants of a given node up to a certain depth

**All descendants**

We will select all descendants of a given node (which is close to the root node and has lots of descendants).

Since there are more than **100,000** descendants, we will select the aggregated length of the stuffing columns rather than a huge resultset of values. All rows are required to be visited anyway, so this is quite good performance test.

**Nested sets**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

1.SELECT  SUM(LEN(hc.stuffing))

2.FROM    [20090925\_nested].t\_hierarchy hp

3.JOIN    [20090925\_nested].t\_hierarchy hc

4.ON      hc.lft BETWEEN hp.lft AND hp.rgt

5.WHERE   hp.id = 42

**[View query details](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

|  |
| --- |
|  |
| 1953100 |
| 1 row fetched in 0.0002s (0.0816s) | |

Table 't\_hierarchy'. Scan count 1, logical reads 59856, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 78 ms, elapsed time = 81 ms.

|--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [globalagg1007]=(0) THEN NULL ELSE [globalagg1009] END))

|--Stream Aggregate(DEFINE:([globalagg1007]=SUM([partialagg1006]), [globalagg1009]=SUM([partialagg1008])))

|--Parallelism(Gather Streams)

|--Stream Aggregate(DEFINE:([partialagg1006]=COUNT\_BIG([Expr1005]), [partialagg1008]=SUM([Expr1005])))

|--Compute Scalar(DEFINE:([Expr1005]=len([test].[20090925\_nested].[t\_hierarchy].[stuffing] as [hc].[stuffing])))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[id], [Expr1011]) WITH UNORDERED PREFETCH)

|--Sort(ORDER BY:([hc].[id] ASC))

| |--Nested Loops(Inner Join, OUTER REFERENCES:([hp].[lft], [hp].[rgt]))

| |--Parallelism(Distribute Streams, RoundRobin Partitioning)

| | |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hp]), SEEK:([hp].[id]=(42)) ORDERED FORWARD)

| |--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_lft] AS [hc]), SEEK:([hc].[lft] >= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hp].[lft] AND [hc].[lft] <= [test].[20090925\_nested].[t\_hierarchy].[rgt] as [hp].[rgt]) ORDERED FORWARD)

|--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id]) LOOKUP ORDERED FORWARD)

Nested sets model is good for this query, since it required only a single range scan on lft to get all descendants.

This query completes in **81 ms**.

**Adjacency list**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.WITH    q AS

02.        (

03.        SELECT  id, stuffing

04.        FROM    [20090925\_nested].t\_hierarchy h

05.        WHERE   id = 42

06.        UNION ALL

07.        SELECT  hc.id, hc.stuffing

08.        FROM    q

09.        JOIN    [20090925\_nested].t\_hierarchy hc

10.        ON      hc.parent = q.id

11.        )

12.SELECT  SUM(LEN(stuffing))

13.FROM    q

**[View query details](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

|  |
| --- |
|  |
| 1953100 |
| 1 row fetched in 0.0002s (0.5312s) | |

Table 'Worktable'. Scan count 2, logical reads 117188, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 't\_hierarchy'. Scan count 19531, logical reads 117218, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 500 ms, elapsed time = 531 ms.

|--Compute Scalar(DEFINE:([Expr1008]=CASE WHEN [Expr1018]=(0) THEN NULL ELSE [Expr1019] END))

|--Stream Aggregate(DEFINE:([Expr1018]=COUNT\_BIG(len([Recr1007])), [Expr1019]=SUM(len([Recr1007]))))

|--Index Spool(WITH STACK)

|--Concatenation

|--Compute Scalar(DEFINE:([Expr1014]=(0)))

| |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [h]), SEEK:([h].[id]=(42)) ORDERED FORWARD)

|--Assert(WHERE:(CASE WHEN [Expr1016]>(100) THEN (0) ELSE NULL END))

|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1016], [Recr1002], [Recr1003]))

|--Compute Scalar(DEFINE:([Expr1016]=[Expr1015]+(1)))

| |--Table Spool(WITH STACK)

|--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[id]) OPTIMIZED)

|--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_parent] AS [hc]), SEEK:([hc].[parent]=[Recr1002]) ORDERED FORWARD)

|--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id]) LOOKUP ORDERED FORWARD)

This query makes several range scans (one per node) using recursion.

These ranges finally sum up to the same set of values (of course), however, the parents of descendants are not contiguous and no single range can select them all. That's why this query is less efficient than nested sets and runs for **531 ms**, or about **6** times as long.

**All ancestors**

**Nested sets**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

1.SELECT  hp.id, hp.parent, hp.lft, hp.rgt, hp.data

2.FROM    [20090925\_nested].t\_hierarchy hc

3.JOIN    [20090925\_nested].t\_hierarchy hp

4.ON      hc.lft BETWEEN hp.lft AND hp.rgt

5.WHERE   hc.id = 1000000

6.ORDER BY

7.        hp.lft

**[View query details](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **parent** | **lft** | **rgt** | **data** |
| 2 | 0 | 585938 | 1171874 | Value 2 |
| 12 | 2 | 703126 | 820312 | Value 12 |
| 63 | 12 | 750001 | 773437 | Value 63 |
| 319 | 63 | 764063 | 768749 | Value 319 |
| 1599 | 319 | 766875 | 767811 | Value 1599 |
| 7999 | 1599 | 767437 | 767623 | Value 7999 |
| 39999 | 7999 | 767549 | 767585 | Value 39999 |
| 199999 | 39999 | 767571 | 767577 | Value 199999 |
| 1000000 | 199999 | 767576 | 767576 | Value 1000000 |
| 9 rows fetched in 0.0006s (16.8588s) | | | | | |

