DBMS, May 1996

DBMS online

SQL for Smarties

Joe Celko

Nontraditional Databases

Joe takes care of some business and continues his discussion of trees in SQL.

In February, I spent five days at an introductory class to Pilot Software's LightShip, a multi-dimensional database (MDDB) product. The first three days dealt with the server, which I wanted to learn; the last two days dealt with Pilot•s GUI front-end builder, which I will never use because I bill at too high a rate. Right now, the big debate in the OLAP world is MDDB vs. relational. Granted, I am a bit predisposed toward the relational products. However, I was seriously underwhelmed by LightShip, and said so on CompuServe in the Information Management forum. I found the server language to be from the old punch card, Fortran, and Basic schools of language design. For instance, the language has one statement per line (similar to punch cards), each command is unique in the first three letters (similar to Basic), and it uses two-letter comparison operator symbols (such as Fortran's "GT" for "greater than"). You perform a "query" in a series of steps that fetch each part of the data into a working matrix.

On CompuServe, Neil Raden sent me an email in which he remembered when John Kopke, then of EPS Consultants (which is now defunct), was developing the product in the early 1980s. Jim Dorrian and Bob Earle, the founders of Arbor Software Corp., were also working there at that time, along with Clark French, inventor of Sybase's IQ indexing product. Raden felt it was an orphan of dubious heritage and should be put out to pasture.

On the other hand, Alan Eager emailed back a reply in which he said that "... I would challenge your statements about being able to do the dimensional and time series analysis in SQL to the same level of sophistication." Well, I think I can. I would use the nested sets model for the dimensional hierarchies and some tricks for crosstabs (which I will discuss in another column). But the problem is that a SQL code generator could build these queries for me. The generated code will probably not optimize as well as

my handwritten stuff, but it will get written a lot quicker. Generated code is also easier to read and maintain because you maintain the higher-level language, not the actual SQL. However, there is still the problem of performance for large, complicated SQL queries against large databases.

I got an unexpected email from Dan Holle of White Cross Systems, a British company that develops a massively parallel computer that uses SQL as its "native language" -- much the same way that the old Burroughs 5000 family used Algol. White Cross' approach is to have strong hardware and a good optimizer, to let you scan a million rows of data in a reasonable time. When you are looking at paying between $100,000 and $250,000 for an MDDB product, buying specialized hardware (which is still more flexible than custom software) is not such a bad idea. Plus, it's a simple way around the performance problem.

A White Cross machine at British Telecom (BT) has a model of BT's business customers set up for OLAP queries. The company is keeping 110 dimensions in the model; the OLAP people were thinking in terms of less than 10 dimensions. Even if the SQL stinks, a massively parallel machine can return million-row joins in less than 10 seconds with simple brute force.

Informix

Also in February, I attended a globally televised Informix presentation at a downtown Atlanta hotel. It was showing off its efforts to integrate Illustra, Michael Stonebreaker's object-relational database, with Informix's Dynamic Scalable Architecture. The idea behind Illustra is that users can define their own storage and tool objects (called "DataBlades" in Illustra jargon -- taken from the Swiss Army knife) for nontraditional data, and immediately plug them into their existing Illustra databases. People do not define new data types every day, so you can buy predefined DataBlades for several of the more common nontraditional data types.

By the time you read this column, Rick Smolan will have finished his "24 Hours in Cyberspace: Painting on the Walls of the Digital Cave" project, and you will know the results. If you do not recognize Smolan's name, you might remember his "A Day in the Life ..." photography books. On February 8, 1996, more than a thousand photographers will have photographed computer users all over the planet and sent their work to a Web site in San Francisco (http://www.cyber24.com), where they will be edited and posted in real time on an Informix database that uses the Illustra technology. Look for the coffee table book.

Trees in SQL -- Part III

Let's continue our discussion of the nested set model for trees in SQL. I am not going to review any of my prior columns on the topic and will assume that you still have a copy of the Personnel table I was using for the examples (DBMS, March 1996, page 24). If you don't have the back issues, you can make my publisher happy by buying some.

I have also been asked why I don't show very much procedural code in the examples. Right now, ANSI and ISO are trying to agree on a standard procedural language for triggers and stored procedures called the SQL/PSM (Persistent Stored Module). However, this standard has not passed yet, which means that I would have to use either a pseudo-code of my own or pick one vendor's proprietary 4GL. I decided to use English commentary for now, but I will start using the SQL/PSM when it is finalized.

The real tricky part of handling trees in SQL is finding a way to convert the adjacency matrix model into the nested set model within the framework of pure SQL. It would be fairly easy to load an adjacency matrix model table into a host language program, and then use a recursive preorder tree traversal program from a college freshman data structures textbook to build the nested set model.

