

# Centura Pro

Visit us at [www.ProPublishing.com](http://www.ProPublishing.com)!

Hot Ideas for Centura® Developers

## Traversing Trees in SQLBase

### Part 2: The Nested Sets Approach

**R.J. David Burke**

In my March article, I presented a common technique for implementing trees and hierarchies in SQL called the adjacency matrix approach (see “Traversing Trees in SQLBase, Part 1: The Adjacency Matrix Approach”).

While easy to implement, the adjacency matrix approach has its shortcomings when you’re trying to write queries that need to navigate an arbitrary path through the tree. Unfortunately, such queries are commonly needed in business applications, for example, in aggregating the salaries of employees in a specific chain of command

(subtree). For more discussion of the issues, please see Joe Celko’s book, *SQL For Smarties* (Morgan Kaufmann Publishers, ISBN 1-55860-323-9).

As an alternative, Joe provides the nested sets approach (also known as the visitation approach) and a detailed discussion of its advantages. For a complete explanation, I suggest you read “Chapter 26: Trees” in his book. What I plan to do in this article is provide a package of stored procedures that shows how to support the nested sets approach in SQLBase.

#### Nested sets in tables

As in part 1 of this series, I will again use an emp table to store a set of employees. However, I’ve modified the table definition to accommodate the nested sets approach.

In the conclusion of this series we look at the nested sets approach (popularized by Joe Celko) for implementing trees and hierarchies in SQLBase databases.

Listing 1 provides the table definition.

Listing 1. An example of implementing a tree in SQL using nested sets in the table definition.

```
create table emp
(
  emp_id smallint,
  name varchar(20),
  salary decimal(8,2),
  lft smallint,
  rgt smallint
)
```

The lft and rgt columns are used to manage the nested sets. (The names lft and rgt are used instead of left and right to prevent any name conflicts or collisions with the potential names of other columns in the table.) The topmost employee will have a lft value of 1 and a rgt value larger than any other employee. Within the table, if a given employee *x* has lft and rgt values between the lft

May 1999

Volume 4, Number 5

- 1 Traversing Trees in SQLBase: Part 2: The Nested Sets Approach  
*R.J. David Burke*
- 2 Outside Looking In  
*Mark Hunter*
- 6 Centura Tip: ActiveX Library Confusion  
*Mark Hunter*
- 7 De-Scribe Your Reports  
*Thomas Althammer*
- 11 Fast Conversions are Now Easy, Too  
*Mark Hunter*



Continues on page 3

# Traversing Trees in SQLBase ...

Continued from page 1

and rgt values of another employee  $y$ , then  $x$  is in the subtree of  $y$ . A leaf node (an employee with no subordinates) will have  $lft + 1 = rgt$ . Another way of saying it is that lft to rgt shows an employee's region of control. Listing 2 shows the INSERT statement that will populate the emp table with the same organization structure used for the emp table from part 1 of this series.

**Listing 2.** Populating the emp table with data to match the organization structure used in the emp table from part 1 of this series.

```
insert into emp (emp_id, name,
salary, lft, rgt)
values (:1, :2, :3, :4, :5)
\
1,Albert,100000,1,28
2,Barry,90000,2,5
3,Charles,90000,6,19
4,Diane,91000,20,27
5,Edward,80000,3,4
6,Fred,80000,7,16
7,George,80000,17,18
8,Heidi,81000,21,26
9,Isabelle,71000,8,9
10,James,70000,10,15
11,Kathy,71000,22,23
12,Larry,70000,24,25
13,Mary,61000,11,12
14,Ned,60000,13,14
/
```

