January, 2006

Querying Hierarchical Data

Learn how to get the results you need without storing anything extra

By Tamar E. Granor, Technical Editor

Have you ever tried modeling the organization chart for a business? Each employee (except, of course, the head honcho) reports to someone above himself or herself in the hierarchy, but everyone is an employee. How can you represent such a situation in your database?

How about a bill of materials, where each item may be built out of other items? For example, a bracket assembly might be a bracket and the necessary hardware and a shelf assembly might be a shelf, and two bracket assemblies. How do you represent this situation in a database?

There are two approaches to dealing with this kind of data. In the first approach, the database stores a pointer from a record to its immediate parent, but no additional information about the record's position in the hierarchy. That makes it easy to add records and update them, but poses challenges for reporting.

With the second strategy, additional information about the hierarchy is stored for each record. This simplifies reporting, since you can look at a single record and know its heritage, but makes the task of creating and maintaining records more complicated.

This article looks at the first approach and shows you how to extract the data you need for reporting. For a discussion of the second approach, check out http://www.sqlteam.com/item.asp?ItemID=8866.

Modeling an Organization Chart

The simple storage model asks each record to keep track of its parent (or parents) in the hierarchy. For example, consider an Employee table, with a field pointing to the employee's supervisor:

```
Field Name          Type                Width
IID                 Integer (AutoInc)       4
CFIRST              Character              15
CLAST               Character              20
ISUPERVISO          Integer                 4
```

Here, iID is the primary key for the employee and iSuperviso is a foreign key to the employee's supervisor. iSuperviso points back into the Employee table and is null for the big boss. You'll find a sample Employee table with this structure on this month's Professional Resource CD.

What are the questions we might reasonably ask about the organizational chart?

- Who is each employee's (or a specific employee's) immediate supervisor?
- Who does a given employee supervise?
- Who's the boss? That is, who is the ultimate supervisor?
- Which employees are supervisor? Which are not?
- What is the chain of supervision from one employee to the top?

You can probably think of a few others, but this list addresses the key problems.

The first two questions can be answered with a single query each. Both involve a self-join, a query where a table is joined to itself. Doing that requires the use of a local alias, a name assigned to the table for the length of the query.

Listing 1 (EmpWithSup.PRG on this month's PRD) shows how to get the list of employees and their immediate supervisors; Figure 1 shows the first few records of the results. The query uses the Employee table twice. The first instance uses the local alias Emp, while the second has the local alias Sup. The query also uses AS to assign unambiguous names to the fields of the result.

Listing 1. One level is easy—Looking one level up in a hierarchy requires a self-join, but is not difficult.

```
SELECT Emp.cFirst AS cEmpFirst, Emp.cLast AS cEmpLast,
       Emp.iSupervisor, ;
       Sup.cFirst AS cSupFirst, Sup.cLast AS cSupLast;
   FROM Employee Emp ;
     LEFT JOIN Employee AS Sup ;
       ON Emp.iSuperviso = Sup.iID ;
   ORDER BY cSupLast, cSupFirst, cEmpLast, cEmpFirst ;
   INTO CURSOR EmpWithSup
```