Table 't\_hierarchy'. Scan count 1, logical reads 2036352, physical reads 10716, read-ahead reads 1991, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 4390 ms, elapsed time = 16856 ms.

|--Parallelism(Gather Streams, ORDER BY:([hp].[lft] ASC))

|--Sort(ORDER BY:([hp].[lft] ASC))

|--Filter(WHERE:([test].[20090925\_nested].[t\_hierarchy].[lft] as [hc].[lft]<=[test].[20090925\_nested].[t\_hierarchy].[rgt] as [hp].[rgt]))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hp].[id], [Expr1004]) WITH UNORDERED PREFETCH)

|--Sort(ORDER BY:([hp].[id] ASC))

| |--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[lft]))

| |--Parallelism(Distribute Streams, RoundRobin Partitioning)

| | |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=(1000000)) ORDERED FORWARD)

| |--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_lft] AS [hp]), SEEK:([hp].[lft] <= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hc].[lft]) ORDERED FORWARD)

|--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hp]), SEEK:([hp].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hp].[id]) LOOKUP ORDERED FORWARD)

This is extremely inefficient: **16 seconds**.

Unlike **PostgreSQL**, **SQL Server** is not capable of building in-memory bitmaps from the index results. It just has to seek the index and join. And the join condition used here is not an equality, thus making all join methods except NESTED LOOPS impossible to use.

All this results in a very poor performance which is far beyond that of even a marginally usable query.

**Adjacency list**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.WITH    q AS

02.        (

03.        SELECT  h.\*, 1 AS level

04.        FROM    [20090925\_nested].t\_hierarchy h

05.        WHERE   id = 1000000

06.        UNION ALL

07.        SELECT  hp.\*, level + 1

08.        FROM    q

09.        JOIN    [20090925\_nested].t\_hierarchy hp

10.        ON      hp.id = q.parent

11.        )

12.SELECT  id, parent, lft, rgt, data

13.FROM    q

14.ORDER BY

15.        level DESC

**[View query details](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **parent** | **lft** | **rgt** | **data** |
| 2 | 0 | 585938 | 1171874 | Value 2 |
| 12 | 2 | 703126 | 820312 | Value 12 |
| 63 | 12 | 750001 | 773437 | Value 63 |
| 319 | 63 | 764063 | 768749 | Value 319 |
| 1599 | 319 | 766875 | 767811 | Value 1599 |
| 7999 | 1599 | 767437 | 767623 | Value 7999 |
| 39999 | 7999 | 767549 | 767585 | Value 39999 |
| 199999 | 39999 | 767571 | 767577 | Value 199999 |
| 1000000 | 199999 | 767576 | 767576 | Value 1000000 |
| 9 rows fetched in 0.0006s (0.0012s) | | | | | |

Table 'Worktable'. Scan count 2, logical reads 56, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 't\_hierarchy'. Scan count 0, logical reads 30, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 1 ms.

|--Sort(ORDER BY:([Recr1019] DESC))

|--Index Spool(WITH STACK)

|--Concatenation

|--Compute Scalar(DEFINE:([Expr1020]=(0)))

| |--Compute Scalar(DEFINE:([Expr1002]=(1)))

| |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [h]), SEEK:([h].[id]=(1000000)) ORDERED FORWARD)

|--Assert(WHERE:(CASE WHEN [Expr1022]>(100) THEN (0) ELSE NULL END))

|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1022], [Recr1003], [Recr1004], [Recr1005], [Recr1006], [Recr1007], [Recr1008], [Recr1009]))

|--Compute Scalar(DEFINE:([Expr1022]=[Expr1021]+(1)))

| |--Table Spool(WITH STACK)

|--Compute Scalar(DEFINE:([Expr1012]=[Recr1009]+(1)))

|--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hp]), SEEK:([hp].[id]=[Recr1004]) ORDERED FORWARD)

Now, this query is a boon for adjacency list model. There are **9** ancestors that can be retrieved in **9** clustered index seeks and these **9** values then need to be sorted. **SQL Server** eats such queries for breakfast.

This query completes in **1 ms** which is the range of time measurement error in **SQL Server** statistics.

In other words, this query is very very fast.

**Descendants up to a given level**

Since the performance on this task may depend on the number of the descendants, we will run **2** queries for each model.

This first query selects descendants of a node that is close to the root (node **42**), the second one selects those of node **31,415** which is further from the root.

**Nested sets**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.SELECT  hc.id, hc.parent, hc.lft, hc.rgt, hc.data

02.FROM    [20090925\_nested].t\_hierarchy hp

03.JOIN    [20090925\_nested].t\_hierarchy hc

04.ON      hc.lft BETWEEN hp.lft AND hp.rgt

05.WHERE   hp.id = ?

