Handling hierarchical data

SQL Server offers two different tools that make working with hierarchies like organization charts and bills of materials much easier than in VFP.

Tamar E. Granor, Ph.D.

In my last article, I showed how SQL Server makes combining data into a single field much easier than in VFP. This time, I'll show how handling hierarchical data that doesn't fit the standard parent-child-grandchild model is easier in SQL Server.

Relational databases handle typical hierarchical relationships very well. When you have something like customers, who place orders, which contain line items, representing products sold, any relational database should do. You create one table for each type of object and link them together with foreign keys.

Reporting on such data is easy, too. Fairly simple SQL queries let you collect the data you want with a few joins and some filters.

But some types of data don't lend themselves to this sort of model. For example, the organization chart for a company contains only people, with some people managed by other people, who might in turn be managed by other people. Clearly, records for all people should be contained in a single table.

But how do you represent the manager relationship? One commonly used approach is to add a field to the person's record that points to the record (in the same table) for his or her manager.

From a data-modeling point of view, this is a simple solution. However, reporting on such data can be complex. How do you trace the hierarchy from a given employee through her manager to the manager's manager and so on up the chain of command? Given a manager, how do you find everyone who ultimately reports to that person (that is, reports to the person directly, or to someone managed by that person, or to someone managed by someone who is managed by that person, and so on down the line)?

This article looks at two approaches to dealing with this kind of data, and show how much easier it is to get what you want in SQL Server than in VFP.

The traditional solution

As described above, the traditional way to handle this type of hierarchy is to add a field to identify a record's parent (such as an employee's manager). For example, the Northwind database that comes with VFP has a field in the Employees table called ReportsTo. It contains the primary key of the employee's manager; since that's also a record in Employees, the table is self-referential.

The AdventureWorks 2008 sample database for SQL Server doesn't have this kind of relationship because it uses the second approach to hierarchies, discussed later in this article. However, the 2005 version of the database has a set-up quite similar to the one in Northwind. The Employee table has a ManagerID field that contains the primary key (in Employee) of the employee's manager. (You can download the 2005 version of AdventureWorks from http://tinyurl.com/y943xr9 and the 2008 version from http://tinyurl.com/cp2fv8w.)

Using the VFP Northwind and SQL Server AdventureWorks 2005 databases, let's try to answer some standard questions about an organization chart.

Who manages an employee?

In both cases, determining the manager of an individual employee is quite simple. It just requires a self-join of the Employee table. That is, you use two instances of the Employee table, one to get the employee and one to get the manager. Listing 1 (EmpPlusMgr.PRG in this month's downloads) shows the VFP version of the query that retrieves this data for a single employee (by specifying the employee's primary key-4, in this case).

Listing 1. Use a self-join to connect an employee with his or her manager.

```
SELECT Emp.FirstName AS EmpFirst, ;
       Emp.LastName AS EmpLast, ;
       Mgr.FirstName AS MgrFirst, ;
       Mgr.LastName AS MgrLast ;
  FROM Employees Emp ;
    JOIN Employees Mgr ;
      ON Emp.ReportsTo = Mgr.EmployeeID ;
  WHERE Emp.EmployeeID = 4 ;
  INTO CURSOR csrEmpAndMgr
```

The AdventureWorks version of the same task is a little more complex, because the database has a separate table for people (called Contact). The Employee table uses a foreign key to Contact to identify the individual; Employee contains only the data related to employment. So extracting an employee's name requires joining Employee to Contact.

The solution still uses a self-join on the Employee table, but now it also requires two instances of the Contact table. Listing 2 (EmpPlusMgr.SQL in this month's downloads) shows the SQL Server query to retrieve the employee's name and his or her manager's name. Again, we retrieve data for a single employee (by specifying EmployeeID=37).

