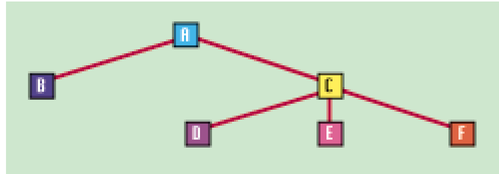
Trees in SQL

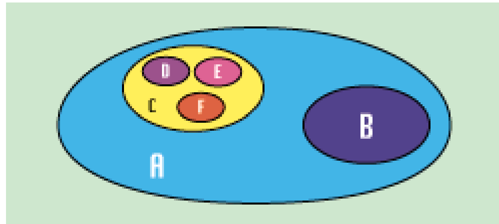
A tree is a special kind of directed graph. Graphs are data structures that are made up of nodes or vertices (usually shown as boxes) connected by arcs or edges (usually shown as lines with arrowheads). Each edge represents a one-way relationship between the two nodes it connects. In an organizational chart, the nodes are personnel, and each edge is the "reports to" relationship. In a parts explosion (also called a bill of materials), the nodes are assembly units (eventually resolving down to individual parts), and each edge is the "is made of" relationship.
The top of the tree is called the root. In an organizational chart, it is the highest authority; in a parts explosion, it is the final assembly. The number of edges coming out of the node are its outdegree, and the number of edges entering it are its ind egree. A binary tree is one in which a parent can have no more than two children; more generally, an n-ary tree is one in which a node can have no more than outdegree n, or any number of child nodes.
The nodes of the tree that have no subtrees beneath them are called the leaf nodes. In a parts explosion, they are the individual parts, which cannot be broken down any further. The descendants, or children, of a parent node are every node at all lower l evels in the subtree that has the parent node as its root.

FIGURE 1



The top of the tree is called the root. The nodes of the tree that have no subtrees beneath them are called the leaf nodes. The descendants, or children, of a parent node are every node at all lower levels in the subtree that has the parent node as its root.

FIGURE 2



Another way of representing trees is to show them as nested sets. Because SQL is a set-oriented language, this is a better model. The root of the tree is a set that contains all the other sets, and the child relationship is shown as set containment.

Trees are often drawn as charts. (See Figure 1.) Americans like to put the root at the top and grow the tree downward; Europeans will often put the root at the bottom and grow the tree upward, or grow the tree from left to right ac ross the page. Another way of representing trees is to show them as nested sets (see Figure 2); this is the basis for the nested set representation in SQL that I use.
In SQL, any relationship must be shown explicitly as data. The typical way to represent trees is to put an adjacency matrix in a table. That is, one column is the parent node, and another column in the same row is the child node (the pair represents an e dge in the graph). For example, consider the organizational chart of this six-person company:

```
CREATE TABLE Personnel(
 emp CHAR(20) PRIMARY KEY,
 boss CHAR(20) REFERENCES Personnel(emp),
 salary DECIMAL(6,2) NOT NULL
);
```

Personnel

emp	boss	salary
'Jerry'	NULL	1000.00
'Bert'	'Jerry'	900.00
'Chuck'	'Jerry'	900.00
'Donna'	'Chuck'	800.00
'Eddie'	'Chuck'	700.00
'Fred'	'Chuck'	600.00

This model has advantages and disadvantages. The PRIMARY KEY is clearly emp, but the boss column is functionally dependent upon it, so you have normalization problems. The REFERENCES constraint will keep you from giving someone a boss who is not also an employee.
However, what happens when 'Jerry' changes his name to 'Geraldo' to get a television talk show? You have to cascade the change to the 'Bert' and 'Chuck' rows as well.

Another disadvantage of the adjacency model is that path enumeration is difficult. To find the name of the boss for each employee, the query is a self-join, such as:

```
SELECT B1.emp, 'bosses', E1.emp
FROM Personnel AS B1, Personnel AS E1
WHERE B1.emp = E1.boss;
```

But something is missing here. This query gives you only the immediate bosses of the personnel. Your boss's boss also has authority over you, and so on up the tree until you find someone who has no subordinates. To go two levels deep in the tree, you nee d to write a more complex self-join, such as:

```
SELECT B1.emp, 'bosses', E2.emp
FROM Personnel AS B1, Personnel AS E1, Personnel AS E2
WHERE B1.emp = E1.boss AND E1.emp = E2.boss;
```

To go more than two levels deep in the tree, simply extend the pattern:

```
SELECT B1.emp, 'bosses', E3.emp
FROM Personnel AS B1,
 Personnel AS E1,
 Personnel AS E2,
 Personnel AS E3
WHERE B1.emp = E1.boss AND E1.emp = E2.boss AND E2.emp = E3.boss;
```

Unfortunately, you have no idea just how deep the tree is, so you must keep extending this query until you get an empty set back as a result.

