Go Beyond VFP's SQL with SQL Server

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

The subset of SQL in Visual FoxPro is useful for many tasks. But there's much more to SQL than what VFP supports. Those additions make it easy to do a number of tasks that are difficult in VFP.

In this session, we'll solve some common problems, using SQL elements that are supported by SQL Server, but not by VFP. Among the problems we'll explore are combining a set of values contained in multiple records into a delimited list in a single record, working with hierarchical data like corporate organization charts, finding the top N records for each group in a result, and including summary records in grouped data.

Introduction

When FoxPro 2.0 was released nearly 25 years ago, it included some SQL commands. I fell in love as soon as I started playing with them. Over the years, Visual FoxPro's SQL subset has grown, but there are still some tasks that are hard or impossible to do with SQL alone in VFP, but a lot easier in other SQL dialects. In this session, I'll take a look at some of these tasks, showing you how VFP requires a blend of SQL and Xbase code, but SQL Server allows them to be done with SQL code only.

You're unlikely to be choosing whether to store your data in VFP or in SQL Server based on which one makes these tasks easier. However, when you switch from working with VFP databases to working with SQL Server databases, it's easy to just keep doing things the way you have been. The goal of this session is to show you how you can code better in SQL Server by learning some new approaches.

The VFP examples in this session use the example Northwind database. Most of the SQL examples in this session use the example AdventureWorks 2008 database, which you can download from http://tinyurl.com/cp2fv8w. One group of examples uses the AdventureWorks 2005 database, because the 2008 version no longer includes the structure being discussed; you can download AdventureWorks 2005 from http://tinyurl.com/y943xr9.

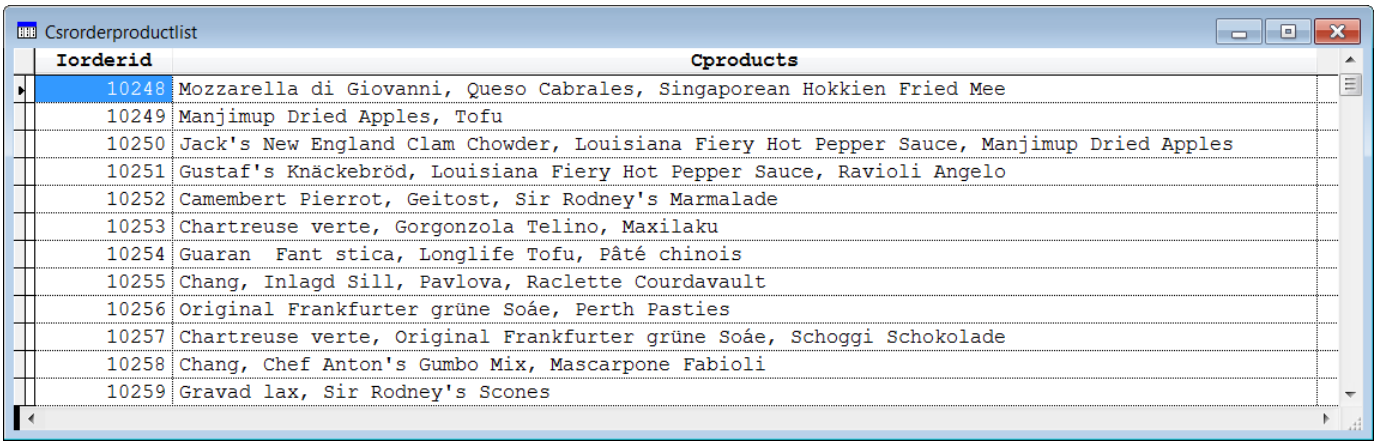
Consolidate data from a field into a list

One of the most common questions I see in online VFP forums is how to group data, consolidating the data from a particular field. If the consolidation you want is counting, summing, or averaging, the task is simple; just use GROUP BY with the corresponding aggregate function.

But if you want to create a comma-separated list of all the values or something like that, there's no SQL-only way to do it in VFP. SQL Server, however, provides not one, but two, ways.

The VFP way

Using the Northwind database that comes with VFP, suppose you want (say, for reporting purposes) to have a list of orders, with a comma-separated list of the products included in each order, something like what you see in **Figure 1**.