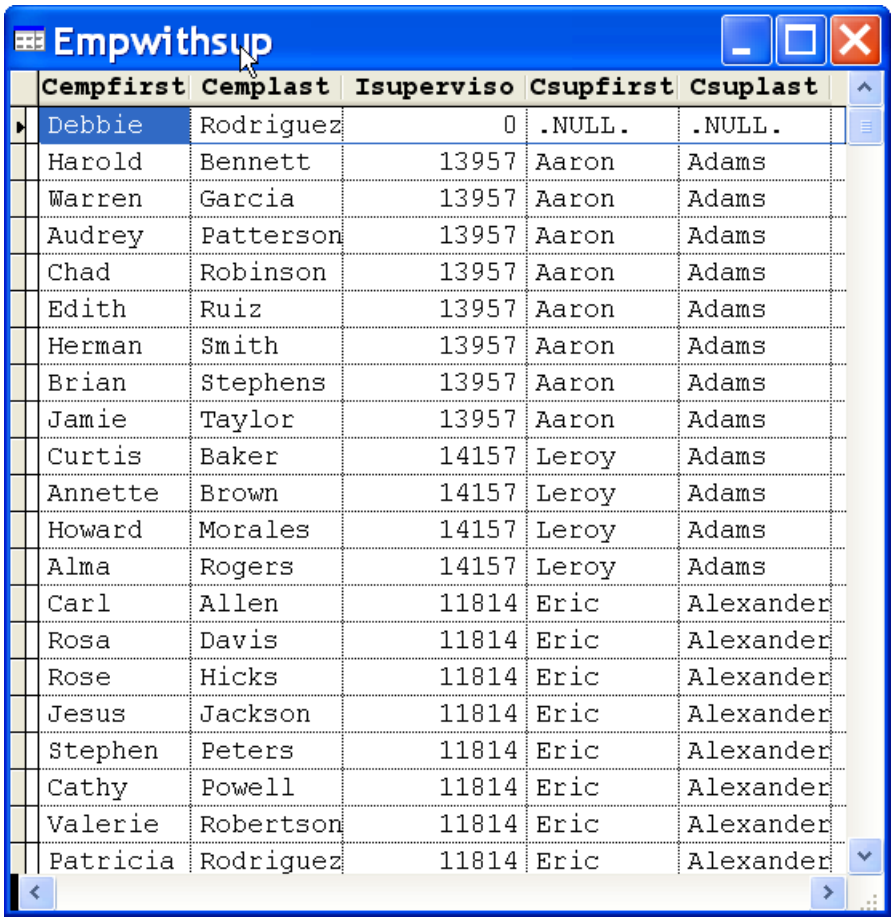


Figure 1. Employees and their supervisors—The query in Listing 1 matches each employee with his or her supervisor.

The same query actually answers the second question—who does a given employee supervise? If you're really interested in only a single supervisor, simply add a filter to the WHERE clause restricting the results, as in Listing 2 (OneSupervisor.PRG on this month's PRD).

Listing 2. Narrow it down—To see only those who report to a given supervisor, simply add a filter.

```
SELECT Emp.cFirst AS cEmpFirst, Emp.cLast AS cEmpLast, ;
       Emp.iSuperviso, ;
       Sup.cFirst AS cSupFirst, Sup.cLast AS cSupLast;
   FROM Employee Emp ;
     LEFT JOIN Employee AS Sup ;
       ON Emp.iSuperviso = Sup.iID ;
   WHERE Sup.cFirst = "Debbie" AND Sup.cLast = "Rodriguez" ;
   ORDER BY cSupLast, cSupFirst, cEmpLast, cEmpFirst ;
   INTO CURSOR EmpWithSup
```

Figuring out who's at the top of the hierarchy is also simple. It doesn't require any joins at all; you just need to find the employee who has no

supervisor, as Listing 3 (HeadHoncho.PRG on this month's PRD) shows.

Listing 3. Who's in charge?—Finding the top boss is straightforward.

```
SELECT cFirst, cLast ;
   FROM Employee ;
   WHERE EMPTY(iSuperviso) ;
   INTO CURSOR HeadHoncho
```

The next pair of questions seeks to divide the employees into those who are supervisors and those who are not. The solutions to these two problems turn out to be very similar. Supervisors are all those who are referenced by at least one other employee's record. Listing 4 (Supervisors.PRG on this month's PRD) shows how to find their names.

Listing 4. Finding supervisor—Any employee referenced in another employee's iSuperviso field is a supervisor.

```
SELECT cFirst, cLast ;
   FROM Employee ;
   WHERE iID IN ;
     (SELECT iSuperviso FROM Employee) ;
   ORDER BY cLast, cFirst ;
   INTO CURSOR AllSupervisors
```

The subquery here extracts the supervisor ID from each employee record. The main query then finds the records for those individuals. While you can use a join rather than a subquery in this case, you then end up with duplicate records (one for each employee under a given supervisor); in that case, you need to use DISTINCT to cut the list down to one record per supervisor.