A leaf node has no children under it. In an adjacency model, this set of nodes is fairly easy to find: It is the personnel who are not bosses to anyone else in the company:

```
SELECT *
FROM Personnel AS E1
WHERE NOT EXISTS (
  SELECT *
  FROM Personnel AS E2
  WHERE E1.emp = E2.boss
);
```

The root of the tree has a boss that is null:

```
SELECT * FROM Personnel WHERE boss IS NULL;
```

The real problems are in trying to compute values up and down the tree. As an exercise, write a query to sum the salaries of each employee and his/her subordinates; the result is:

Personnel

emp	boss	salary
'Jerry'	NULL	4900.00
'Bert'	'Jerry'	900.00
'Chuck'	'Jerry'	3000.00
'Donna'	'Chuck'	800.00
'Eddie'	'Chuck'	700.00
'Fred'	'Chuck'	600.00

Nested-Set Model of Trees

Another way of representing trees is to show them as nested sets. Because SQL is a set-oriented language, this is a better model. The root of the tree is a set that contains all the other sets, and the child relationship is shown as set containment.
There are several ways to think about transforming the organizational chart into nested sets. One way is to imagine that you are pulling the subordinate ovals inside their parents using the edge lines as ropes. The root is the largest oval and contains e very other node. The leaf nodes are the innermost ovals, with nothing else inside them, and the nesting shows the hierarchical relationship. This is a natural way to model a parts explosion, because a final assembly is made of physically nested assemblie s that finally break down into separate parts.
Another approach is to visualize a little worm crawling along the "boxes and arrows" of the tree. The worm starts at the top, the root, and makes a complete trip around the tree. Computer science majors will recognize this as a modified preorder tree-tra versal algorithm.
But now let's get a smarter worm with a counter that starts at one. When the worm comes to a node box, 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 side.
This has some predictable results that you can use for building queries. The Personnel table will look like the following, with the left and right numbers in place:

```
CREATE TABLE Personnel (
 emp CHAR(10) PRIMARY KEY,
 salary DECIMAL(6,2) NOT NULL,
 left INTEGER NOT NULL,
 right INTEGER NOT NULL
);
```

Personnel

emp	salary	left	right
'Jerry'	1000.00	1	12
'Bert'	900.00	2	3
'Chuck'	900.00	4	11
'Donna'	800.00	5	6
'Eddie'	700.00	7	8
'Fred'	600.00	9	10

The root will always have a 1 in its left-visit column and twice the number of nodes `(2*n)` in its right-visit column. This is easy to understand: The worm has to visit each node twice, once for the left side and once for the right side, so the final coun t has to be twice the number of nodes in the entire tree.
In the nested-set model, the difference between the left and right values of leaf nodes is always 1. Think of the little worm turning the corner as it crawls along the tree. Therefore, you can find all leaf nodes with the following simple query:

```
SELECT * FROM Personnel WHERE (right - left) = 1;
```

You can use another trick to speed up queries. Build a unique index on the left column, then rewrite the query to take advantage of the index:

```
SELECT * FROM Personnel WHERE left = (right - 1);
```

The reason this improves performance is that the SQL engine can use the index on the left column when it does not appear in an expression. Don't use (right - left) = 1, because it will not take advantage of the index.
In the nested-set model, paths are shown as nested sets, which are represented by the nested sets' numbers and BETWEEN predicates. For example, to find out all of the bosses to whom a particular person reports in the company hierarchy, you would write:

```
SELECT :myworker, B1.emp, (right - left) AS height
```

```
FROM Personnel AS B1, Personnel AS E1
WHERE E1.left BETWEEN B1.left AND B1.right
 AND E1.right BETWEEN B1.left AND B1.right
 AND E1.emp = :myworker;
```