06.        AND

07.        (

08.        SELECT  COUNT(\*)

09.        FROM    [20090925\_nested].t\_hierarchy hn

10.        WHERE   hc.lft BETWEEN hn.lft AND hn.rgt

11.                AND hn.lft BETWEEN hp.lft AND hp.rgt

12.        ) <= 3

**[View results for node 42](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.SELECT  hc.id, hc.parent, hc.lft, hc.rgt, hc.data

02.FROM    [20090925\_nested].t\_hierarchy hp

03.JOIN    [20090925\_nested].t\_hierarchy hc

04.ON      hc.lft BETWEEN hp.lft AND hp.rgt

05.WHERE   hp.id = 42

06.        AND

07.        (

08.        SELECT  COUNT(\*)

09.        FROM    [20090925\_nested].t\_hierarchy hn

10.        WHERE   hc.lft BETWEEN hn.lft AND hn.rgt

11.                AND hn.lft BETWEEN hp.lft AND hp.rgt

12.        ) <= 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **parent** | **lft** | **rgt** | **data** |
| 42 | 8 | 257814 | 281250 | Value 42 |
| 211 | 42 | 257815 | 262501 | Value 211 |
| 212 | 42 | 262502 | 267188 | Value 212 |
| 213 | 42 | 267189 | 271875 | Value 213 |
| 214 | 42 | 271876 | 276562 | Value 214 |
| 215 | 42 | 276563 | 281249 | Value 215 |
| 1056 | 211 | 257816 | 258752 | Value 1056 |
| 1057 | 211 | 258753 | 259689 | Value 1057 |
| 1058 | 211 | 259690 | 260626 | Value 1058 |
| 1059 | 211 | 260627 | 261563 | Value 1059 |
| 1060 | 211 | 261564 | 262500 | Value 1060 |
| 1061 | 212 | 262503 | 263439 | Value 1061 |
| 1062 | 212 | 263440 | 264376 | Value 1062 |
| 1063 | 212 | 264377 | 265313 | Value 1063 |
| 1064 | 212 | 265314 | 266250 | Value 1064 |
| 1065 | 212 | 266251 | 267187 | Value 1065 |
| 1066 | 213 | 267190 | 268126 | Value 1066 |
| 1067 | 213 | 268127 | 269063 | Value 1067 |
| 1068 | 213 | 269064 | 270000 | Value 1068 |
| 1069 | 213 | 270001 | 270937 | Value 1069 |
| 1070 | 213 | 270938 | 271874 | Value 1070 |
| 1071 | 214 | 271877 | 272813 | Value 1071 |
| 1072 | 214 | 272814 | 273750 | Value 1072 |
| 1073 | 214 | 273751 | 274687 | Value 1073 |
| 1074 | 214 | 274688 | 275624 | Value 1074 |
| 1075 | 214 | 275625 | 276561 | Value 1075 |
| 1076 | 215 | 276564 | 277500 | Value 1076 |
| 1077 | 215 | 277501 | 278437 | Value 1077 |
| 1078 | 215 | 278438 | 279374 | Value 1078 |
| 1079 | 215 | 279375 | 280311 | Value 1079 |
| 1080 | 215 | 280312 | 281248 | Value 1080 |
| 31 rows fetched in 0.0000s (103.1061s) | | | | | |

Table 't\_hierarchy'. Scan count 2, logical reads 119875, physical reads 0, read-ahead reads 56945, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 19531, logical reads 7970942, physical reads 1017, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 71750 ms, elapsed time = 103096 ms.

|--Parallelism(Gather Streams)

|--Filter(WHERE:([Expr1006]<=(3)))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hp].[lft], [hp].[rgt], [hc].[lft]))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[id], [Expr1010]) WITH UNORDERED PREFETCH)

| |--Sort(ORDER BY:([hc].[id] ASC))

| | |--Nested Loops(Inner Join, OUTER REFERENCES:([hp].[lft], [hp].[rgt]))

| | |--Parallelism(Distribute Streams, RoundRobin Partitioning)

| | | |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hp]), SEEK:([hp].[id]=(42)) ORDERED FORWARD)

| | |--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_lft] AS [hc]), SEEK:([hc].[lft] >= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hp].[lft] AND [hc].[lft] <= [test].[20090925\_nested].[t\_hierarchy].[rgt] as [hp].[rgt]) ORDERED FORWARD)

| |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id]) LOOKUP ORDERED FORWARD)

|--Compute Scalar(DEFINE:([Expr1006]=CONVERT\_IMPLICIT(int,[Expr1011],0)))

|--Stream Aggregate(DEFINE:([Expr1011]=Count(\*)))

|--Filter(WHERE:([test].[20090925\_nested].[t\_hierarchy].[lft] as [hc].[lft]<=[test].[20090925\_nested].[t\_hierarchy].[rgt] as [hn].[rgt] AND [test].[20090925\_nested].[t\_hierarchy].[lft] as [hn].[lft]<=[test].[20090925\_nested].[t\_hierarchy].[rgt] as [hp].[rgt]))

|--Index Spool(SEEK:([hn].[lft] >= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hp].[lft] AND [hn].[lft] <= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hc].[lft]))

|--Clustered Index Scan(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hn]))

**[View results for node 31,415](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.SELECT  hc.id, hc.parent, hc.lft, hc.rgt, hc.data