Listing 2. The SQL Server version of the query is a little more complex, due to additional normalization, but still uses a self-join.

```
SELECT EmpContact.FirstName AS EmpFirst,
       EmpContact.LastName AS EmpLast,
       MgrContact.FirstName AS MgrFirst,
       MgrContact.LastName AS MgrLast
  FROM Person.Contact EmpContact
    JOIN HumanResources.Employee Emp
      ON Emp.ContactID = EmpContact.ContactID
    JOIN HumanResources.Employee Mgr
      ON Emp.ManagerID = Mgr.EmployeeID
    JOIN Person.Contact MgrContact
      ON Mgr.ContactID = MgrContact.ContactID
  WHERE Emp.EmployeeID = 37
```

It's easy to extend these queries to retrieve the names of all employees with each one's manager. Just remove the WHERE clause from each query.

What's the management hierarchy for an employee?

Things start to get a lot more interesting when you want to trace the whole management hierarchy for an employee. That is, given a particular employee, retrieve the name of her manager and of the manager's manager and of the manager's manager's manager and so on up the line until you reach the person in charge.

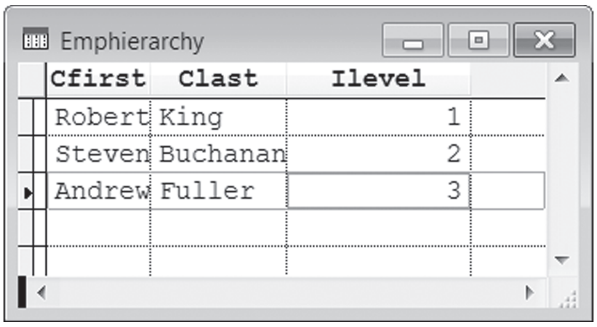
Since you don't know how many levels you might have, rather than putting all the data into a single record, here we create a cursor with one record for each level. The specified employee comes first, and then you climb the hierarchy so that the big boss is last.

VFP's SQL alone doesn't offer a solution for this problem. Instead, you need to combine a little bit of SQL with some Xbase code, as in Listing 3. (This program is included in this month's downloads as EmpHierarchy.PRG.)

Listing 3. To track a hierarchy to the top in VFP calls for a mix of SQL and Xbase code.

```
* Start with a single employee and create a
* hierarchy up to the top dog.
LPARAMETERS iEmpID

LOCAL iCurrentID , iLevel

OPEN DATABASE HOME(2) + "Northwind\Northwind"

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

USE Employees IN 0 ORDER EmployeeID

iCurrentID = iEmpID
iLevel = 1
DO WHILE NOT EMPTY(iCurrentID)

   SEEK iCurrentID IN Employees

   INSERT INTO EmpHierarchy ;
      VALUES (Employees.FirstName, ;
              Employees.LastName, ;
              m.iLevel)

   iCurrentID = Employees.ReportsTo
   iLevel = m.iLevel + 1
ENDDO

USE IN Employees
SELECT EmpHierarchy
```

The strategy is to start with the employee you're interested in, insert her data into the result cursor, then grab the PK for her manager and repeat until you reach an employee whose PK is empty. Figure 1 shows the results when you pass 7 as the parameter.



Cfirst	Clast	Ilevel
Robert	King	1
Steven	Buchanan	2
Andrew	Fuller	3

Figure 1. Running the query in Listing 3, passing 7 as the parameter, gives these results.

SQL Server provides a simpler solution, by using a Computed Table Expression (CTE). A CTE is a query that precedes the main query and provides a result that is then used in the main query. While similar to a derived table, CTEs have a couple of advantages.

First, the result can be included multiple times in the main query (with different aliases). A derived table is created in the FROM clause; if you need the same result again, you have to include the whole definition for the derived table again.

Second, and relevant to this problem, a CTE can have a recursive definition, referencing itself. That allows it to walk a hierarhcy.

Listing 4 shows the structure of a query that uses a CTE. (It's worth noting that a single query can have multiple CTEs; just separate them with commas.)

Listing 4. The definition for a CTE precedes the query that uses it.

```
WITH CTEAlias(Field1, Field2, ...)
AS
(
 SELECT <fieldlist>
   FROM <tables>
   ...
)
SELECT <main fieldlist>
  FROM <main query tables>
  ...
```

The query inside the parentheses is the CTE; its alias is whatever you specify in the WITH line. The WITH line also must contain a list of the fields in the CTE, though you don't indicate their types or sizes.