Csrorderproductlist

Iorderid	Cproducts
10248	Mozzarella di Giovanni, Queso Cabrales, Singaporean Hokkien Fried Mee
10249	Manjimup Dried Apples, Tofu
10250	Jack's New England Clam Chowder, Louisiana Fiery Hot Pepper Sauce, Manjimup Dried Apples
10251	Gustaf's Knäckebröd, Louisiana Fiery Hot Pepper Sauce, Ravioli Angelo
10252	Camembert Pierrot, Geitost, Sir Rodney's Marmalade
10253	Chartreuse verte, Gorgonzola Telino, Maxilaku
10254	Guaran Fant stica, Longlife Tofu, Pâté chinois
10255	Chang, Inlagd Sill, Pavlova, Raclette Courdavault
10256	Original Frankfurter grüne Soáe, Perth Pasties
10257	Chartreuse verte, Original Frankfurter grüne Soáe, Schoggi Schokolade
10258	Chang, Chef Anton's Gumbo Mix, Mascarpone Fabioli
10259	Gravad lax, Sir Rodney's Scones

Figure 1. This cursor includes each order from the Northwind database with a comma-separated list of the products ordered.

VFP's SQL commands offer no way to combine the products like that. Instead, you have to run a query to collect the raw data and then use a loop to combine the products for each order. **Listing 1** shows the code used to produce the cursor for the figure. (Like all the VFP examples in this paper, this one assumes you've already opened the Northwind database.)

Listing 1. To consolidate data into a comma-separated list in VFP requires a combination of SQL and Xbase code.

```
SELECT DISTINCT Orders.OrderID, Products.ProductName ;
  FROM Orders ;
    JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
    JOIN Products ;
      ON OrderDetails.ProductID = Products.ProductID ;
  ORDER BY Orders.OrderID, ProductName ;
  INTO CURSOR csrOrderProducts

LOCAL cProducts, cCurOrderID
CREATE CURSOR csrOrderProductList (iOrderID I, cProducts C(150))

SELECT csrOrderProducts
cCurOrderID = csrOrderProducts.OrderID
cProducts = ''

SCAN
  IF csrOrderProducts.OrderID <> m.cCurOrderID
    * Finished this order
    INSERT INTO csrOrderProductList ;
      VALUES (m.cCurOrderID, SUBSTR(m.cProducts, 3))
    cProducts = ''
    cCurOrderID = csrOrderProducts.OrderID
  ENDIF

  cProducts = m.cProducts + ', ' + ALLTRIM(csrOrderProducts.ProductName)
ENDSCAN
```

The query uses DISTINCT because we only want to include each product in the list once for each order. It also sorts the results by OrderID, which is necessary for the SCAN loop, and then by name within the order, so the result has the products in alphabetical order.

The SCAN loop builds up the list of products for a single order and then when we reach a new order, adds a record to the result cursor and clears the cProducts variable, so we can start over for the new order.

The code in **Listing 1** is included in the materials for this session as VFPProductsByOrder.PRG

The SQL way

SQL Server offers two ways to solve this problem. Each approach teaches something about elements of SQL Server that don't exist in VFP's SQL, so we'll look at both.

Using the AdventureWorks 2008 database, to get an example analogous to the VFP example, we can join the PurchaseOrderDetail table to the Product table to get a list of the products included in each purchase order, as in **Listing 2**.

Listing 2. This query, based on the AdventureWorks 2008 database, produces a list of products for each purchase order.

```
SELECT PurchaseOrderID, Name
 FROM Production.Product
  Inner Join Purchasing.PurchaseOrderDetail
   On Production.Product.ProductID = PurchaseOrderDetail.ProductID
 ORDER BY PurchaseOrderID
```

We'll use this query as a basis for getting one record per purchase order with the list of products comma-separated.

FOR XML

The first approach uses the FOR XML clause. In general, this clause allows you to convert SQL results to XML. There are four variations of FOR XML; three of them produce XML results and vary only in how much control you have over the format of the result. For example, if you add the clause FOR XML AUTO at the end of the query in **Listing 2**, you get results like those in **Listing 3**.