02.FROM    [20090925\_nested].t\_hierarchy hp

03.JOIN    [20090925\_nested].t\_hierarchy hc

04.ON      hc.lft BETWEEN hp.lft AND hp.rgt

05.WHERE   hp.id = 31415

06.        AND

07.        (

08.        SELECT  COUNT(\*)

09.        FROM    [20090925\_nested].t\_hierarchy hn

10.        WHERE   hc.lft BETWEEN hn.lft AND hn.rgt

11.                AND hn.lft BETWEEN hp.lft AND hp.rgt

12.        ) <= 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **parent** | **lft** | **rgt** | **data** |
| 31415 | 6282 | 445651 | 445687 | Value 31415 |
| 157076 | 31415 | 445652 | 445658 | Value 157076 |
| 157077 | 31415 | 445659 | 445665 | Value 157077 |
| 157078 | 31415 | 445666 | 445672 | Value 157078 |
| 157079 | 31415 | 445673 | 445679 | Value 157079 |
| 157080 | 31415 | 445680 | 445686 | Value 157080 |
| 785381 | 157076 | 445653 | 445653 | Value 785381 |
| 785382 | 157076 | 445654 | 445654 | Value 785382 |
| 785383 | 157076 | 445655 | 445655 | Value 785383 |
| 785384 | 157076 | 445656 | 445656 | Value 785384 |
| 785385 | 157076 | 445657 | 445657 | Value 785385 |
| 785386 | 157077 | 445660 | 445660 | Value 785386 |
| 785387 | 157077 | 445661 | 445661 | Value 785387 |
| 785388 | 157077 | 445662 | 445662 | Value 785388 |
| 785389 | 157077 | 445663 | 445663 | Value 785389 |
| 785390 | 157077 | 445664 | 445664 | Value 785390 |
| 785391 | 157078 | 445667 | 445667 | Value 785391 |
| 785392 | 157078 | 445668 | 445668 | Value 785392 |
| 785393 | 157078 | 445669 | 445669 | Value 785393 |
| 785394 | 157078 | 445670 | 445670 | Value 785394 |
| 785395 | 157078 | 445671 | 445671 | Value 785395 |
| 785396 | 157079 | 445674 | 445674 | Value 785396 |
| 785397 | 157079 | 445675 | 445675 | Value 785397 |
| 785398 | 157079 | 445676 | 445676 | Value 785398 |
| 785399 | 157079 | 445677 | 445677 | Value 785399 |
| 785400 | 157079 | 445678 | 445678 | Value 785400 |
| 785401 | 157080 | 445681 | 445681 | Value 785401 |
| 785402 | 157080 | 445682 | 445682 | Value 785402 |
| 785403 | 157080 | 445683 | 445683 | Value 785403 |
| 785404 | 157080 | 445684 | 445684 | Value 785404 |
| 785405 | 157080 | 445685 | 445685 | Value 785405 |
| 31 rows fetched in 0.0000s (38.1238s) | | | | | |

Table 't\_hierarchy'. Scan count 2, logical reads 60129, physical reads 4, read-ahead reads 56653, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 31, logical reads 7268474, physical reads 1274, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 11235 ms, elapsed time = 38121 ms.

|--Parallelism(Gather Streams)

|--Filter(WHERE:([Expr1006]<=(3)))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hp].[lft], [hp].[rgt], [hc].[lft]))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[id], [Expr1010]) WITH UNORDERED PREFETCH)

| |--Sort(ORDER BY:([hc].[id] ASC))

| | |--Nested Loops(Inner Join, OUTER REFERENCES:([hp].[lft], [hp].[rgt]))

| | |--Parallelism(Distribute Streams, RoundRobin Partitioning)

| | | |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hp]), SEEK:([hp].[id]=(31415)) ORDERED FORWARD)

| | |--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_lft] AS [hc]), SEEK:([hc].[lft] >= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hp].[lft] AND [hc].[lft] <= [test].[20090925\_nested].[t\_hierarchy].[rgt] as [hp].[rgt]) ORDERED FORWARD)

| |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id]) LOOKUP ORDERED FORWARD)

|--Compute Scalar(DEFINE:([Expr1006]=CONVERT\_IMPLICIT(int,[Expr1011],0)))

|--Stream Aggregate(DEFINE:([Expr1011]=Count(\*)))

|--Filter(WHERE:([test].[20090925\_nested].[t\_hierarchy].[lft] as [hc].[lft]<=[test].[20090925\_nested].[t\_hierarchy].[rgt] as [hn].[rgt] AND [test].[20090925\_nested].[t\_hierarchy].[lft] as [hn].[lft]<=[test].[20090925\_nested].[t\_hierarchy].[rgt] as [hp].[rgt]))

|--Index Spool(SEEK:([hn].[lft] >= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hp].[lft] AND [hn].[lft] <= [test].[20090925\_nested].[t\_hierarchy].[lft] as [hc].[lft]))

|--Clustered Index Scan(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hn]))

These queries complete in **108 seconds** and **38 seconds**, respectively. Both these queries use nested loops with a full table scan which doesn't add to performance. This makes the queries unusable.

**Adjacency list**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.WITH    q AS

02.        (

03.        SELECT  id, parent, lft, rgt, data, 1 AS level,

04.                CAST(LEFT(CAST(id AS VARCHAR) + REPLICATE('0', 10), 10) AS VARCHAR) AS bc

05.        FROM    [20090925\_nested].t\_hierarchy hc

06.        WHERE   id = ?

07.        UNION ALL

08.        SELECT  hc.id, hc.parent, hc.lft, hc.rgt, hc.data, level + 1,

09.                CAST(bc + '.' + LEFT(CAST(hc.id AS VARCHAR) + REPLICATE('0', 10), 10) AS VARCHAR)

10.        FROM    q

11.        JOIN    [20090925\_nested].t\_hierarchy hc

12.        ON      hc.parent = q.id

13.        WHERE   level < 3

14.        )

15.SELECT  id, parent, lft, rgt, data

16.FROM    q

17.ORDER BY

18.        bc

The field bc defines breadcrumbs which can be used to return the nodes in tree order.

In **PostgreSQL 8.4**, we used an array for that.

**SQL Server** lacks array type, that's why we need to use a hack: converting the id's into zero-padded strings of fixed length (10 characters) and concatenating them.

