Celko

October 20, 2000

Email Article
Print Article

Trees in SQL

Some answers to some common questions about SQL trees and hierarchies.

By Joe Celko

Continued from Page 1

If that mental model does not work, then imagine a little worm crawling counter-clockwise along the tree. Every time it gets to the left or right side of a node, the worm numbers it. The worm stops when it gets all the way around the tree and back to the top.

This model is a natural way to show a parts explosion, because a final assembly is made of physically nested assemblies that break down into separate parts.

At this point, the boss column is both redundant and denormalized, so it can be dropped. Also, note that the tree structure can be kept in one table and all the information about a node can be put in a second table, and you can join both on employee number for queries.

To convert the graph into a nested sets model think of a little worm crawling along the tree. The worm starts at the top -- the root -- and makes a complete trip around the tree. When it comes to a node, it puts a number in the cell on the side that it is visiting and increments its counter. Each node will get two numbers -- one for the right side and one for the left. (Computer Science majors will recognize this as a modified preorder tree traversal algorithm.) Finally, drop the unneeded "Personnel.boss" column, which represents the edges of a graph.

This has some predictable results that we can use for building queries. The root is always of the form (left = 1, right = 2 * (`SELECT COUNT(*) FROM` TreeTable)); leaf nodes always have (left + 1 = right); the `BETWEEN` predicate defines the subtrees; and so on. Here are some common queries that you can use to build others:

1. Find an employee and all his/her supervisors, no matter how deep the tree.

Other Resources
IE Imperatives
Archives
Realware Awards
Career Center
Research Library
Event Calendar
Editorial Calendar
Advanced Search
Translate


```
SELECT P2.*
  FROM Personnel AS P1, Personnel AS P2
 WHERE P1.lft BETWEEN P2.lft AND P2.rgt
   AND P1.emp = :myemployee;
```

2. Find the employee and all his/her subordinates. (This query has a nice symmetry with the first query.)

```
SELECT P1.*
  FROM Personnel AS P1, Personnel AS P2
 WHERE P1.lft BETWEEN P2.lft AND P2.rgt
   AND P2.emp = :myemployee;
```

3. Add a GROUP BY and aggregate functions to these basic queries and you have hierarchical reports. For example, the total salaries that each employee controls:

```
SELECT P2.emp, SUM(S1.salary)
  FROM Personnel AS P1, Personnel AS P2,
       Salaries AS S1
 WHERE P1.lft BETWEEN P2.lft AND P2.rgt
   AND P1.emp = S1.emp
 GROUP BY P2.emp;
```

In the adjacency list method, this has to be done with a cursor.

4. Find the level of each node, so you can print the tree as an indented listing.

```
SELECT COUNT(P2.emp) AS indentation, P1.emp
  FROM Personnel AS P1, Personnel AS P2
 WHERE P1.lft BETWEEN P2.lft AND P2.rgt
 GROUP BY P1.emp
 ORDER BY P1.lft;
```

5. The nested set model has an implied ordering of siblings that the adjacency list model does not. To insert a new node as the rightmost sibling:

```
BEGIN
DECLARE right_most_sibling INTEGER;

SET right_most_sibling
    = (SELECT rgt
         FROM Personnel
        WHERE emp = :your_boss);

UPDATE Personnel
   SET lft = CASE WHEN lft > right_most_sibling
```

```
                  THEN lft + 2
                  ELSE lft END,
       rgt = CASE WHEN rgt >= right_most_sibling
                  THEN rgt + 2
                  ELSE rgt END
 WHERE rgt >= right_most_sibling;

INSERT INTO Personnel (emp, lft, rgt)
VALUES ('New Guy', right_most_sibling,
                  (right_most_sibling + 1))
END;
```

6. To convert an adjacency list model into a nested set model, use a push down stack algorithm. Assume that we have these tables:

```
-- Tree holds the adjacency model

CREATE TABLE Tree
(emp CHAR(10) NOT NULL,
 boss CHAR(10));

INSERT INTO Tree
SELECT emp, boss FROM Personnel;

-- Stack starts empty, will holds the nested set model
CREATE TABLE Stack
(stack_top INTEGER NOT NULL,
 emp CHAR(10) NOT NULL,
 lft INTEGER,
 rgt INTEGER);

BEGIN ATOMIC
DECLARE counter INTEGER;
DECLARE max_counter INTEGER;
DECLARE current_top INTEGER;

SET counter = 2;
SET max_counter = 2 * (SELECT COUNT(*) FROM Tree);
SET current_top = 1;

INSERT INTO Stack
SELECT 1, emp, 1, NULL
  FROM Tree
 WHERE boss IS NULL;

DELETE FROM Tree
 WHERE boss IS NULL;
```

```
WHILE counter <= (max_counter - 2)
LOOP IF EXISTS (SELECT *
                  FROM Stack AS S1, Tree AS T1
                 WHERE S1.emp = T1.boss
                   AND S1.stack_top = current_top)
     THEN
     BEGIN -- push when top has subordinates, set lft value
       INSERT INTO Stack
       SELECT (current_top + 1), MIN(T1.emp), counter, NULL
         FROM Stack AS S1, Tree AS T1
        WHERE S1.emp = T1.boss
          AND S1.stack_top = current_top;

        DELETE FROM Tree
         WHERE emp = (SELECT emp
                        FROM Stack
                       WHERE stack_top = current_top + 1);

        SET counter = counter + 1;
        SET current_top = current_top + 1;
     END
     ELSE
     BEGIN  -- pop the stack and set rgt value
       UPDATE Stack
          SET rgt = counter,
              stack_top = -stack_top -- pops the stack
        WHERE stack_top = current_top
       SET counter = counter + 1;
       SET current_top = current_top - 1;
     END IF;
 END LOOP;
END;
```

Although this procedure works, you might want to use a language that has arrays in it, instead of trying to stick to pure SQL.

Joe Celko is an Atlanta-based independent consultant. He is the author of *Instant SQL Programming* (Wrox Press, 1997). You can contact him at www.celko.com or 71062.1056@compuserve.com.