The greater the height, the farther up the corporate hierarchy that boss is from the employee. The nested-set model uses the fact that each containing set is larger in size (where size = (right - left)) than the sets it contains. Obviously, the root will always have the largest size.
The level function is the number of edges between two given nodes, and it is fairly easy to calculate. For example, to find the levels of bureaucracy between a particular worker and manager, you would use:

```
SELECT E1.emp, B1.emp, COUNT(*) - 1 AS levels
FROM Personnel AS B1, Personnel AS E1
WHERE E1.left BETWEEN B1.left AND B1.right
 AND E1.right BETWEEN B1.left AND B1.right
 AND E1.node = :myworker
 AND B1.node = :mymanager;
```

The reason for using the expression (COUNT(*) - 1) is to remove the duplicate count of the node itself as being on another level, because a node is zero levels removed from itself.
You can build other queries from this basic template using views to hold the path sets. For example, to find the common bosses of two employees, union their path views and find the nodes that have (COUNT(*)>1). To find the nearest common ancestors of two nodes, UNION the path views, find the nodes that have (COUNT(*)>1), and pick the one with the smallest depth number.
I will get into more programming tricks with this model next month, but for now, try to write the sum of all subordinate salaries with this table and compare it to what you did for the edge model version of this hierarchy.

Puzzle

I am not going to give you a problem with trees this month -- I will wait until I get further into the topic and the problems are harder. Instead, suppose you have a table with the addresses of consumers to whom you wish to send junk mail. The table has a fam (family) column that links Consumers with the same stree t address. You need this because one of your rules is that you can mail only one offer to a household. The field contains the primary-key value of the Consumers record of the first person who has this address, thus:

Consumers

name	address	id	fam
Bob	A	1	NULL
Joe	B	3	NULL
Mark	C	5	NULL
Mary	A	2	1
Vickie	B	4	3
Wayne	D	6	NULL

You need to delete those rows in which fam is null, but there are other family members on the mailing list. In the above example, you need to delete Bob and Joe, but not Mark or Wayne.

Puzzle Answer

During your first attempt, you might try to do too much work. For instance, translating the English specification directly into SQL gives you the following:

```
DELETE FROM Consumers
WHERE fam IS NULL    - this guy has a NULL family value
 AND EXISTS (   - . .and there is someone who is
   SELECT *
   FROM Consumers AS C1
   WHERE C1.id  id        - a different person
    AND C1.address = address   - at the same address
    AND C1.fam IS NOT NULL - who has a family value
 );
```

But if you think about it, you will see that the `count(*)` for the household must be greater than one.

```
DELETE FROM Consumers
WHERE fam IS NULL   - this guy has a NULL family value
 AND (
   SELECT COUNT(*)
   FROM Consumers AS C1
   WHERE C1.address = address
 ) > 1;
```

The trick is that the `count(*)` aggregate will include NULLs in its tally.--*Joe Celko*

Nested Set Model of Trees, Part 2

LISTING 1

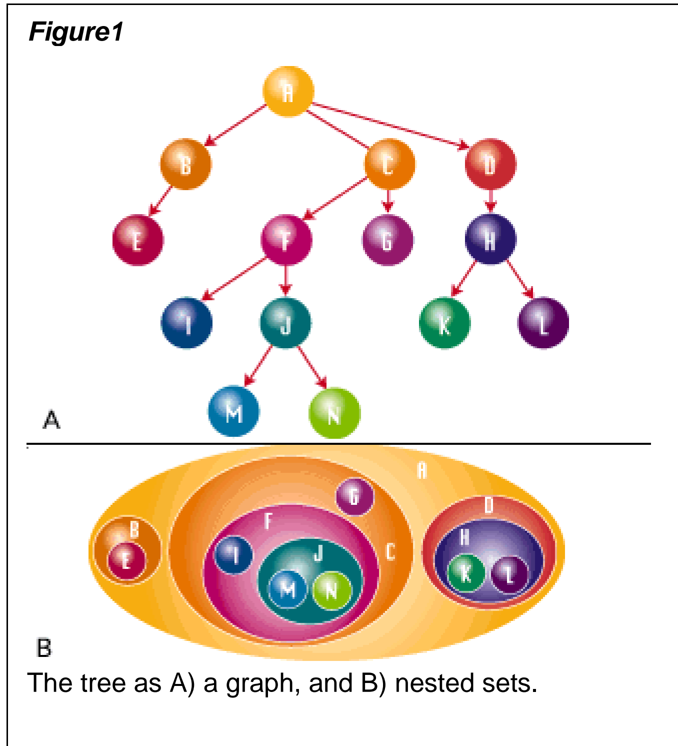
```
CREATE TABLE Personnel (
 emp CHAR (10) PRIMARY KEY,
 salary DECIMAL (8,2) NOT NULL
   CHECK (salary >= 0.00),
 lft INTEGER NOT NULL,
 rgt INTEGER NOT NULL,
 CHECK (lft < rgt)
);
INSERT INTO Personnel VALUES ('Albert', 1000.00, 1, 28);
INSERT INTO Personnel VALUES ('Bert', 900.00, 2, 5);
INSERT INTO Personnel VALUES ('Charles', 900.00, 6, 19);
INSERT INTO Personnel VALUES ('Diane', 900.00, 20, 27);
INSERT INTO Personnel VALUES ('Edward', 750.00, 3, 4);
INSERT INTO Personnel VALUES ('Fred', 800.00, 7, 16);
INSERT INTO Personnel VALUES ('George', 750.00, 17, 18);
INSERT INTO Personnel VALUES ('Heidi', 800.00, 21, 26);
INSERT INTO Personnel VALUES ('Igor', 500.00, 8, 9);
INSERT INTO Personnel VALUES ('Jim', 100.00, 10, 15);
INSERT INTO Personnel VALUES ('Kathy', 100.00, 22, 23);
INSERT INTO Personnel VALUES ('Larry', 100.00, 24, 25);
INSERT INTO Personnel VALUES ('Mary', 100.00, 11, 12);
INSERT INTO Personnel VALUES ('Ned', 100.00, 13, 14);
```