Here are the query results:

**[View results for node 42](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.WITH    q AS

02.        (

03.        SELECT  id, parent, lft, rgt, data, 1 AS level,

04.                CAST(LEFT(CAST(id AS VARCHAR) + REPLICATE('0', 10), 10) AS VARCHAR) AS bc

05.        FROM    [20090925\_nested].t\_hierarchy hc

06.        WHERE   id = 42

07.        UNION ALL

08.        SELECT  hc.id, hc.parent, hc.lft, hc.rgt, hc.data, level + 1,

09.                CAST(bc + '.' + LEFT(CAST(hc.id AS VARCHAR) + REPLICATE('0', 10), 10) AS VARCHAR)

10.        FROM    q

11.        JOIN    [20090925\_nested].t\_hierarchy hc

12.        ON      hc.parent = q.id

13.        WHERE   level < 3

14.        )

15.SELECT  id, parent, lft, rgt, data

16.FROM    q

17.ORDER BY

18.        bc

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **parent** | **lft** | **rgt** | **data** |
| 42 | 8 | 257814 | 281250 | Value 42 |
| 211 | 42 | 257815 | 262501 | Value 211 |
| 1056 | 211 | 257816 | 258752 | Value 1056 |
| 1057 | 211 | 258753 | 259689 | Value 1057 |
| 1058 | 211 | 259690 | 260626 | Value 1058 |
| 1059 | 211 | 260627 | 261563 | Value 1059 |
| 1060 | 211 | 261564 | 262500 | Value 1060 |
| 212 | 42 | 262502 | 267188 | Value 212 |
| 1061 | 212 | 262503 | 263439 | Value 1061 |
| 1062 | 212 | 263440 | 264376 | Value 1062 |
| 1063 | 212 | 264377 | 265313 | Value 1063 |
| 1064 | 212 | 265314 | 266250 | Value 1064 |
| 1065 | 212 | 266251 | 267187 | Value 1065 |
| 213 | 42 | 267189 | 271875 | Value 213 |
| 1066 | 213 | 267190 | 268126 | Value 1066 |
| 1067 | 213 | 268127 | 269063 | Value 1067 |
| 1068 | 213 | 269064 | 270000 | Value 1068 |
| 1069 | 213 | 270001 | 270937 | Value 1069 |
| 1070 | 213 | 270938 | 271874 | Value 1070 |
| 214 | 42 | 271876 | 276562 | Value 214 |
| 1071 | 214 | 271877 | 272813 | Value 1071 |
| 1072 | 214 | 272814 | 273750 | Value 1072 |
| 1073 | 214 | 273751 | 274687 | Value 1073 |
| 1074 | 214 | 274688 | 275624 | Value 1074 |
| 1075 | 214 | 275625 | 276561 | Value 1075 |
| 215 | 42 | 276563 | 281249 | Value 215 |
| 1076 | 215 | 276564 | 277500 | Value 1076 |
| 1077 | 215 | 277501 | 278437 | Value 1077 |
| 1078 | 215 | 278438 | 279374 | Value 1078 |
| 1079 | 215 | 279375 | 280311 | Value 1079 |
| 1080 | 215 | 280312 | 281248 | Value 1080 |
| 31 rows fetched in 0.0018s (0.0023s) | | | | | |

Table 'Worktable'. Scan count 2, logical reads 188, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 't\_hierarchy'. Scan count 6, logical reads 111, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 2 ms.

|--Sort(ORDER BY:([Recr1021] ASC))

|--Index Spool(WITH STACK)

|--Concatenation

|--Compute Scalar(DEFINE:([Expr1023]=(0)))

| |--Compute Scalar(DEFINE:([Expr1002]=(1)))

| |--Compute Scalar(DEFINE:([Expr1003]=CONVERT(varchar(30),substring(CONVERT(varchar(30),[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id],0)+'0000000000',(1),(10)),0)))

| |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=(42)) ORDERED FORWARD)

|--Assert(WHERE:(CASE WHEN [Expr1025]>(100) THEN (0) ELSE NULL END))

|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1025], [Recr1004], [Recr1005], [Recr1006], [Recr1007], [Recr1008], [Recr1009], [Recr1010]))

|--Compute Scalar(DEFINE:([Expr1025]=[Expr1024]+(1)))

| |--Table Spool(WITH STACK)

|--Compute Scalar(DEFINE:([Expr1013]=[Recr1009]+(1), [Expr1014]=CONVERT(varchar(30),([Recr1010]+'.')+[Expr1022],0)))

|--Compute Scalar(DEFINE:([Expr1022]=substring(CONVERT(varchar(30),[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id],0)+'0000000000',(1),(10))))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[id]) OPTIMIZED)

|--Filter(WHERE:(STARTUP EXPR([Recr1009]<(3))))

| |--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_parent] AS [hc]), SEEK:([hc].[parent]=[Recr1004]) ORDERED FORWARD)

|--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id]) LOOKUP ORDERED FORWARD)

**[View results for node 31,415](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/)**

[view source](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "viewSource" \o "view source)[print](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "printSource" \o "print)[?](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "about" \o "?)