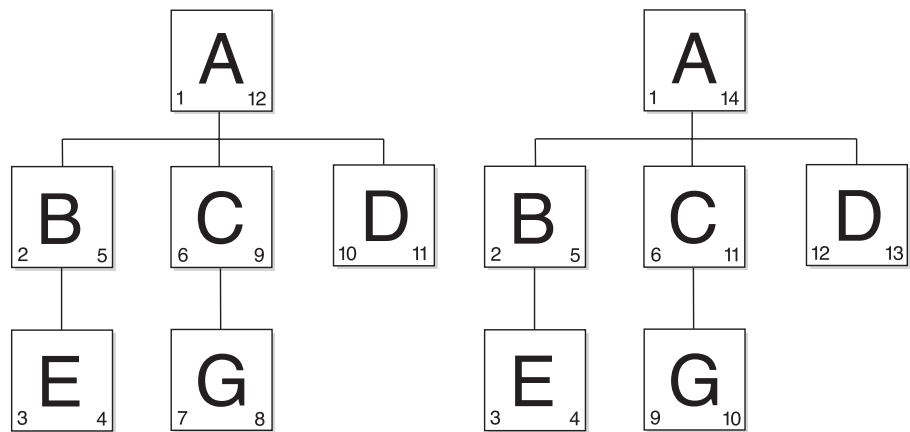
With the data used in Listing 2, Edward (lft 3, rgt 4) is managed by Barry (lft 2, rgt 5), who in turn is managed by Albert (lft 1, rgt 28). Albert is the highest executive and has no manager (in other words, he resides at the root node of the tree).

With the nested sets approach, various simple queries can be used to discover properties of the tree. For example, the code in Listing 3 shows how to find the highest executive and the bottom-most level of workers.

**Listing 3.** Simple queries to discover properties of the tree when using the nested sets approach.

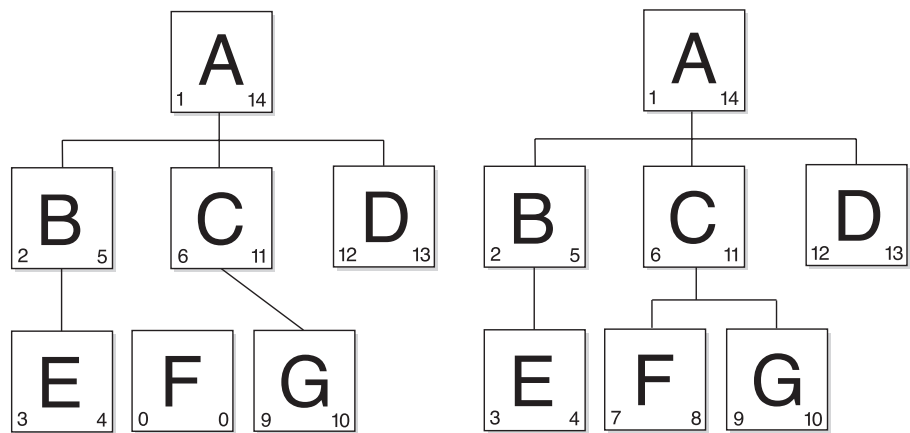
```
-- find root
select emp_id, name, salary
from emp
where lft = 1
/
-- find leaves
select emp_id, name, salary
from emp
where rgt - lft = 1
/
-- find leaves, optimized if there is an index on lft
select emp_id, name, salary
from emp
where lft = rgt - 1
/
```

## The Process for Inserting F as a Child of C



**Figure 1.** Existing hierarchical organization. Left and right values are in lower left and right corners respectively.

**Figure 2.** Expand nested set sizes to allow insertion of the new node.



**Figure 3.** Insert new node.

**Figure 4.** Update new node to be part of the nested node.

The real power of the nested sets approach is in how it handles queries that traverse the tree down a path. There's no need for recursion or procedural logic; it can easily be done in a SELECT statement with self-joins. [Listing 4](#) shows a query (and the result) for getting a payroll amount at the different levels of organization in the company.