I am going to assume that you have my March 1996 column in front of you, so I do not have to review. The nested set model of trees actually has some properties that I did not mention last month. But first, let's build a table (see Listing 1) to hold personnel information. I will refer to this table throughout the rest of this discussion.

The tree in *Figure 1* is represented as A) a graph and B) nested sets. Therefore, it can be navigated in only one direction along its edge. The direction is shown by the nesting; that is, you know that someone is the subordinate of someone else in the company hierarchy because that person's left and right set numbers are between those of their bosses. The same thing is true of the adjacency

matrix model, but it represents the direction of the edge with columns that have the start a nd finish nodes.

Another property I did not mention last time is that the children are ordered; that is, you can use the set numbers to order the children from left to right. This is not true of the adjacency matrix model, which has no ordering among the children. You sh ould consider this fact when you are inserting, updating, or deleting nodes in the tree.



The tree as A) a graph, and B) nested sets.

A defining property of a tree is that it is a graph without cycles. That is, no path folds back on itself to catch you in an endless loop when you follow it in the tree. Another defining property is that there is always a path from the root to any other node in the tree. In the nested set model, paths are shown as nested sets, which are represented by the nested set's numbers and between predicates. For example, to find out all the managers to whom a particular worker reports in the company hierarchy, y ou would write:

```
SELECT 'Mary', P1.emp, (P1.rgt - P1.lft) AS size
FROM Personnel AS P1, Personnel AS P2
WHERE P2.emp = 'Mary' AND P2.lft BETWEEN P1.lft AND P1.rgt;
```

Mary	emp	size
Mary	Albert	27
Mary	Charles	13
Mary	Fred	9
Mary	Jim	5
Mary	Mary	1

Notice that when the size is equal to one, you are dealing with Mary as her own boss. If you don't allow employees to think for themselves, you may want to exclude this case.

The nested set model uses the fact that each containing set is larger in size (size is defined as right to left) than the sets it contains. Obviously, the root will always have the largest size. JOINS and ORDER BY clauses do not interfere with the nested set model like they do with the adjacency graph model. Plus, the results are not dependent on the order in which the rows are displayed. Users of Oracle tree extensions should be aware of this advantage.

The level of a node is the number of edges between it and a path back to the root. You can compute the level of a given node with the following query:

```
SELECT P2.emp, COUNT(*) AS level
FROM Personnel AS P1, Personnel AS P2
WHERE P2.lft BETWEEN P1.lft AS P2
GROUP BY P2.emp;
```

This query finds the levels of bureaucracy among managers, as follows:

emp	level
Albert	1
Bert	2
Charles	2
Diane	2
Edward	3
Fred	3
George	3
Heidi	3
Igor	4
Jim	4
Kathy	4
Larry	4
Mary	5
Ned	5

In some books on graph theory, you will see the root counted as level zero instead of level one. If you like that convention, use the expression `"(COUNT(*)-1)"` instead.

The self-join and BETWEEN predicate combination is the basic template for other queries. In particular, you can use views based on the template to answer a wide range of questions. In fact, this month's puzzle involves a few of thos e questions.