The main query follows the parentheses and presumably includes the CTE in its list of tables and some of the CTE's fields in the field list.

For a recursive CTE, you combine two queries with UNION ALL. The first query is an "anchor"; it provides the starting record or records. The second query references the CTE itself to drill down recursively.

A recursive CTE continues drilling down until the recursive portion returns no records.

Listing 5 shows a query that produces the management hierarchy for the employee whose EmployeeID is 37. (Just change the assignment to @iEmpID to specify a different employee.) The query is included in this month's downloads as EmpHierarchyViaCTE.SQL.

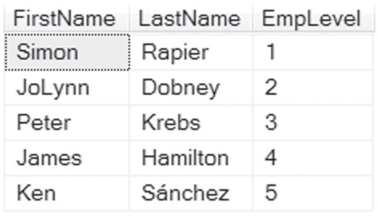
Listing 5. To retrieve the management hierarchy for an employee in the SQL Server AdventureWorks 2005 database, use a Computed Table Expression.

```
DECLARE @iEmpID INT = 37;

WITH EmpHierarchy (
  FirstName, LastName, ManagerID, EmpLevel)
AS
(
SELECT Contact.FirstName, Contact.LastName,
       Employee.ManagerID, 1 AS EmpLevel
  FROM Person.Contact
    JOIN HumanResources.Employee
      ON Employee.ContactID =
         Contact.ContactID
  WHERE EmployeeID = @iEmpID
UNION ALL
SELECT Contact.FirstName, Contact.LastName,
       Employee.ManagerID,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel
  FROM Person.Contact
    JOIN HumanResources.Employee
      ON Employee.ContactID =
         Contact.ContactID
    JOIN EmpHierarchy
      ON Employee.EmployeeID =
         EmpHierarchy.ManagerID
)

SELECT FirstName, LastName, EmpLevel
  FROM EmpHierarchy
```

The alias for the CTE here is EmpHierarchy. The anchor portion of the CTE selects the specified person (WHERE EmployeeID = @iEmpID), including that person's ManagerID in the result and setting up a field to track the level in the database.

The recursive portion of the query joins the Employee table to the EmpHierarchy table-in-progress (that is, the CTE itself), matching the ManagerID from EmpHierarchy to Employee. EmployeeID. It also increments the EmpLevel field, so that the first time it executes, EmpLevel is 2, and the second time, it's 3, and so forth.

Once the CTE is complete, the main query pulls the desired information from it. Figure 2 shows the result of the query in Listing 5.



FirstName	LastName	EmpLevel
Simon	Rapier	1
JoLynn	Dobney	2
Peter	Krebs	3
James	Hamilton	4
Ken	Sánchez	5

Figure 2. The query in Listing 5 returns one record for each level of the management hierarchy for the specified employee.

Who does an employee manage?

The problem gets a little tougher, at least on the VFP side, when you want to put together a list of all employees a particular person manages at all levels of the hierarchy. That is, not only those she manages directly, but people who report to those people, and so on down the line.

To make the results more meaningful, we want to include the name of the employee's direct manager in the results.

What makes this difficult in VFP is that at each level, you may (probably do) have multiple employees. You need not only to add each to the result, but to check who each of them manages. That means you need some way of keeping track of who you've checked and who you haven't.

The solution uses two cursors. One (MgrHierarchy) holds the results, while the other (EmpsToProcess) holds the list of people to check. Listing 6 shows the code; it's called MgrHierarchy.PRG in this month's downloads.