To be honest, tree traversal might also be faster than what I am about to show you. But I want to do it in pure SQL to prove a point; you can do anything in a declarative language that you can do in a procedural language. Because this is a teaching exercise, I will explain things in painful detail.

A classic problem-solving approach is to take the simplest statement of the problem, and see if you can apply it to the more complex cases. If the tree has zero nodes, then the conversion is easy -- don't do anything. If the tree has one node, then the

conversion is easy -- set the left value to 1 and the right value to 2. The nature of the adjacency matrix is that you can travel only one level at a time, so let's look at a tree with two levels -- a root and some immediate children. The adjacency model table would look like the following:

```
CREATE TABLE Personnel
(emp CHAR (10) NOT NULL PRIMARY KEY,
boss CHAR (10));
```

```
Personnel

emp        boss
================
'Albert'   NULL
'Bert'    'Albert'
'Charles' 'Albert'
'Diane'   'Albert'
```

Let's put the nested set model into a working table of its own:

```
CREATE TABLE WorkingTree (emp CHAR (10),
boss CHAR (10),
lft INTEGER NOT NULL DEFAULT 0,
rgt INTEGER NOT NULL DEFAULT 0);
```

From the previous columns in this series, you know that the root of this tree is going to have a left value of 1, and that the right value is twice the number of nodes in the tree. However, I am going to introduce a convention in the working table; namely, that the boss column will always contain the key value of the root of the original tree. In effect, this will be the name of the nested set:

```
INSERT INTO WorkingTree
--convert the root node
SELECT P0.boss, P0.boss, 1,
2 * (SELECT COUNT(*) + 1
FROM Personnel AS P1
WHERE P0.boss = P1.boss)
FROM Personnel AS P0;
```

Now, you need to add the children to the nested set table. The original boss will stay the same. The ordering of the children is the natural ordering of the data type used to represent the key; in this case, emp char(10):

```
INSERT INTO WorkingTree
--convert the children
SELECT DISTINCT P0.emp, P0.boss,
2 * (SELECT COUNT(DISTINCT emp)
FROM Personnel AS P1
WHERE P1.emp < P0.emp
AND P0.boss IN (P1.emp, P1.boss)),
2 * (SELECT COUNT(DISTINCT emp)
FROM Personnel AS P1
WHERE P1.emp < P0.emp
AND P0.boss IN (P1.boss, P1.emp)) + 1
FROM Personnel AS P0;
```

In fact, you can use this procedure to convert an adjacency matrix model into a forest of trees, each of which is a nested set model identified by its root (boss) value. Thus, the Albert family tree is the set of rows that have Albert as the boss, the Bert family tree is the set of rows that have Bert as the boss, and so on. (This concept is illustrated in Figures 1 and 2.)

Because the original adjacency matrix table repeated the non-leaf, non-root nodes in both the emp and boss columns, the WorkingTree table will duplicate nodes as a root in one tree and as a child in another. The query will also behave strangely with the null value in the boss column of the original adjacency matrix table, so you will need to clean up the WorkingTree table with the following statement:

```
DELETE FROM WorkingTree
WHERE boss IS NULL OR emp IS NULL;
```

To get these trees to merge into one final tree, you need a way to attach a subordinate tree to its superior. In a procedural language, you could accomplish this with a program that would take the following steps:

1. Find the size of the subordinate tree.

2. Find where the subordinate tree inserts into the superior tree.

3. Stretch the superior tree at the insertion point.

4. Insert the subordinate tree into the insertion point.

In a nonprocedural language, you would perform these steps all at once by using logic on all of the rows involved. You begin this process by asking questions and noting facts:

Q) How do you pick out a superior and its subordinate tree in the forest?

A) Look for a single key value that is a child in the superior tree and the root of the subordinate tree.

Q) How do you tell how far to stretch the superior tree?

A) It has to be the size of the subordinate tree, which is (2 * (select count(*) from Subordinate)).

Q) How do you locate the insertion point?

A) It is the row in the superior table where the emp value is equal to the boss value in the subordinate table. You want to put the subordinate tree just to the left of the left value of this common node. A little algebra gives you the amount to add to all the left and right values to the right of the insertion point.

The easiest way to explain this is with the decision table shown in Table 1.

Look at the conditions that will cause errors. A row cannot be in both the superior and the subordinate tree (rule 6). This condition is already handled by the way you constructed the original and working tree tables.

If a row is in the superior table, it cannot have a left value to the right of the insertion point whose right is not also the right of the insertion point -- that is because (left < right) for all rows (rule 3).

When you update a node, you change the boss in the subordinate to the head of the new family into which its tree has been assimilated. That is the easy part!

The rules for the lft and rgt columns are harder, but there are only a few options: 1) you leave the left and right values alone; 2) you add the size of the subordinate to the left, the right, or both; and 3) you add the displacement necessary to get the row into the opening in the superior table to the left, the right, or both.