Aggregate Functions on Trees

Obtaining a simple total of the salaries of manager's subordinates works the same way. Notice that this total will also include the boss's salary:

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

emp	payroll
Albert	7800.00
Bert	1650.00
Charles	3250.00
Diane	1900.00
Edward	750.00
Fred	1600.00
George	750.00
Heidi	1000.00
Igor	500.00
Jim	300.00
Kathy	100.00
Larry	100.00
Mary	100.00
Ned	100.00

Deleting Subtrees

The following query will take the fired (oops -- I mean "downsized") employee as a parameter and remove the subtree rooted under him/her. The trick in this query is that you are using the key, but you need to get the left and right values to do the work. The answer is a set of scalar subqueries:

```
DELETE FROM Personnel
WHERE lft BETWEEN
 (SELECT lft FROM Personnel WHERE emp = :downsized)
AND
 (SELECT rgt FROM Personnel WHERE emp = :downsized);
```

The problem is that this query results in gaps in the sequence of nested set numbers. You can still perform most tree queries on a table with such gaps because the nesting property is preserved. That means you can use the between predicates in your queri es, but other operations that depend on the density property will not work in a tree with gaps. For example, you will not find leaf nodes by using the predicate (right-left=1), nor will you find the number of nodes in a subtree by using the left and righ t values of its root.
Unfortunately, you just lost some information that would be very useful in closing those gaps -- namely the right and left numbers of the subtree's root. Therefore, forget the query and write a procedure instead:

```
CREATE PROCEDURE DropTree (downsized IN CHAR(10) NOT NULL)
BEGIN ATOMIC
DECLARE dropemp CHAR(10) NOT NULL;
DECLARE droplft INTEGER NOT NULL;
DECLARE droprgt INTEGER NOT NULL;
```

Now save the dropped subtree data with a single select:

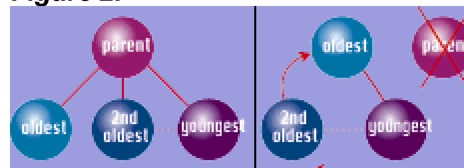
```
SELECT emp, lft, rgt
 INTO dropemp, droplft, droprgt
 FROM Personnel
 WHERE emp = downsized;
```

The deletion is easy:
```
DELETE FROM Personnel WHERE lft BETWEEN droplft and droprgt;
```

Now close up the gap:
```
UPDATE Personnel SET lft = CASE  WHEN lft > droplf
   THEN lft - (droprgt - droplft + 1) ELSE lft END,
rgt = CASE WHEN rgt > droplft
   THEN rgt - (droprgt - droplft + 1) ELSE rgt END;
END;
```

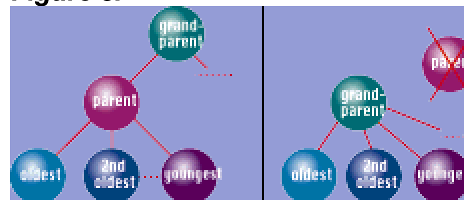
(A real procedure should have some error handling included with it, but I am leaving that as an exercise for the reader.)

Figure 2.



Deleting a single node in the middle of the tree is more difficult than removing entire subtrees. When you remove a node in the middle of the tree, you must decide how to fill the hole. One way to accomplish this is to promote one of the children to th e original node's position (suppose Dad dies and the oldest son takes over the business, as shown here). The oldest child is always shown as the left-most child node under its parent.

Figure 3.



The second method for deleting a single node in the middle of a tree is to connect the children to the parent of the original node (say Mom dies and the kids are adopted by Grandma, as shown here).

<u>Deleting a Single Node</u>

Deleting a single node in the middle of the tree is harder than removing entire subtrees. When you remove a node in the middle of the tree, you have to decide how to fill the hole. There are two ways you can accomplish this. The first method is to promot e one of the children to the original node's position (suppose Dad dies and the oldest son takes over the business, as shown in Figure 2). The oldest child is always shown as the left-most child node under its parent. There is a pr oblem with this operation, however. If the older child has children of his own, you must decide how to handle them, and so on down the tree until you get to a leaf node.