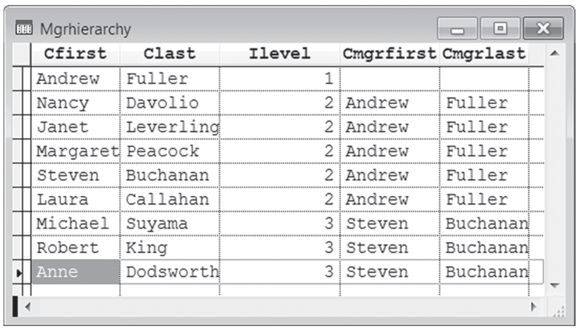
Listing 6. Putting together the list of people a specified person manages directly or indirectly is harder than climbing up the hierarchy.

```
* Start with a single employee and determine
* all the people that employee manages,
* directly or indirectly.
LPARAMETERS iEmpID

LOCAL iCurrentID, iLevel, cFirst, cLast,
LOCAL nCurRecNo, cMgrFirst, cMgrLast

OPEN DATABASE HOME(2) + "Northwind\Northwind"

CREATE CURSOR MgrHierarchy ;
  (cFirst C(15), cLast C(20), iLevel I, ;
   cMgrFirst C(15), cMgrLast C(15))
CREATE CURSOR EmpsToProcess ;
  (EmployeeID I, cFirst C(15), cLast C(20), ;
   iLevel I, cMgrFirst C(15), cMgrLast C(15))

INSERT INTO EmpsToProcess ;
  SELECT m.iEmpID, FirstName, LastName, 1, ;
         "", "" ;
    FROM Employees ;
    WHERE EmployeeID = m.iEmpID

SELECT EmpsToProcess

SCAN
  iCurrentID = EmpsToProcess.EmployeeID
  iLevel = EmpsToProcess.iLevel
  cFirst = EmpsToProcess.cFirst
  cLast = EmpsToProcess.cLast
  cMgrFirst = EmpsToProcess.cMgrFirst
  cMgrLast = EmpsToProcess.cMgrLast

  * Insert this records into result
  INSERT INTO MgrHierarchy ;
    VALUES (m.cFirst, m.cLast, m.iLevel, ;
            m.cMgrFirst, m.cMgrLast)

  * Grab the current record pointer
  nCurRecNo = RECNO("EmpsToProcess")

  INSERT INTO EmpsToProcess ;
    SELECT EmployeeID, FirstName, LastName, ;
           m.iLevel + 1, m.cFirst, m.cLast ;
      FROM Employees ;
      WHERE ReportsTo = m.iCurrentID

  * Restore record pointer
  GO m.nCurRecNo IN EmpsToProcess
ENDSCAN

SELECT MgrHierarchy
```

To kick the process off, we add a single record to EmpsToProcess, with information about the specified employee. Then, we loop through EmpsToProcess, handling one employee at a time. We insert a record into MgrHierarchy for that employee, and then we add records to EmpsToProcess for everyone directly managed by the employee we're now processing.

The most interesting bit of this code is that the SCAN loop has no problem with the cursor we're scanning growing as we go. We just have to keep track of the record pointer, and after adding records, move it back to the record we're currently processing.

Figure 3 shows the result cursor when you pass 2 as the employee ID.


Mgrhierarchy

Cfirst	Clast	Ilevel	Cmgrfirst	Cmgrlast
Andrew	Fuller	1		
Nancy	Davolio	2	Andrew	Fuller
Janet	Leverling	2	Andrew	Fuller
Margaret	Peacock	2	Andrew	Fuller
Steven	Buchanan	2	Andrew	Fuller
Laura	Callahan	2	Andrew	Fuller
Michael	Suyama	3	Steven	Buchanan
Robert	King	3	Steven	Buchanan
Anne	Dodsworth	3	Steven	Buchanan

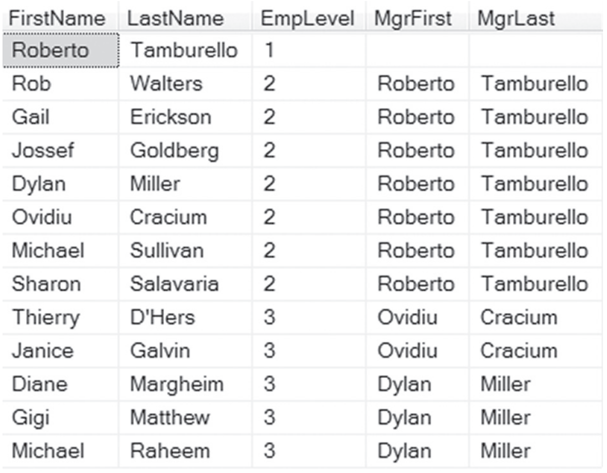
Figure 3. When you specify an EmployeeID of 2, you get all the Northwind employees.