Listing 3. Adding FOR XML AUTO to the query in **Listing 2** produces this XML. (Only a few records are shown.)

```
<A PurchaseOrderID="1">
  <Production.Product Name="Adjustable Race" />
</A>
<A PurchaseOrderID="2">
  <Production.Product Name="Thin-Jam Hex Nut 9" />
  <Production.Product Name="Thin-Jam Hex Nut 10" />
</A>
<A PurchaseOrderID="3">
```

```
  <Production.Product Name="Seat Post" />
</A>
<A PurchaseOrderID="4">
  <Production.Product Name="Headset Ball Bearings" />
</A>
```

Using FOR XML RAW, instead, produces one element of type <row> for each record, with each field included as an attribute. **Listing 4** shows the first few records of the result.

Listing 4. FOR XML RAW produces simpler XML.

```
<row PurchaseOrderID="1" Name="Adjustable Race" />
<row PurchaseOrderID="2" Name="Thin-Jam Hex Nut 9" />
<row PurchaseOrderID="2" Name="Thin-Jam Hex Nut 10" />
<row PurchaseOrderID="3" Name="Seat Post" />
```

A third version, FOR XML EXPLICIT, gives you tremendous control over the format of the output, at the cost of writing a more complex query. The details are beyond the scope of this session, and the documentation indicates that you can do the same things using FOR XML PATH much more easily. However, if you're interested, see http://technet.microsoft.com/en-us/library/ms189068.aspx.

The fourth version of FOR XML, using the PATH keyword, provides what we need to consolidate the product data into a single record. FOR XML PATH treats columns as XPath expressions. XPath, which stands for XML Path language, lets you select items in an XML document. Again, the full details are beyond the scope of this article.

What you need to know to solve the problem of creating a comma-separated list is that if you specify FOR XML PATH(''), the expression you specify in the query is consolidated into a single list, rather than one record per value. For example, the query in **Listing 5** produces the results shown in **Listing 6**.

Listing 5. Use FOR XML PATH('') to combine data into a single string.

```
SELECT ', ' + Name
 FROM Production.Product
  Inner Join Purchasing.PurchaseOrderDetail As A
   On Production.Product.ProductID = A.ProductID
 WHERE A.PurchaseOrderID = 7
 ORDER BY Name
 FOR XML PATH('')
```

Listing 6. The query in **Listing 5** produces a single string.

```
, HL Crankarm, LL Crankarm, ML Crankarm
```

The query here assembles the list for a single purchase order, due to the WHERE clause. The ORDER BY clause makes sure the products are listed in alphabetical order.

The field list in this case must either be an expression, as in the example, or must include the clause: AS "Data()". Otherwise, you get XML rather than a simple list. Since you'll usually want some punctuation between items, this isn't a particularly onerous restriction.

However, the query in **Listing 5** doesn't deal with duplicate products in a single order. To demonstrate, specify 4008 as the purchase order ID to match rather than 7 (because order 4008 has a couple of duplicate products). When you do so, you get the result shown in **Listing 7**. (I've added line breaks to make it more readable; the actual result is one long string with no breaks. Note also that the product names include commas, so it might actually be better to separate the items with something else, perhaps semi-colons.)

Listing 7. The query in **Listing 5** doesn't remove duplicates.

```
, Classic Vest, L, Classic Vest, L, Classic Vest, M, Classic Vest, M,
Classic Vest, M, Classic Vest, S, Full-Finger Gloves, L, Full-Finger Gloves, M,
Full-Finger Gloves, S, Half-Finger Gloves, L, Half-Finger Gloves, M,
Half-Finger Gloves, S, Women's Mountain Shorts, L, Women's Mountain Shorts, M,
Women's Mountain Shorts, S
```

To remove the duplicates, we need to use a derived table within this query, as in **Listing 8**. The derived table extracts the list of distinct product names for the purchase order and then the main query can sort them. I use the derived table because using requires the field(s) listed in the ORDER BY clause to be included in the SELECT list; in this case, we're sorting by Name, but the SELECT list includes only the expression (', ' + Name). (You could in fact, do this without the derived table by using ", " + Name in the ORDER BY clause, but I think the derived table version is more readable.)