You are now ready to write the following procedure, which will merge two trees:

```
CREATE PROCEDURE TreeMerge
   (superior NOT NULL, subordinate NOT NULL)

BEGIN

DECLARE size INTEGER NOT NULL;
DECLARE insert_point INTEGER NOT NULL;

SET size = 2 * (SELECT COUNT(*)
                        FROM WorkingTree
                        WHERE emp = subordinate);

SET insert_point = (SELECT MIN(lft)
                                FROM WorkingTree
                               WHERE emp = subordinate
                               AND boss = superior) - 1;

UPDATE WorkingTree
   SET boss = CASE WHEN boss = subordinate
                                THEN CASE WHEN emp = subordinate
                                                    THEN NULL
                                                    ELSE superior END
                        ELSE boss END,

lft = CASE WHEN (boss = superior AND lft > size)
                  THEN lft + size
                  ELSE CASE WHEN boss = subordinate
                                                    AND emp <> subordinate
                                  THEN lft + insert_point
                                  ELSE lft END
       END,

 rgt = CASE WHEN (boss = superior AND rgt > size)
                     THEN rgt + size
                     ELSE CASE WHEN boss = subordinate
                                                       AND emp <> subordinate
                                    THEN rgt + insert_point
                                    ELSE rgt END
          END

 WHERE boss IN (superior, subordinate);

 -- delete the redundant copies of the subordinate tree root

 DELETE FROM WorkingTree
    WHERE boss IS NULL OR emp IS NULL;
```

```
END;
```

Finding pairs of superiors and subordinates in the WorkingTree table is easy with a view. The following view becomes empty when all the bosses are set to the same value:

```
CREATE VIEW AllPairs (superior, subordinate)
AS SELECT W1.boss, W1.emp
FROM WorkingTree AS W1
WHERE EXISTS (SELECT *
FROM WorkingTree AS W2
WHERE W2.boss = W1.emp)
AND W1.boss <> W1.emp;
```

But you would really like to get just one pair, which you will pass to the procedure you just designed. To pull one pair, say the left-most pair, from the view, use the following query:

```
CREATE VIEW LeftmostPairs (superior, subordinate) AS SELECT DISTINCT superior,
(SELECT MIN (subordinate)
FROM AllPairs AS A2
WHERE A1.superior = A2.superior)
FROM AllPairs AS A1
WHERE superior = (SELECT MIN(superior) FROM AllPairs);
```

Now all you have to do is fold this query into the original procedure, and you have a routine that will merge the forest of trees together from left to right. Use a while loop controlled by the existence of values in the LeftmostPairs view to drive the calls to the procedure. This is the only procedural control structure in the entire stored procedure.

Clearly, this procedure works better for flat, wide trees than for tall, thin trees with the same number of nodes. I have not performed a formal analysis of the algorithm, but it is related to the formula ([number of nodes] - [1 root] - [number of leaf nodes]), which gives the number of "trees in the forest" after the first conversion step.

Puzzle

This finishes the series on trees and hierarchies. For a puzzle this month, you are to use your database's procedural language to implement the routines we just discussed and to submit the code and any test results to me via CompuServe (see the address in my bio). I will mention the best solutions for each product in an upcoming column, and the winner will receive a book as a prize. The deadline for the best answer is June 15, 1996.

For extra points, you can submit a recursive procedure (assuming that the 4GL in your SQL product supports recursion) that performs a tree traversal and a comparison of the performance of the two approaches.

The real tricky part of handling trees in SQL is finding a way to convert the adjacency matrix model into the nested set model within the framework of pure SQL.

Table 1. Sample decision table

```
C1 |  row in superior    | y | y | y | y | n | y | n |
C2 |  row in subord      | n | n | n | n | y | y | n |
```

```
C3 |  lft > cut          | n | n | y | y | - | - | - |
C4 |  rgt > cut          | n | y | n | y | - | - | - |

=====================================================

A1 |  Error              |   |   | 1 |   |   | 1 |   |
A2 |  lft := lft + size  |   |   |   | 1 |   |   |   |
A3 |  rgt := rgt + size  |   | 1 |   | 2 |   |   |   |
A4 |  lft := lft         | 1 | 2 |   |   |   |   | 1 |
A5 |  rgt := rgt         | 2 |   |   |   |   |   | 2 |
A6 |  lft := lft + cut   |   |   |   |   | 1 |   |   |
A7 |  rgt := rgt + cut   |   |   |   |   | 2 |   |   |
```

Joe Celko is a member of the ANSI X3H2 Database Standards Committee, a widely published author, and a consultant with OSoft Development Corp. in Atlanta. Joe is also a frequent contributor to the DBMS Forum on CompuServe. You can email him at 71062.1056@compuserve.com.

Table of Contents - May 1996 | Home Page

Please send questions or comments to mfrank@mfi.com
Updated Tuesday, June 18, 1996