In fact, you can do this with a single cursor that represents both the results and the list of people yet to check, but doing so makes the code a little confusing.

In SQL Server, solving this problem is no harder than solving the upward hierarchy. Again, you use a CTE, and all that really changes is the join condition in the recursive part of the CTE. (Because we want the direct manager's name, the field list is slightly different, as well). **Listing 7** shows the query (MgrHierarchyViaCTE.SQL in this month's downloads), along with a variable declaration to indicate which employee we want to start with; **Figure 4** shows the results for this example.

Listing 7. Walking down the hierarchy of employees is no harder in SQL Server than climbing up.

```
DECLARE @iEmpID INT = 3;

WITH EmpHierarchy
  (FirstName, LastName, EmployeeID, EmpLevel,
   MgrFirst, MgrLast)
AS
(
SELECT Contact.FirstName, Contact.LastName,
       Employee.EmployeeID, 1 AS EmpLevel,
       CAST('' AS NVARCHAR(50)) AS MgrFirst
       CAST('' AS NVARCHAR(50)) AS MgrLast
  FROM Person.Contact
    JOIN HumanResources.Employee
      ON Employee.ContactID =
         Contact.ContactID
  WHERE EmployeeID = @iEmpID
UNION ALL
SELECT Contact.FirstName, Contact.LastName,
       Employee.EmployeeID,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel,
       EmpHierarchy.FirstName AS MgrFirst,
       EmpHierarchy.LastName AS MgrLast
  FROM Person.Contact
    JOIN HumanResources.Employee
      ON Employee.ContactID =
         Contact.ContactID
    JOIN EmpHierarchy
      ON Employee.ManagerID =
         EmpHierarchy.EmployeeID
)

SELECT FirstName, LastName, EmpLevel,
       MgrFirst, MgrLast
    FROM EmpHierarchy
```



FirstName	LastName	EmpLevel	MgrFirst	MgrLast
Roberto	Tamburello	1		
Rob	Walters	2	Roberto	Tamburello
Gail	Erickson	2	Roberto	Tamburello
Jossef	Goldberg	2	Roberto	Tamburello
Dylan	Miller	2	Roberto	Tamburello
Ovidiu	Cracium	2	Roberto	Tamburello
Michael	Sullivan	2	Roberto	Tamburello
Sharon	Salavaria	2	Roberto	Tamburello
Thierry	D'Hers	3	Ovidiu	Cracium
Janice	Galvin	3	Ovidiu	Cracium
Diane	Margheim	3	Dylan	Miller
Gigi	Matthew	3	Dylan	Miller
Michael	Raheem	3	Dylan	Miller

Figure 4. These are the people managed by Roberto Tamburello, whose EmployeeID is 3.

Using the HierarchyID type

SQL Server 2008 introduced a new way to handle this kind of hierarchy. A new data type called HierarchyID encodes the path to any node in a hierarchy into a single field; a set of methods for the data type make both maintainance and navigation straightforward.