01.WITH    q AS

02.        (

03.        SELECT  id, parent, lft, rgt, data, 1 AS level,

04.                CAST(LEFT(CAST(id AS VARCHAR) + REPLICATE('0', 10), 10) AS VARCHAR) AS bc

05.        FROM    [20090925\_nested].t\_hierarchy hc

06.        WHERE   id = 42

07.        UNION ALL

08.        SELECT  hc.id, hc.parent, hc.lft, hc.rgt, hc.data, level + 1,

09.                CAST(bc + '.' + LEFT(CAST(hc.id AS VARCHAR) + REPLICATE('0', 10), 10) AS VARCHAR)

10.        FROM    q

11.        JOIN    [20090925\_nested].t\_hierarchy hc

12.        ON      hc.parent = q.id

13.        WHERE   level < 3

14.        )

15.SELECT  id, parent, lft, rgt, data

16.FROM    q

17.ORDER BY

18.        bc

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **parent** | **lft** | **rgt** | **data** |
| 42 | 8 | 257814 | 281250 | Value 42 |
| 211 | 42 | 257815 | 262501 | Value 211 |
| 1056 | 211 | 257816 | 258752 | Value 1056 |
| 1057 | 211 | 258753 | 259689 | Value 1057 |
| 1058 | 211 | 259690 | 260626 | Value 1058 |
| 1059 | 211 | 260627 | 261563 | Value 1059 |
| 1060 | 211 | 261564 | 262500 | Value 1060 |
| 212 | 42 | 262502 | 267188 | Value 212 |
| 1061 | 212 | 262503 | 263439 | Value 1061 |
| 1062 | 212 | 263440 | 264376 | Value 1062 |
| 1063 | 212 | 264377 | 265313 | Value 1063 |
| 1064 | 212 | 265314 | 266250 | Value 1064 |
| 1065 | 212 | 266251 | 267187 | Value 1065 |
| 213 | 42 | 267189 | 271875 | Value 213 |
| 1066 | 213 | 267190 | 268126 | Value 1066 |
| 1067 | 213 | 268127 | 269063 | Value 1067 |
| 1068 | 213 | 269064 | 270000 | Value 1068 |
| 1069 | 213 | 270001 | 270937 | Value 1069 |
| 1070 | 213 | 270938 | 271874 | Value 1070 |
| 214 | 42 | 271876 | 276562 | Value 214 |
| 1071 | 214 | 271877 | 272813 | Value 1071 |
| 1072 | 214 | 272814 | 273750 | Value 1072 |
| 1073 | 214 | 273751 | 274687 | Value 1073 |
| 1074 | 214 | 274688 | 275624 | Value 1074 |
| 1075 | 214 | 275625 | 276561 | Value 1075 |
| 215 | 42 | 276563 | 281249 | Value 215 |
| 1076 | 215 | 276564 | 277500 | Value 1076 |
| 1077 | 215 | 277501 | 278437 | Value 1077 |
| 1078 | 215 | 278438 | 279374 | Value 1078 |
| 1079 | 215 | 279375 | 280311 | Value 1079 |
| 1080 | 215 | 280312 | 281248 | Value 1080 |
| 31 rows fetched in 0.0018s (0.0023s) | | | | | |

Table 'Worktable'. Scan count 2, logical reads 188, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 't\_hierarchy'. Scan count 6, logical reads 111, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 2 ms.

|--Sort(ORDER BY:([Recr1021] ASC))

|--Index Spool(WITH STACK)

|--Concatenation

|--Compute Scalar(DEFINE:([Expr1023]=(0)))

| |--Compute Scalar(DEFINE:([Expr1002]=(1)))

| |--Compute Scalar(DEFINE:([Expr1003]=CONVERT(varchar(30),substring(CONVERT(varchar(30),[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id],0)+'0000000000',(1),(10)),0)))

| |--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=(42)) ORDERED FORWARD)

|--Assert(WHERE:(CASE WHEN [Expr1025]>(100) THEN (0) ELSE NULL END))

|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1025], [Recr1004], [Recr1005], [Recr1006], [Recr1007], [Recr1008], [Recr1009], [Recr1010]))

|--Compute Scalar(DEFINE:([Expr1025]=[Expr1024]+(1)))

| |--Table Spool(WITH STACK)

|--Compute Scalar(DEFINE:([Expr1013]=[Recr1009]+(1), [Expr1014]=CONVERT(varchar(30),([Recr1010]+'.')+[Expr1022],0)))

|--Compute Scalar(DEFINE:([Expr1022]=substring(CONVERT(varchar(30),[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id],0)+'0000000000',(1),(10))))

|--Nested Loops(Inner Join, OUTER REFERENCES:([hc].[id]) OPTIMIZED)

|--Filter(WHERE:(STARTUP EXPR([Recr1009]<(3))))

| |--Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[IX\_hierarchy\_parent] AS [hc]), SEEK:([hc].[parent]=[Recr1004]) ORDERED FORWARD)

|--Clustered Index Seek(OBJECT:([test].[20090925\_nested].[t\_hierarchy].[PK\_\_t\_hierarchy\_\_49EEDF40] AS [hc]), SEEK:([hc].[id]=[test].[20090925\_nested].[t\_hierarchy].[id] as [hc].[id]) LOOKUP ORDERED FORWARD)

Both queries use the same efficient plans (which basically include **31** index seeks and ordering of the values fetched) and complete in **2 ms**, which may be considered instant.

**Summary**

We made an **SQL Server** table combining two popular models for hierarchical data: **adjacency list** and **nested sets** and tested three most used queries against these data:

1. Find all descendants of a given node
2. Find all ancestors of a given node
3. Find all descendants of a given node up to a certain depth

As in the similar test for **PostgreSQL 8.4** (conducted in the [previous article](http://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/)), nested sets model has shown a little performance benefit for the first query (finding all descendants).

However, **SQL Server**, unlike **PostgreSQL**, is not able to build in-memory bitmaps which makes nested loops (required for the queries **2** and **3** against the nested sets model) very inefficient.

This makes the nested sets model very slow for queries **2** and **3** in **SQL Server**. More or less significant number of records in the table makes these queries too slow to be of practical use.