To find those who aren't supervisors, add NOT to the previous query. In this case, you can't replace the subquery with a join. Listing 5 (WorkerBees.PRG on this month's PRD) shows the solution.

Listing 5. Worker bees—To find the employees who don't supervise anyone, throw out the list of supervisors and keep the rest of the records.

```
SELECT cFirst, cLast ;
   FROM Employee ;
   WHERE iID NOT IN ;
     (SELECT iSuperviso FROM Employee) ;
   ORDER BY cLast, cFirst ;
   INTO CURSOR WorkerBees
```

The last question on the list can't be answered with just a query, unless you know how many levels there are in the hierarchy. Tracing

from a given employee to the top of the hierarchy calls for some procedural code. Listing 6 (EmpHierarchy.PRG on this month's PRD) shows a solution that combines procedural code with a query and works in all versions of VFP.

Listing 6. Tracing the hierarchy—One way to work back from a single employee to the big boss combines a query with procedural code.

```
LPARAMETERS iEmpID

LOCAL iCurrentID

CREATE CURSOR EmpHierarchy ;
   (cFirst C(15), cLast C(20))

iCurrentID = iEmpID
DO WHILE NOT EMPTY(iCurrentID)

   SELECT cFirst, cLast, iSuperviso ;
      FROM Employee ;
       WHERE iID = m.iCurrentID ;
       INTO CURSOR NextEmp NOFILTER

   INSERT INTO EmpHierarchy ;
      VALUES (NextEmp.cFirst, NextEmp.cLast)

   iCurrentID = NextEmp.iSuperviso
ENDDO

USE IN NextEmp
USE IN Employee
SELECT EmpHierarchy
```

The code receives the employee to start with as a parameter. Each time through the loop, you find the supervisor for the current employee and add it to the result. Then, focus on that supervisor for the next pass through the loop. Continue until you reach the top of the chain.

Listing 7 (EmpHierarchyProc.PRG on this month's PRD) shows another approach that replaces the query with SEEK. This version is 5-10% faster than the previous version. The price you pay is having to explicitly open the Employee table.

Listing 7. Tracing the hierarchy—Another approach to tracing the hierarchy uses the Xbase SEEK instead of a query.

```
LPARAMETERS iEmpID

LOCAL iCurrentID

CREATE CURSOR EmpHierarchy ;
```

```
   (cFirst C(15), cLast C(20))

USE Employee IN 0 ORDER iID

iCurrentID = iEmpID
DO WHILE NOT EMPTY(iCurrentID)

   SEEK iCurrentID IN Employee

   INSERT INTO EmpHierarchy ;
      VALUES (Employee.cFirst, Employee.cLast)

   iCurrentID = Employee.iSuperviso
ENDDO

USE IN Employee
SELECT EmpHierarchy
```

In VFP 8 and later, there's another way to do this. The INSERT command accepts a query instead of a list of values, so you can combine the SELECT and INSERT of the first version into a single command. Listing 8 (EmpHierarchy2.PRG on this month's PRD) shows this approach, which is very slightly (less than 1%) slower than the version in Listing 6.

Listing 8. Tracing the hierarchy—In VFP 8 and 9, you can combine INSERT and SELECT to simplify the code.

```
LPARAMETERS iEmpID

LOCAL iCurrentID

CREATE CURSOR EmpHierarchy ;
   (cFirst C(15), cLast C(20), iSuperviso I)

iCurrentID = iEmpID
DO WHILE NOT EMPTY(iCurrentID)

   INSERT INTO EmpHierarchy ;
      SELECT cFirst, cLast, iSuperviso ;
         FROM Employee ;
          WHERE iID = m.iCurrentID ;

   iCurrentID = EmpHierarchy.iSuperviso
ENDDO

USE IN Employee
SELECT EmpHierarchy
```

The Bill of Materials Problem

With an organization chart, there's a single hierarchy. Each employee has a single supervisor. The bill of materials problem is more complex. Here, there are some basic materials (such as shelves, nuts, bolts, brackets, etc.) that can be combined in various ways to create more complex materials (bracket assemblies comprising a bracket and the necessary hardware, for example) and those items can themselves be used to create even more complex materials (such as shelf assemblies).