Listing 8. To have only distinct product names and be able to sort them requires a derived table.

```
SELECT ', ' + Name
 FROM (SELECT DISTINCT Name
   FROM Production.Product
    Inner Join Purchasing.PurchaseOrderDetail As A
    On Production.Product.ProductID = A.ProductID
   WHERE A.PurchaseOrderID = 4008) DistNames
   ORDER BY Name
 FOR XML PATH('')
```

Listing 9 shows the results of the query in **Listing 8**. As before, they've been reformatted for readability.

Listing 9. With the more complex query in **Listing 8**, the results don't include duplicates.

```
, Classic Vest, L, Classic Vest, M, Classic Vest, S, Full-Finger Gloves, L,
Full-Finger Gloves, M, Full-Finger Gloves, S, Half-Finger Gloves, L,
Half-Finger Gloves, M, Half-Finger Gloves, S, Women's Mountain Shorts, L,
Women's Mountain Shorts, M, Women's Mountain Shorts, S
```

The next issue is the leading comma in the result. To remove it, we use the STUFF() function , which is identical to the VFP STUFF() function. It replaces part of a string with

another string. In this case, we want to replace the first two characters with the empty string.

However, you don't put the STUFF() function quite where you might expect. It has to wrap the entire query that produces the list. **Listing 10** shows the query that produces the list without the leading comma. Note that the query inside STUFF() has to be wrapped with parentheses, just like a derived table. (The opening parenthesis is before the keyword SELECT, while the closing parenthesis follows the XML PATH('') clause. That's followed by the additional parameters for STUFF().) Here, though, the subquery isn't a derived table; it's a computed field.

Listing 10. To remove the leading comma on the list, we wrap the whole query with STUFF().

```
SELECT STUFF( (SELECT ', ' + Name
 FROM (SELECT DISTINCT Name
  FROM Production.Product
   Inner Join Purchasing.PurchaseOrderDetail As A
    On Production.Product.ProductID = A.ProductID
  WHERE A.PurchaseOrderID = 7) DistNames
 ORDER BY Name
 FOR XML PATH('')), 1, 2, '')
```

We now have all the pieces we need to produce results analogous to those in **Figure 1**. In the outer query, we simply need to include the purchase order's ID. **Listing 11** shows the query and **Figure 2** shows part of the result, as displayed in SQL Server Management Studio (SSMS).

Listing 11. Combining the query from **Listing 10** with code to include the purchase order number gives us the desired results.

```
SELECT PurchaseOrderID,
       STUFF((SELECT ', ' + Name
  FROM (SELECT DISTINCT Name
   FROM Production.Product
    Inner Join Purchasing.PurchaseOrderDetail As A
     On Production.Product.ProductID = A.ProductID
   WHERE Purchasing.PurchaseOrderDetail.PurchaseOrderID = A.PurchaseOrderID) DistName
  ORDER BY Name
  FOR XML PATH('')), 1, 2, '') OrderProducts
 FROM Purchasing.PurchaseOrderDetail
 GROUP BY PurchaseOrderID
 ORDER BY 1
```

	PurchaseOrder...	OrderProducts
1	1	Adjustable Race
2	2	Thin-Jam Hex Nut 10, Thin-Jam Hex Nut 9
3	3	Seat Post
4	4	Headset Ball Bearings
5	5	HL Road Rim
6	6	Touring Rim
7	7	HL Crankarm, LL Crankarm, ML Crankarm
8	8	External Lock Washer 3, External Lock Washer 4, ...
9	9	Thin-Jam Lock Nut 1, Thin-Jam Lock Nut 10, Thin-...
10	10	Chainring, Chainring Bolts, Chainring Nut
11	11	Lock Nut 16, Lock Nut 17, Lock Nut 5, Lock Nut 6
12	12	Touring Pedal
13	13	Chainring, Chainring Bolts, Chainring Nut

Figure 2. The query in **Listing 11** produces this result.

This solution is included in the materials for this session as RollupOrdersForXML.sql.

Using a function

The second approach to producing the desired list uses a function that consolidates the list of products. The downside of this approach is that you either have to have the function in the database, or create it on the fly and then drop it afterward. If you need the comma-separated list of products regularly, of course, there's really no reason not to add the function to the database.

The secret here is that the function accumulates the list in a variable, which it then returns to the main query. VFP doesn't allow you to store query results to a variable, but SQL Server does, using the syntax in **Listing 12**. You can even assign results to multiple variables in a single query. The variables must be declared before the query.