The second method for deleting a single node in the middle of a tree is to connect the children to the parent of the original node (say Mom dies and the kids are adopted by Grandma, as shown in Figure 3). This happens automatically in the nested set model; you simply delete the node and its children are already contained in their ancestor nodes. However, you must be careful when you try to close the gap left by the deletion. There is a difference in renumbering the descendants of the deleted node and renumbering other nodes to the right. Following is a procedure for doing so:

```
CREATE PROCEDURE DropNode (downsized IN CHAR(10) NOT NULL)
BEGIN ATOMIC
DECLARE dropemp CHAR(10) NOT NULL;
DECLARE droplft INTEGER NOT NULL;
DECLARE droprgt INTEGER NOT NULL;
```

Now save the dropped node data with a single select:
```
SELECT emp, lft, rgt
 INTO dropemp, droplft, droprgt
 FROM Personnel
 WHERE emp = downsized;
```

The deletion is easy:
```
DELETE FROM Personnel WHERE emp = downsized;
```

And now close up the gap:
```
UPDATE Personnel SET lft = CASE
 WHEN lft BETWEEN droplft AND droprgt THEN lft - 1
 WHEN lft > droprgt THEN lft - 2 ELSE lft END
rgt = CASE
 WHEN rgt BETWEEN droplft AND droprgt THEN rgt - 1
 WHEN rgt > droprgt THEN rgt -2 ELSE rgt END;
WHERE lft > droplft;
END;
```

I will discuss insertions next month, but you can try it on your own before then.

Puzzle
The puzzle this month is a pop quiz to see if you have been paying attention to this column.
Find all the subordinates of an employee.
Find all the common bosses of two employees (this is a query for "cousins" in a family tree).
All I had in the original declaration of the Personnel table was one constraint (lft<rgt); can you think of more constraints that would help ensure a correct table? (See Puzzle Answer.)

Puzzle Answer

1. This is a matter of "flipping" the original query inside out by reversing the self join in the predicate:

```
SELECT M1.emp, ' manages ', Subordinates.emp
FROM Personnel AS M1, Personnel AS Subordinates
WHERE Subordinates.left BETWEEN M1.left AND M1.right;
```

2. First create a view, which will be useful later:

```
CREATE VIEW ReportsTo (emp, boss)
AS SELECT P2.emp, P1.emp
FROM Personnel AS P1, Personnel AS P2
WHERE P2.lft BETWEEN P1.lft AND P1.rgt;
```

Then, use the following query to obtain the chain of command for one employee, keeping only the bosses he/she has in common with the other employees:

```
SELECT :firstguy, :secondguy, boss
 FROM ReportsTo
 WHERE emp = :firstguy
 AND boss IN (
   SELECT boss
   FROM ReportsTo
   WHERE emp = :secondguy
  );
```

3. The simplest constraints should ensure that lft and rgt are unique and positive numbers:

```
CREATE TABLE Personnel (
 emp CHAR (10) PRIMARY KEY,
 salary DECIMAL (8,2) NOT NULL CHECK (salary >= 0.00),
 lft INTEGER NOT NULL UNIQUE CHECK(lft > 0),
 rgt INTEGER NOT NULL UNIQUE CHECK(rgt > 0),
 CHECK (lft < rgt)
);
```

Frankly, it is probably not a good idea to get fancier than this because updates, deletes, and inserts could yield a table that is not in its final form at some step in the process. The unique will put an index on the lft and rgt columns, so you are also getting a query performance boost.

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)
);
```

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.

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		

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.

Trees -- Part IV

Yes, I know that the piece on trees was supposed to be a three-part article. I lied. Well, actually, there are a few things I need to say about optimizing the nested set model of trees. Some of you have already discovered these tricks. Ted Holt of Midrange Computing and several others pointed out that the where clause in queries such as:

```
SELECT DISTINCT B1.emp, (rgt - lft) AS height
FROM Personnel AS B1, Personnel AS E1
WHERE E1.lft BETWEEN B1.lft AND B1.rgt -- required
   AND E1.rgt BETWEEN B1.lft AND B1.rgt; -- redundant
```

does not need the second between predicate on the `rgt` columns of an employee. If their `lft` value is between the values of the `lft` and `rgt` columns of their bosses further up in the hierarchy, then the `rgt` value also has to be contained within the boss. Be careful; this trick will not work if you drop the first between predicate, which is based on the rgt value.
This is a little hard to see the first time, so you might want to draw a picture and convince yourself. Because both comparisons are being performed on the same rows, there is little or no performance hit, but the extra predicate is redundant.
The constraints that I have shown on the tables are easy to implement in almost all SQL-92 products, but they are not complete. For example, you have nothing that prevents two sets from overlapping instead of nesting. A simple query to locate such problems is:

```
SELECT P1.emp, ' overlaps ', P2.emp
 FROM Personnel AS P1, Personnel AS P2
 WHERE P2.lft BETWEEN P1.lft AND P1.rgt
  AND P1.rgt BETWEEN P2.lft AND P2.rgt
  AND P1.emp < P2.emp;
```

In a full implementation of SQL-92, this could be converted into the subquery of a not exists( ) predicate in a check( ) clause on the table.

Another handy validation trick is a view of the numbers currently being used in the lft and rgt columns of an Organization table:

```
CREATE VIEW OrgLftRgt (num)
AS SELECT lft FROM Organization
UNION
SELECT rgt FROM Organization;
```

Some SQL products will still not allow you to use a union in a view, so this might not work for you. Following are some queries that you can use to check the status of the tree:

```
SELECT 'Tree has duplicate node numbers = ', num
 FROM OrgLftRgt
 GROUP BY num
 HAVING COUNT(*) > 1;

SELECT 'Tree has gaps in node numbers'
 FROM OrgLftRgt
 GROUP BY num
 HAVING COUNT(*) <> MAX(num);
```

Another tip is to put the tree structure information in one table and the node information in a second table. The tree table will be quite small and you can change the structure or the nodes independently of one another. For example, in the examples in this series I used a simplified Personnel table that looked like the following:

```
CREATE TABLE Personnel(
 emp_id CHAR(10) PRIMARY KEY,
 salary DECIMAL(8, 2) NOT NULL CHECK (salary >= 0.00),
 lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
 rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 0),
 CHECK (lft < rgt)
);
```

What you would have had in a real schema, which would have far more data items, is several tables; one for the tree structure itself and one for each of the entities involved in a node (in this example, that would be employees and the job positions):

```
CREATE TABLE Organization (
 position CHAR(10) NOT NULL PRIMARY KEY,
 emp_id CHAR(10) NOT NULL REFERENCES Personnel (emp_id),
 lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
 rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),
 CHECK (lft < rgt)
);
```

```
CREATE TABLE Personnel (
 emp_id CHAR(10) NOT NULL PRIMARY KEY,
 emp_name CHAR(30) NOT NULL,
 emp_address CHAR(30) NOT NULL,
 salary DECIMAL(8, 2) NOT NULL CHECK (salary >= 0.00),
 ...
);
```

```
CREATE TABLE Positions (
 position CHAR(10) NOT NULL PRIMARY KEY,
 job_title CHAR(20) NOT NULL,
 high_salary DECIMAL(10, 2) NOT NULL,
 low_salary DECIMAL(10, 2) NOT NULL,
 ...
);
```

You then join the tables to see all of the details of who holds which position. Notice that the way we have this set up, the same person can hold multiple positions within an organization. If you wish to disallow this, put a unique constraint on the emp_id column in the Organization table. Generally speaking, a organizational chart disallows multiple roles and a parts explosion has lots of them.

As an aside, the adjacency matrix model has problems in separating the hierarchical structure and the node data. For example, consider the sample table:

Personnel

emp	boss	salary
'Jerry'	NULL	1000.00
'Bert'	'Jerry'	900.00
'Chuck'	'Jerry'	900.00
'Donna'	'Chuck'	800.00
'Eddie'	'Chuck'	700.00
'Fred '	'Chuck'	600.00

Employee Chuck decides that he will now be called Charles instead, in fitting with his new promotion. This means that you must change a primary key in the table, which does happen in the real world. Oops, we also have to change Chuck to Charles in the boss column, so that Donna, Eddie, and Fred answer to the correct boss. This means that the table declaration should be:

```
CREATE TABLE Personnel (
 emp CHAR(20) PRIMARY KEY REFERENCES Personnel(boss)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
 boss CHAR(20) REFERENCES Personnel(emp)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
 salary DECIMAL(6,2) NOT NULL
);
```

Let's say you decide to fire Charles, but first you need to reassign Donna, Eddie, and Fred to a new boss or terminate them and their subordinates because the referential constraint will not let us just drop Charles from the table.

Puzzle

Sissy Kubu sent me a strange question on CompuServe. She has a table like this:

```
CREATE TABLE Inventory (
 goods CHAR(10) NOT NULL PRIMARY KEY,
 pieces INTEGER NOT NULL CHECK (pieces >= 0)
);
```

She wants to deconsolidate the table; that is, get a view or table with one row for each pieces. For example, given a row with `('cd-rom', 3)` in the original table, she would like to get three rows with `('cd-rom', 1)` in them. Before you ask me, I have no idea why she wants to do this; consider it a training exercise.

Because SQL has no "un-count(*) ... de-group by.." operators, you will have to use a cursor or the vendor's 4GL to do this. Frankly, I would do this in a report program instead of a SQL query because the results will not be a table with a key.

The obvious procedural way to do this would be to write a routine in your SQL's 4GL that reads a row from the Inventory table, and then writes the value of good to a second table in a loop

driven by the value of pieces. This will be pretty slow, because it requires (select sum(pieces) from Inventory) single-row insertions into the working table. Can you do better?

Puzzle Answer

I always stress the need to think in terms of sets in SQL. The way to build a better solution is to perform repeated self-insertion operations, using a technique based on the "Russian peasant's algorithm," which was used for multiplication and division in early computers. You can look it up in a history of mathematics or a computer science book -- it is based on binary arithmetic and can be implemented with right and left shift operators in assembly languages.

You will still need a 4GL to do this, but it won't be so bad. First, create two working tables and one for the final answer:

```
CREATE TABLE WorkingTable1 (
 goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL
);
CREATE TABLE WorkingTable2 (
 goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL
);
CREATE TABLE Answer (
 goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL
);
```

Now start by loading the goods that have only one piece in inventory into the answer table:

```
INSERT INTO Answer SELECT * FROM Inventory WHERE pieces = 1;
```

Now put the rest of the data into the first working table:

```
INSERT INTO WorkingTable1 SELECT * FROM Inventory WHERE pieces > 1;
```

The following block of code will load the second working table with pairs of rows that each have half (or half plus one) the piece counts of those in the first working table:

```
INSERT INTO WorkingTable2
 SELECT goods, FLOOR(pieces/2.0)
 FROM WorkingTable1
 WHERE pieces > 1
 UNION ALL
 SELECT goods, CEILING(pieces/2.0)
 FROM WorkingTable1
 WHERE pieces > 1;
```

The floor(x) and ceiling(x) functions return, respectively, the greatest integer that is lower than x and smallest integer that is higher than x. If your SQL does not have them, you can write them with rounding and truncation functions. It is also important to divide by (2.0) and not by 2 because this will make the result into a decimal number.

Now harvest the rows that have gotten down to a piece count of one and clear out the first working table:

```
INSERT INTO Answer
SELECT *
FROM WorkingTable2
WHERE pieces = 1;
DELETE FROM WorkingTable1;
```

Exchange the roles of WorkingTable1 and WorkingTable2, and repeat the process until both working tables are empty. That is simple, straightforward procedural coding. The ways that the results shift from table to table are interesting to follow. Think of these diagrams as an animated cartoon:

Step One: Load the first working table, harvesting any goods that already had a piece count of one.

```
WorkingTable1                 WorkingTable2
goods      pieces             goods      pieces
========== =======            ========== =======
Alpha         4
Beta          5
Delta        16
Gamma        50
```

The row (Epsilon, 1) goes immediately to Answer table.

Step Two: Halve the piece counts and double the rows in the second working table. Empty the first working table.

```
WorkingTable1                 WorkingTable2
goods      pieces             goods      pieces
========== =======            ========== =======
                              Alpha         2
                              Alpha         2
                              Beta          2
                              Beta          3
                              Delta         8
                              Delta         8
                              Gamma        25
                              Gamma        25
```

Step Three: Repeat the process until both working tables are empty.

```
WorkingTable1                  WorkingTable2
goods      pieces              goods      pieces
========== =======             ========== =======
Alpha        1
Alpha        1
Alpha        1
Alpha        1
Beta         1  Alpha and Beta are ready to harvest
Beta         1
Beta         1
Beta         1
Beta         1
 --------------
Delta        4
Delta        4
Delta        4
Delta        4
Gamma       12
Gamma       12
Gamma       13
Gamma       13
```

The cost of completely emptying a table is usually very low. Likewise, the cost of copying sets of rows (which are in physical blocks of disk storage that can be moved as whole buffers) from one table to another is much lower than inserting one row at a time.

The code could have been written to leave the results in one of the working tables, but instead this approach allows the working tables to get smaller and smaller so that you get better buffer usage. This algorithm uses (select sum(pieces) from Inventory) rows of storage and (log2((select max(pieces) from Inventory)) + 1) moves, which is pretty good on both counts.

-- Joe Celko