Modeling this structure requires more than one table. One table (Items, in Figure 2) lists all the items available, while another (Contents) links items to the other items used to build them. Each record in Contents has two foreign keys to the Items table, one for the item being built (iContID), and one for the contained item (iItemID). It also has a field to indicate how many of the specified part are needed. For example, for a bracket assembly, the Contents table might have two records, one for the bracket and one for the screws used to attach the bracket. Figure 3 and Figure 4 show the Items and Contents tables, respectively. (They're also included on this month's PRD and in the downloads for this article.)

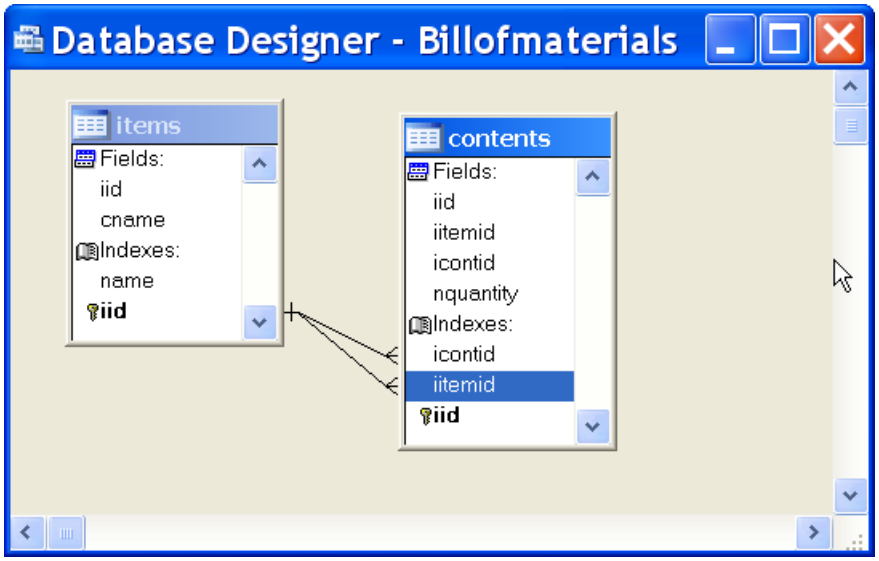
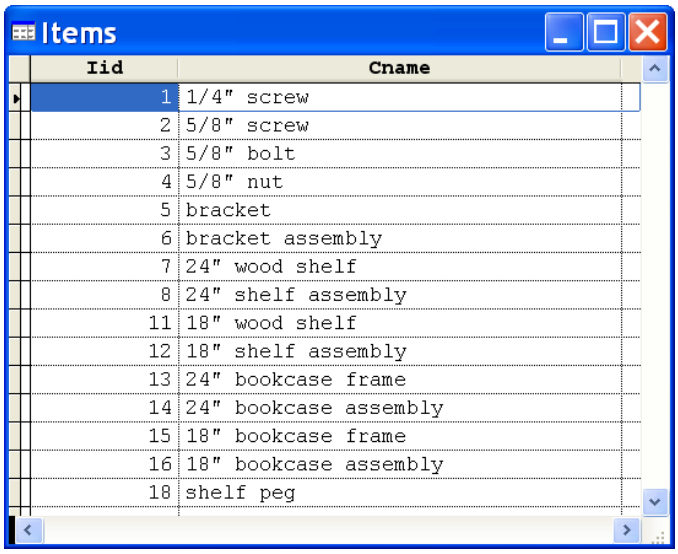