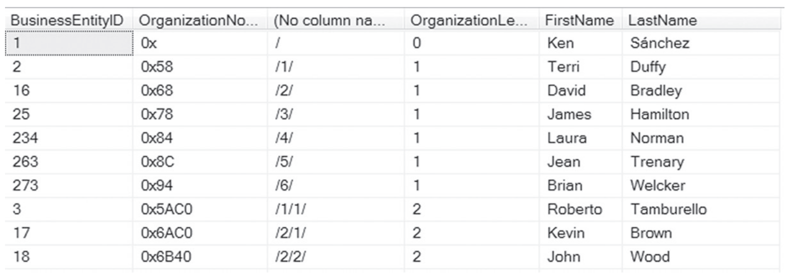
The SQL Server 2008 version of AdventureWorks use the HierarchyID type to handle the management hierachy (which is why we couldn't use it for the earlier examples). There are other changes, as well. AdventureWorks 2008 is even more normalized than the 2005 version; a new BusinessEntity table contains information about people (including employees) and businesses. So, instead of an EmployeeID, each employee now has a BusinessEntityID. In addition, the Contact table has been renamed Person. However, there's still a relationship between that table and the Employee table that we can use to retrieve an employee's name.

HierarchyID essentially creates a string that shows the path from the root (top) of the hierarchy to a particular record. The root node is indicated as "/"; then, at each level, a number indicates which child of the preceding node is in this node's hierarchy. So, for example, a hierachyID of "/4/3/" means that the node is descended from the fourth child of the root node, and is the third child of that child. However, hierarchy IDs are actually stored in a binary string created from the plain text version.

The HierarchyID type has a set of methods that allow you to easily navigate the hierarchy. First, the ToString method converts the encoded hierarchy ID to a string in the form shown above. Listing 8 (ShowHierarchyID.SQL in this month's downloads) shows a query to extract the name and hierarchy ID, both in encoded and plain text form, of the AdventureWorks employees; Figure 5 shows a portion of the result.

Listing 8. The ToString method of the HierarchyID type converts the hierarchy ID into a human-readable form.

```
SELECT Person.[BusinessEntityID]
      ,[OrganizationNode]
      ,[OrganizationNode].ToString()
      ,[OrganizationLevel]
      , FirstName
      , LastName
  FROM [HumanResources].[Employee]
    JOIN Person.Person
      ON Employee.BusinessEntityID =
         Person.BusinessEntityID
```



BusinessEntityID	OrganizationNo...	(No column na...	OrganizationLe...	FirstName	LastName
1	0x	/	0	Ken	Sánchez
2	0x58	/1/	1	Terri	Duffy
16	0x68	/2/	1	David	Bradley
25	0x78	/3/	1	James	Hamilton
234	0x84	/4/	1	Laura	Norman
263	0x8C	/5/	1	Jean	Trenary
273	0x94	/6/	1	Brian	Welcker
3	0x5AC0	/1/1/	2	Roberto	Tamburello
17	0x6AC0	/2/1/	2	Kevin	Brown
18	0x6B40	/2/2/	2	John	Wood

Figure 5. The unnamed column here shows the text version of the OrganizationNode column.

To move through the hierarchy, we use the GetAncestor method. As you'd expect, GetAncestor returns an ancestor of the node you apply it to. You pass a parameter to indicate how many levels up the hierarchy you want to go, so GetAncestor(1) returns the parent of the node.

That's actually all we need to retrieve the management hierarchy for a particular employee. As in the earlier example, we use a CTE to handle the recursive requirement. Listing 9 shows the query; it's included in this month's downloads as EmpHierarchyWithHierarchyID.SQL.

Listing 9. Retrieving the management hierarchy for a given employee when using the HierarchyID data type isn't much different from doing it with a "reports to" field.

```
DECLARE @iEmpID INT = 40;

WITH EmpHierarchy
  (FirstName, LastName,
   OrganizationNode, EmpLevel)
AS
(
SELECT Person.FirstName, Person.LastName,
       Employee.OrganizationNode,
       1 AS EmpLevel
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID =
         Person.BusinessEntityID
  WHERE Employee.BusinessEntityID = @iEmpID
UNION ALL
SELECT Person.FirstName, Person.LastName,
       Employee.OrganizationNode,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID =
         Person.BusinessEntityID
    JOIN EmpHierarchy
      ON Employee.OrganizationNode =
  EmpHierarchy.OrganizationNode.GetAncestor(1)
)

SELECT FirstName, LastName, EmpLevel
    FROM EmpHierarchy
```

The big difference between this query and the earlier query is in the join between Employee and EmpHierarchy. Rather than matching fields directly, we call GetAncestor to retrieve the hierarchy for a node's parent and compare that to the Employee table's OrganizationNode field.