Listing 12. SQL Server lets you store a query result into a variable.

```
SELECT @VarName = <expression>
  FROM <rest of query>
```

To create the comma-separated list, the expression on the right-hand side of the equal sign references the variable on the left-hand side. The code in **Listing 13** shows how to do this for a single purchase order. To display the results in SSMS, add SELECT @Products at the end of the code block.

Listing 13. The ability to store a query result in a variable provides a way to accumulate the list of products for a single purchase order.

```
DECLARE @Products VARCHAR(1000)

SELECT @Products = COALESCE(@Products + ',', '') + Name
  FROM Production.Product
    Inner Join Purchasing.PurchaseOrderDetail As A
    On Production.Product.ProductID = A.ProductID
  WHERE A.PurchaseOrderID = 7
  ORDER BY Name
```

The COALESCE() function accepts a list of expressions and returns the first one with a non-null value. Since @Products is initially null (because it's not given an initial value), on the first record, COALESCE() chooses the empty string and the result doesn't have a leading comma.

As in the FOR XML PATH case, the query here doesn't remove duplicates. The solution is the same here; use a derived query to produce the list of distinct products before combining them. (In this case, you need the derived table; there's not a way to collect distinct product names without it.) **Listing 14** shows the code that produces a sorted list of distinct products for one purchase order.

Listing 14. To include each product only once in the list, we again use a derived query inside the query that assembles the comma-separated list.

```
DECLARE @Products VARCHAR(1000)

SELECT @Products = COALESCE(@Products + ',', '') + Name
  FROM (SELECT DISTINCT Name
   FROM Production.Product
    Inner Join Purchasing.PurchaseOrderDetail As A
      On Production.Product.ProductID = A.ProductID
   WHERE A.PurchaseOrderID = 4008) DistNames
  ORDER BY Name
```

We can use this code in a function to return the rolled-up list for a single purchase order. The main query calls the function for each purchase order. **Listing 15** shows the full code for this solution. Note that it creates the function, uses it and then drops it. As noted earlier, if you're going to do this regularly, just create the function once and keep it.

Listing 15. This solution to getting a comma-separated list of values from multiple records uses a function that rolls up the products for a single order.

```
CREATE FUNCTION ProductList (@POId INT)
  RETURNS VARCHAR(1000)
  AS
BEGIN
  DECLARE @Products VARCHAR(1000)

  SELECT @Products = COALESCE(@Products + ',', '') + Name
   FROM (SELECT DISTINCT Name
    FROM Production.Product
     Inner Join Purchasing.PurchaseOrderDetail As A
     On Production.Product.ProductID = A.ProductID
    WHERE A.PurchaseOrderID = @POId) DistNames
   ORDER BY Name

RETURN @Products
END
go

SELECT DISTINCT PurchaseOrderID, dbo.productList(PurchaseOrderID)
      AS ProductList
```

```
  FROM Purchasing.PurchaseOrderDetail
go

DROP FUNCTION dbo.ProductList
GO
```

Using DISTINCT in the main query ensures that we see each purchase order only once; otherwise, each would appear once for each included product.

This solution is included in the materials for this session as RollupOrdersByFunction.sql.

Which one?

Given two solutions, which one should you use? In my tests, the FOR XML PATH solution seems to be faster. However, the dataset in AdventureWorks is fairly small, so may not provide a good test bed. I recommend testing both solutions against your actual data.

If you find no significant difference in execution, then use the one that you find easier to read and comprehend, since you're likely to have to revisit it at some point.

Handle self-referential hierarchies

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)?

Well look 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 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 in "Using the HierarchyID type," later in this paper. 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.

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 16** (EmpPlusMgr.PRG in the materials for this session) 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 16. 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 17** (EmpPlusMgr.SQL in the materials for this session) 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 17. 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 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 we don't know how many levels we 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 we 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 18**. (This program is included in the materials for this session as EmpHierarchy.PRG.)

Listing 18. 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 manager field is empty. **Figure 3** shows the results when you pass 7 as the parameter.