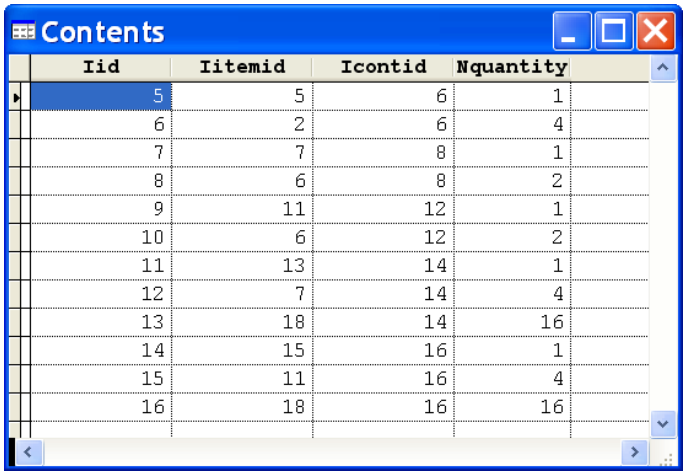
Figure 2. Representing a bill of materials—You normally use two tables to represent a bill of materials situation, where items can be constructed out of other items.



Items

Iid	Cname
1	1/4" screw
2	5/8" screw
3	5/8" bolt
4	5/8" nut
5	bracket
6	bracket assembly
7	24" wood shelf
8	24" shelf assembly
11	18" wood shelf
12	18" shelf assembly
13	24" bookcase frame
14	24" bookcase assembly
15	18" bookcase frame
16	18" bookcase assembly
18	shelf peg

Figure 3. Items available—The Items table lists all the parts that can be used and sold.



Contents

Iid	Iitemid	Icontid	Nquantity
5	5	6	1
6	2	6	4
7	7	8	1
8	6	8	2
9	11	12	1
10	6	12	2
11	13	14	1
12	7	14	4
13	18	14	16
14	15	16	1
15	11	16	4
16	18	16	16

Figure 4. Combining items—The Contents table shows which items go into building other items.

There are basically two questions to ask about bill of materials data:

- What parts are needed to build a particular item?
- What items use a particular part?

It turns out that each of the questions has a couple of variants, as well. The first question is probably the most common. The answer lets you create the part list that can be used to assemble the item.

Building the part list requires you to climb down the hierarchy. At each stage, you have to check the items you find to see whether they contain additional items. Because you may find several items at each

stage, you need a way to keep track of the items yet to be checked, such as storing them in a cursor.

As you build the list, you may or may not want to include assemblies in the result. For example, when building the part list for a shelf assembly, you may want to show that there are two bracket assemblies and then show the parts contained in those assemblies. Alternatively, you may simply want to show the parts and leave out any reference to the bracket assemblies.

The code in Listing 9 (PartList.PRG on this month's PRD) builds the part list, optionally removing the assemblies. In addition, it sorts the results so that if the assemblies are listed, each is followed by the items used to build the assembly. That makes reporting on the results easier. Note also that the code figures out the quantity needed for each item, using the quantity of the part in a given assembly and the quantity of that assembly.

Listing 9. Building a part list—You need to drill down checking each item you find to see whether it's composed of additional parts.

```
LPARAMETERS iItemID, lDeleteAssemblies

LOCAL iCurrentID, nCurPart, lDeleteCurRec

iCurrentID = m.iItemID

SELECT iItemID, cName, nQuantity, ;
       m.iCurrentID AS iAssemblyID, 000 AS nSortOrder ;
   FROM Contents ;
     JOIN Items ;
       ON Contents.iItemID = Items.iID ;
   WHERE Contents.iContID = m.iCurrentID ;
   INTO CURSOR Parts READWRITE

nSortOrder = 1

SCAN

   nCurPart = RECNO("Parts")
   iCurrentID = Parts.iItemID
   nHowMany = Parts.nQuantity
   IF nSortOrder = 0
      REPLACE nSortOrder WITH m.nSortOrder
   ENDIF

   INSERT INTO Parts ;
      SELECT iItemID, cName, m.nHowMany * nQuantity, ;
             m.iCurrentID, m.nSortOrder + 1 ;
        FROM Contents ;
          JOIN Items ;
```

```
            ON Contents.iItemID = Items.iID ;
         WHERE Contents.iContID = m.iCurrentID
   lDeleteCurRec = lDeleteAssemblies AND (_TALLY > 0)
   m.nSortOrder = m.nSortOrder + _TALLY + 1

   GO (nCurPart) IN Parts
   IF lDeleteCurRec
      DELETE NEXT 1
   ENDIF

ENDSCAN

SELECT * FROM Parts ;
   INTO CURSOR Parts ;
   ORDER BY nSortOrder
```

The other interesting question is which items use a given part, that is, building a list of products that include a specified part. You might do this when it becomes difficult to get a particular part, so that you can mark certain items as unavailable or look for substitutes.