**Listing 4.** Showing the power of the nested sets approach in aggregate queries.

```
select e1.name, sum(e1.salary) as payroll
from emp e1, emp e2
where e2.lft between e1.lft and e1.rgt
group by e1.name
/
```

E1.NAME	PAYROLL
Albert	140000
Barry	180000
Charles	630000
Diane	364000
Edward	80000
Fred	400000
George	80000
Heidi	243000
Isabelle	71000
James	210000
Kathy	71000
Larry	70000
Mary	61000
Ned	60000

14 ROWS SELECTED

### Managing nested sets

The main issue with the nested set approach is preserving the integrity of the lft and rgt values as employees are inserted and deleted from the tree. This is the meat of this article; to provide stored procedures for such operations.

### Inserting new nodes into the tree

When inserting new employees, the tree needs to be “widened” to accommodate the new employee. [Listing 5](#) shows a stored procedure for inserting new employees into the emp table.

**Listing 5.** The ORG stored procedure starts the recursive tree traversal.

```
store ins_emp
procedure ins_emp static
parameters
  number p_parent_emp_id
  number p_emp_id
  string p_name
  number p_salary
local variables
  sql handle hsq1
  number ind
  number parent_lft
  number parent_rgt
actions
  on procedure startup
    call sqlconnect(hsq1)
  on procedure execute
    call sqlprepareandexecute(hsq1, \
'select lft, rgt from emp where emp_id = \
```

```
:p_parent_emp_id into :parent_lft, :parent_rgt')
call sqlfetchnext(hsq1, ind)
call sqlprepareandexecute(hsq1, \
'update emp set rgt = rgt + 2
where rgt > :parent_lft')
call sqlprepareandexecute(hsq1, \
'update emp set lft = lft + 2
where lft > :parent_lft')
call sqlprepareandexecute(hsq1, \
'insert into emp (emp_id, name, salary, lft, rgt) \
values (:p_emp_id, :p_name, :p_salary, 0, 0)')
call sqlprepareandexecute(hsq1, \
'update emp set lft = :parent_lft + 1, rgt = \
:parent_lft + 2 where emp_id = :p_emp_id')
on procedure close
  call sqldisconnect(hsq1)
/
```

First, ins\_emp finds the lft and rgt values of the parent node. These values are needed in order to discover what other lft and rgt values will be affected by widening the tree. Then the lft and rgt values of affected rows are widened by adding two. Next, the new employee row is inserted into the table with zero values for lft and rgt. And finally, the inserted row is updated with the correct lft and rgt values. I find it a little difficult to explain why it works. What might work for you to understand it is to draw a “before” picture of the table, and then walk through the logic of the stored procedure to see how it arrives at the “after” picture.

An important point about this particular implementation is that new nodes at the same level are always added to the “left” of the parent node. If you're trying to establish a particular order of nodes at the same level with the same parent, then add the least important nodes first.

### Deleting nodes from the tree

When a node is deleted from the tree, it's important to decide how to preserve the integrity of the tree if the node has any subordinate nodes. There are two approaches. The first approach is to promote one of the deleted node's subordinate nodes to take over. The remaining subordinate nodes of the deleted node become subordinate to the promoted node. The other approach is to promote the entire subtree of the deleted node so that all of the nodes subordinate to the deleted node are modified to be subordinate to the deleted node's parent node.

The stored procedure in [Listing 6](#) can handle either approach. You can specify the latter approach of promoting the entire subtree by passing 1 or TRUE in the second parameter. If you pass 0 or FALSE in the second parameter, then the “leftmost” child node is promoted to replace the deleted node.