As in the earlier examples, finding everyone an employee manages uses a very similar query, but in the join condition between Employee and EmpHierarchy, we apply GetAncestor to the Employee field. Listing 10 (MgrHierarchyWithHierarchyID.SQL in this month's downloads) shows the code.

Listing 10. To find everyone an individual manages using HierarchyID, just change the direction of the join between Employee and EmpHierarchy.

```
DECLARE @iEmpID INT = 3;

WITH EmpHierarchy
  (FirstName, LastName, BusinessEntityID,
   EmpLevel, MgrFirst, MgrLast, OrgNode)
AS
```

```
(
SELECT Person.FirstName, Person.LastName,
       Employee.BusinessEntityID,
       1 AS EmpLevel,
       CAST('' AS NVARCHAR(50)) AS MgrFirst,
       CAST('' AS NVARCHAR(50)) AS MgrLast,
       OrganizationNode AS OrgNode
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID =
         Person.BusinessEntityID
  WHERE Employee.BusinessEntityID = @iEmpID
UNION ALL
SELECT Person.FirstName, Person.LastName,
       Employee.BusinessEntityID,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel,
       EmpHierarchy.FirstName AS MgrFirst,
       EmpHierarchy.LastName AS MgrLast,
       OrganizationNode AS OrgNode
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID =
         Person.BusinessEntityID
    JOIN EmpHierarchy
  ON Employee.OrganizationNode.GetAncestor(1) =
         EmpHierarchy.OrgNode
)

SELECT FirstName, LastName, EmpLevel,
       MgrFirst, MgrLast
    FROM EmpHierarchy
```

Setting up HierarchyIDs

Populating a HierarchyID field turns out to be simple. You can specify the plain text version and SQL Server will handle encoding it. You can also use the GetRoot and GetDescendant methods to populate the field.

GetDescendant is particularly useful for inserting a child of an existing record. You call the GetDescendant method of the parent record, passing parameters that indicate where the new record goes among the children of the parent. A complete explanation of the method is beyond the scope of this article, but Listing 11 shows code that creates a temporary table and adds a few records, and then shows the results. This code is included in this month's downloads as CreateHierarchy.SQL.

Listing 11. You can specify the hierarchyID value directly or use the GetRoot and GetDescendant methods.

```
CREATE TABLE #temp
  (orgHier HIERARCHYID, NodeName CHAR(20))

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( '/', 'Root')          )

DECLARE @Root HIERARCHYID,
        @curNode HIERARCHYID
SELECT @Root = hierarchyID::GetRoot()

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( @Root.GetDescendant(NULL, NULL),
          'First child'  )

SELECT @curNode = MAX(orgHier)
    FROM #temp
    WHERE orgHier.GetAncestor(1) = @Root

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( @curNode.GetDescendant(NULL, NULL),
          'First grandchild')

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( @Root.GetDescendant(@curNode, NULL),
          'Second child')

SELECT orgHier, orgHier.ToString(),
       NodeName
  FROM #temp

DROP TABLE #temp
```

You'll find a good tutorial on the HierarchyID type, including a discussion of the methods, at http://tinyurl.com/n6kk6jm.

What about VFP?

Obviously, VFP has no analogue of the HierarchyID data type. However, you can create your own. Marcia Akins describes an approach to doing so in her paper "Modeling Hierachies," available at http://tightlinecomputers.com/Downloads.htm; scroll down near the bottom of the page.

Of course, a home-grown version won't include the methods that SQL Server's HierarchyID type comes with. You'll have to write your own code to handle look-ups and insertions.

Summing up

While most hierarchies we encounter in modeling data involve different entities at each level, there are many that are self-referential. While you can work with such hierarchies in Visual FoxPro, SQL Server offers a much stronger set of tools for this kind of data.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer, *available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*