As with the organizational chart, if you only want to go up one level, this is a simple problem and the necessary query is analogous to Listing 1. Listing 10 (UsedDirectly.PRG on this month's PRD) shows how to find this information for all items.

Listing 10. Who uses what?—A single query can build a list showing direct use of each item.

```
SELECT Items.iID, Items.cName, iContID, Assem.cName ;
   FROM Items ;
     JOIN Contents ;
       ON Items.iID = Contents.iItemID ;
     JOIN Items Assem ;
       ON Contents.iContID = Assem.iID ;
   ORDER BY Items.iID ;
   INTO CURSOR UsedDirectly
```

The harder problem is tracing all the way back to the top of the hierarchy for a given item. There are two variations here, finding only final products that use the specified item, and finding all items that use the item. The big difference between the two is whether you keep all the records you find as you climb the hierarchy.

Listing 11 (UsesItem.PRG on this month's PRD) shows code to build a list of only the final products. It uses two cursors, one to hold the list of items to be checked, and one to hold the results.

Listing 11. Products using an item—To build the list of final products that use a particular part, you have to trace up the hierarchy.

```
LPARAMETERS iItemID

LOCAL iCurrentID, nRecNo

CREATE CURSOR UsesItem (iItemID I, cName C(25))

iCurrentID = m.iItemID

SELECT iContID ;
   FROM Contents ;
   WHERE iItemID = m.iCurrentID ;
   INTO CURSOR CheckItems READWRITE

SCAN
   iCurrentID = CheckItems.iContID
   nRecNo = RECNO("CheckItems")

   * Look for items containing the current item
   INSERT INTO CheckItems ;
      SELECT iContID ;
         FROM Contents ;
         WHERE iItemID = m.iCurrentID
   GO nRecNo IN CheckItems

   IF _Tally = 0
      * This is the top of the chain
      INSERT INTO UsesItem ;
         SELECT iID, cName ;
            FROM Items ;
            WHERE iID = m.iCurrentID
   ENDIF

ENDSCAN

SELECT UsesItem
```

If you want to see all the items that use a particular part, not just those that aren't contained in any other items, you can use a single cursor to hold both the items yet to check and the results. The code in Listing 12 (ContainerList.PRG on this month's PRD) does so, being careful to keep track of the record pointer position. The code here is surprisingly similar to the code in Listing 9 that traces the hierarchy downwards.

Listing 12. All items using an item—Creating a list of all items that use a particular item, whether those items are used in other items or not, is similar to finding all the items used in a particular product.

```
LPARAMETERS iItemID
```

```
LOCAL iCurrentID

iCurrentID = m.iItemID

SELECT iContID as iItemID, cName ;
   FROM Contents ;
     JOIN Items ;
       ON Contents.iContID = Items.iID ;
   WHERE Contents.iItemID = m.iCurrentID ;
   INTO CURSOR Containers READWRITE

SCAN

   nCurPart = RECNO("Containers")
   iCurrentID = Containers.iItemID

   INSERT INTO Containers ;
      SELECT iContID, cName ;
         FROM Contents ;
           JOIN Items ;
             ON Contents.iContID = Items.iID ;
         WHERE Contents.iItemID = m.iCurrentID

   GO (nCurPart) IN Containers

ENDSCAN
```

The Bottom Line

The world is full of hierarchies, both simple and complex. The approach demonstrated in this article is appropriate in situations where the data in the hierarchy is likely to change and where tracing the hierarchy quickly isn't a high priority. For more static hierarchies where quick access to an item's lineage is important, consider storing that information with each record.