Adjacency list, on the contrary, shows decent performance for the query **1** and amazing performance for the queries **2** and **3**.

This fact, along with the simplicity of adjacency list implementation and management, make **adjacency list** a preferred way to store hierarchical data in **SQL Server** versions that support recursive **CTE**s, that is versions from **2005** onwards.

**Share this:**

* [Facebook2](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/?share=facebook&nb=1" \o "Share on Facebook" \t "_blank)
* [Twitter](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/?share=twitter&nb=1" \o "Click to share on Twitter" \t "_blank)

Written by Quassnoi

September 25th, 2009 at 11:00 pm

Posted in [SQL Server](http://explainextended.com/category/sqlserver/)

« [Adjacency list vs. nested sets: PostgreSQL](http://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/)

[Adjacency list vs. nested sets: Oracle](http://explainextended.com/2009/09/28/adjacency-list-vs-nested-sets-oracle/) »

**5 Responses to 'Adjacency list vs. nested sets: SQL Server'**

Subscribe to comments with [RSS](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/feed/)

1. Hi Quassnoi,

Excellent article, I appreciate the depth to which you’ve investigated these structures. I’ve learned a great deal from reading this and other articles on your site. Most impressive!

I would like to understand further about your nested loop setup. I’ve made a few adjustments which result in much better performance (I’m speaking in relative terms as our equipment I’m sure is different), and would like your thoughts about what I’ve done.

First, in the environment I’m in (non-dag bills of materials (BOMs)), the most common use cases differ from what you have described. I would agree that “all descendants” is one case, and that “all ancestors” is another, but I would remove “descendants to a particular level” completely from the list (not just move it down in priority), and replace it with “immediate descendants”. Here is why. With a BOM, if the current row is called an “assembly”, then descendants can either be another “assembly” or a “component”. I will define an assembly as that which is made up of a collection of components. This fits well into the model of the tree structure, where components are leaf nodes and assemblies are non-leaf nodes. If we want to see a BOM for a given assembly, then we want “immediate descendants”, as it allows us to collect all the parts (assemblies or components) that we need to make the current assembly. If we wanted to see \_everything\_ that went into the highest level assembly, then we want “all descendants”. If we drill to a fixed layer, we get neither of these desired results, instead, its a useless mix.

Perhaps in some environments that have completely balanced hierarchies, the specific level is desired, but in this one its not. In fact, if the hierarchies were completely balanced, then would you need something so flexible as the nested loop or the adjacency list anyway?

Another note about this first point, when you have the environment where you don’t need the “particular level” query at all, the need for recursion goes away and allows MySQL and other non-recursive engines to play in the game again.

Next, if we stick with your nested loop query for “all ancestors”, the addition of an index on both lft and rgt columns improved my performance by ~15x. Something like this:

CREATE NONCLUSTERED INDEX IX\_hierarchy\_lft\_rgt ON [20090925\_nested].t\_hierarchy (lft,rgt)

Finally, the nested loop query for “all ancestors” should be simpler than what you have designed because there really is no need to join anything here (AFAIK). It should be just two comparisons in the WHERE clause:

DECLARE @clft INT; DECLARE @crgt INT;

SELECT @clft=lft, @crgt=rgt  
FROM [20090925\_nested].t\_hierarchy  
WHERE id=1000000;

SELECT id, parent, lft, rgt, data  
FROM [20090925\_nested].t\_hierarchy  
WHERE lft=@crgt  
ORDER BY lft;

Admittedly, this takes two separate queries, but all included, the performance increases another ~3x for a total of ~45x. In fact, even without the index this new query structure performs at the same level. Also, when I made these changes, my nested loop “all ancestors” query outperformed the adjacency list query, even if only by 20%.

Again, I appreciate you laying the groundwork and putting forth the effort required to document all this. You have motivated me to learn a great deal about these structures. I use the nested loop a lot in MySQL so I look forward to reading whatever comments you have about my suggestions.

ps1. All my experiments and numbers came from using SQL Server to stay in the same realm as this article, even though my typical platform is MySQL.

ps2. When I ran your setup query, I only got 480k rows instead of the 2.4M you got. I suspect there is a < / <= difference somewhere, but I didn't debug it.

http://2.gravatar.com/avatar/81d45d3798ce1a16f2f8f675b0f87a67?s=32&d=mm&r=g

**Mike Tallroth**

27 Oct 12 at [06:23](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "comment-2261)

1. Sorry, I must not have formatted my code properly, its supposed to have two comparisons in the final select statement. I’ll try again,

DECLARE @clft INT; DECLARE @crgt INT;

SELECT @clft=lft, @crgt=rgt  
FROM [20090925\_nested].t\_hierarchy  
WHERE id=1000000;

SELECT id, parent, lft, rgt, data  
FROM [20090925\_nested].t\_hierarchy  
WHERE lft <= @clft AND rgt >= @crgt  
ORDER BY lft;

http://2.gravatar.com/avatar/81d45d3798ce1a16f2f8f675b0f87a67?s=32&d=mm&r=g

**Mike Tallroth**

27 Oct 12 at [06:31](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "comment-2262)

1. @Mike: thanks for your input.

Could you please post the queries and response times when you are comparing performances? I. e. “I’ve run this query, it completed in X seconds, then I’ve run that query, it completed in Y seconds”.

Also please run these two commands:

SET STATISTICS TIME ON  
SET STATISTICS IO ON

prior to running the queries. This will make SQL Server output IO stats (which are crucial for understanding the query performance).

Thanks!