**Listing 6.** Two ways to delete employee nodes from the tree.

```
store del_emp
procedure del_emp static
parameters
  number del_emp_id
  boolean promo_subtree
```

```

local variables
  sql handle hsq1
  number ind
  number del_lft
  number del_rgt
actions
  on procedure startup
    call sqlconnect(hsq1)
  on procedure execute
    call sqlprepareandexecute(hsq1, \
'select lft, rgt from emp where emp_id = \
:del_emp_id into :del_lft, :del_rgt')
    call sqlfetchnext(hsq1, ind)
    if promo_subtree
      call sqlprepareandexecute(hsq1, \
'delete from emp where emp_id = :del_emp_id')
      call sqlprepareandexecute(hsq1, \
'update emp set lft = lft - 1, rgt = rgt - 1 \
where lft between :del_lft and :del_rgt')
      call sqlprepareandexecute(hsq1, \
'update emp set rgt = rgt - 2 \
where rgt > :del_rgt')
      call sqlprepareandexecute(hsq1, \
'update emp set lft = lft - 2 \
where lft > :del_rgt' )
    else
      call sqlprepareandexecute(hsq1, \
'update emp set lft = lft - 1, rgt = :del_rgt \
where lft = :del_lft + 1')
      call sqlprepareandexecute(hsq1, \
'update emp set rgt = rgt - 2 \
where rgt > :del_lft')
      call sqlprepareandexecute(hsq1, \
'update emp set lft = lft - 2 \
where lft > :del_lft')
      call sqlprepareandexecute(hsq1, \
'delete from emp where emp_id = :del_emp_id')
    on procedure close
      call sqldisconnect(hsq1)
/

```

Again, it's easiest to understand if you "draw" the before and after states of the table and follow the steps of the procedure to see how it gets there.

## Deleting a subtree

Sometimes you want to delete an entire subtree (a node and all the subordinate nodes, immediate and beyond). The stored procedure in [Listing 7](#) provides this capability.

**Listing 7.** The stored procedure for deleting an entire subtree.

```

store del_emporg
procedure del_emporg static
parameters
  number del_emp_id
local variables
  sql handle hsq1
  number ind
  number del_lft
  number del_rgt
  number delta
actions
  on procedure startup
    call sqlconnect(hsq1)
    call sqlsetisolationlevel(hsq1,'r1')
  on procedure execute
    call sqlprepareandexecute(hsq1, \
'select lft, rgt from emp where emp_id = \
:del_emp_id into :del_lft, :del_rgt')
    call sqlfetchnext(hsq1, ind)
    call sqlprepareandexecute(hsq1, \
'delete from emp where lft between :del_lft \
and :del_rgt')

```

```

set delta = del_rgt - del_lft + 1
call sqlprepareandexecute(hsq1, \
'update emp set lft = lft - :delta \
where lft > :del_lft')
call sqlprepareandexecute(hsq1, \
'update emp set rgt = rgt - :delta \
where rgt > :del_rgt')
on procedure close
  call sqldisconnect(hsq1)
/

```

Changing the isolation level in the procedure startup is a fix for getting consistent, reliable results. Without it, it seems the root node's rgt value wasn't always getting updated correctly.

This code works a little faster than deleting node by node. It deletes all the nodes in a particular subtree and then "closes the gap" by narrowing the tree.

## Checking the integrity of the tree

While the insertion and deletion procedures should work consistently and reliably, it never hurts to have some tools to verify the integrity of the tree. The code presented in [Listing 8](#) verifies the tree's integrity, first by checking that there are no duplicates in the lft and rgt values, and then checking that there are no gaps in the lft and rgt values. The queries depend on a view of the lft and rgt values. Fortunately, SQLBase supports unions in views.

**Listing 8.** A view and queries to verify the integrity of a nested set tree.

```

create view orglfttgt (num)
as select lft from emp
union all
select rgt from emp
/

select 'duplicate ' || @string(num,0)
  from orglfttgt
  group by num
  having count(*) > 1
/

select 'gap before ' || @string(o2.num, 0)
  from orglfttgt o1, orglfttgt o2
  where o1.num <= o2.num
  group by 1
  having count(*) <> max(o1.num)'
/

```

If there are any integrity problems in the tree, the procedure will return messages like "duplicate 8" or "gap before 10".

## Harnessing stored procedures

This concludes this mini-series on traversing trees in SQL. I hope you have also picked up on the power of SQLBase stored procedures and have a better perspective on why some logic is better on the server than on the client. **CP**

R.J. David Burke is a Senior Consulting Engineer with Centura Software Corp. He can be contacted through the editorial staff of *Centura Pro*.