P.S. Just a side note: the model is called “nested sets”. “Nested loops” is a join algorithm used by the optimizer.

http://2.gravatar.com/avatar/b5428af26786d8751d34a7d96434ff18?s=32&d=mm&r=g

**[Quassnoi](http://explainextended.com)**

28 Oct 12 at [00:24](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "comment-2265)

1. @Quassnoi: Will do, and thanks for the feedback already.

I was looking at Client Statistics, not the Server Stats reporting that you showed me here. I’ve re-run and have posted the results below. I thought I was on to something because the client stats made it appear as though my method was faster than it really is. Also, I found a sequence to clear cache, etc which helps to stabilize the results.

The results in a nutshell;  
1) I didn’t get the AdjList algo down under 1ms per your numbers, but it is still the fastest as you suggest (were you clearing cache each time to get your numbers?),  
2) your Nested Set algo seems to be improved by using the multi-column index I suggested earlier,  
3) my Nested Set algo is unaffected by the extra index,  
4) my Nested Set algo is the slowest of the three options once the new index is applied to your NS algo.

Here are the details. First, I’ll iterate and label the queries, then I’ll show the sequence I used and the execution times that resulted.

[Create Table and Single Indexes] resulted in 480k rows.  
———————————————————————  
No changes from your queries.

[Clear Cache]  
probably contains some redundancy, or unnecessary bits in my case,  
but I had a hard time finding good references on this topic.  
———————————————————————  
CHECKPOINT;  
GO  
DBCC DROPCLEANBUFFERS;  
GO  
DBCC FREESYSTEMCACHE(‘ALL’);  
GO  
DBCC FREESESSIONCACHE;  
GO  
DBCC FREEPROCCACHE;  
GO

[Create Multi Index] Created 1 multi column index.  
———————————————————————  
CREATE NONCLUSTERED INDEX IX\_hierarchy\_lft\_rgt ON [20090925\_nested].t\_hierarchy (lft,rgt) include (id,stuffing);

[AA ExpExt NS] All ancestors from Explain Extended, Nested Set model.  
Modified only to accomodate smaller number of rows in table.  
———————————————————————  
SELECT hp.id, hp.parent, hp.lft, hp.rgt, hp.data  
FROM [20090925\_nested].t\_hierarchy hc  
JOIN [20090925\_nested].t\_hierarchy hp  
ON hc.lft BETWEEN hp.lft AND hp.rgt  
WHERE hc.id = 450000  
ORDER BY hp.lft

[AA Simple NS] All ancestors with simple, Nested Set model.  
———————————————————————  
DECLARE @clft INT; DECLARE @crgt INT;  
select @clft=lft, @crgt=rgt from [20090925\_nested].t\_hierarchy where id=450000;  
select id, parent, lft, rgt, data from [20090925\_nested].t\_hierarchy where lft <=@clft and rgt >=@crgt order by lft;

[AA ExpExt AdjList] All ancestors with Explain Extended, Adjacency List model.  
Modified only to accomodate smaller number of rows in table.  
———————————————————————  
WITH q AS  
(  
SELECT h.\*, 1 AS level  
FROM [20090925\_nested].t\_hierarchy h  
WHERE id = 450000  
UNION ALL  
SELECT hp.\*, level + 1  
FROM q  
JOIN [20090925\_nested].t\_hierarchy hp  
ON hp.id = q.parent  
)  
SELECT id, parent, lft, rgt, data  
FROM q  
ORDER BY level DESC

Sequence with times : Execution Elapsed Time  
————————————————–  
[Create Table and Single Indexes]  
[Clear Cache]  
[AA ExpExt AdjList] : 11m  
[Clear Cache]  
[AA ExpExt NS] : 1722m  
[Clear Cache]  
[AA Simple NS] : 621m (2m + 619m)

[Create Multi Index]  
[Clear Cache]  
[AA ExpExt AdjList] : 12m  
[Clear Cache]  
[AA ExpExt NS] : 306m  
[Clear Cache]  
[AA Simple NS] : 607m (2m + 605m)

So clearly not what I had originally thought, but still good as an exercise to show that Nested Sets can still play in the game. IMHO they are a more simplistic way of thinking about the problem, and the queries are “more straightforward”. But one cannot discount the performance difference, which would only compound as we increase the number of layers in the hierarchy.

If you could comment a little more on how your setup achieved the 0-1ms response time on the AdjList query, I would appreciate that.

thanks for your feedback and your great work on Explain Extended,  
Mike

http://2.gravatar.com/avatar/81d45d3798ce1a16f2f8f675b0f87a67?s=32&d=mm&r=g

**Mike Tallroth**

4 Nov 12 at [21:57](http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/" \l "comment-2279)

1. Hi Quassnoi,

thank you for the great explanation. I’ve tested it and found one issue concerning the wrong indexes which gave me a tremendous bost because of avoiding expensive key lookups.

CREATE INDEX IX\_hierarchy\_parent ON [20090925\_nested].t\_hierarchy (parent) INCLUDE (stuffing);  
CREATE INDEX IX\_hierarchy\_lft ON [20090925\_nested].t\_hierarchy (lft) INCLUDE (stuffing);  
CREATE INDEX IX\_hierarchy\_rgt ON [20090925\_nested].t\_hierarchy (rgt) INCLUDE (stuffing);

When I run both examples with id = 42 together the costs are separated by

97% for the “nested set”  
03% for the adjacency list

I was really impressed of the difference because I expected a really bad performance when using CTE ;)

@Mike,  
you don’t have to put Id to the index because it is the PK and – because not explicit disabled – the clustered key. Keep in mind that the cl is ALWAYS part of any NC-index!