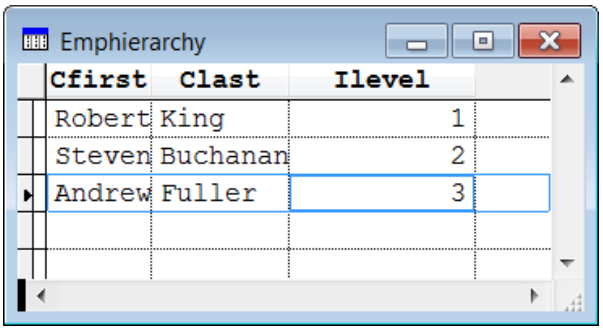


Figure 3. Running the query in **Listing 18**, passing 7 as the parameter, gives these results.

SQL Server provides a simpler solution, by using a Common 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 hierarchy.

Listing 19 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 19. 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 or the WHERE clause.

For example, the query in **Listing 8** could instead use a CTE as in **Listing 20**, which is included in the materials for this session as SimpleCTE.SQL.

Listing 20. You can usually replace a derived table with a CTE.

```
WITH DistNames (Name) AS
(SELECT DISTINCT Name
   FROM Production.Product
    Inner Join Purchasing.PurchaseOrderDetail As A
    On Production.Product.ProductID = A.ProductID
   WHERE A.PurchaseOrderID = 4008)

SELECT ', ' + Name
 FROM DistNames
   ORDER BY Name
 FOR XML PATH('')
```

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 21 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 the materials for this session as EmpHierarchyViaCTE.SQL.

Listing 21. To retrieve the management hierarchy for an employee in the SQL Server AdventureWorks 2005 database, use a Common 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, the second time, it's 3, and so forth.

Once the CTE is complete, the main query pulls the desired information from it. **Figure 4** shows the result of the query in **Listing 21**.

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

Figure 4. The query in **Listing 21** 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 we 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, we may (probably do) have multiple employees. We need not only to add each to the result, but to check who each of them manages. That means we need some way of keeping track of who we've checked and who we haven't.

We use two cursors. One (MgrHierarchy) holds the results, while the other (EmpsToProcess) holds the list of people to check. **Listing 22** shows the code; it's called MgrHierarchy.PRG in the materials for this session.

Listing 22. Putting together the list of people a specified person manages directly or indirectly is harder than climbing up the hierarchy.

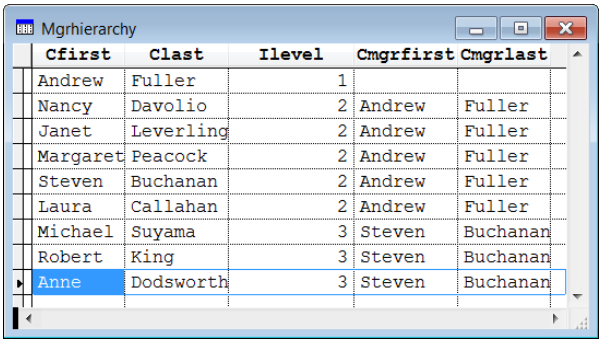
```
* Start with a single employee and determine
* all the people that employee manages,
* directly or indirectly.
LPARAMETERS iEmpID

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



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 5. 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, we 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 23** shows the query (MgrHierarchyViaCTE.SQL in the materials for this session), along with a variable declaration to indicate which employee we want to start with; **Figure 6** shows the results for this example.

Listing 23. 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 6. 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 maintaining and navigating straightforward. (The idea of a data type with methods is unusual. Think of the data type as essentially a class that you can use as a field.)

The SQL Server 2008 version of AdventureWorks uses the HierarchyID type to handle the management hierarchy (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, HierarchyIDs 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 24** (ShowHierarchyID.SQL in the materials for this session) shows a query to extract the name and hierarchy ID, both in encoded and plain text form, of the AdventureWorks employees; **Figure 7** shows a portion of the result.

Listing 24. 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 7. 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. A parameter indicates how many levels up the hierarchy 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 25** shows the query; it's included in the materials for this session as EmpHierarchyWithHierarchyID.SQL.

Listing 25. 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 field from Employee. **Listing 26** (MgrHierarchyWithHierarchyID.SQL in the materials for this session) shows the code.

Listing 26. 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 27** shows code that creates a temporary table and adds a few records, and then shows the results. This code is included in the materials for this session as CreateHierarchy.SQL.

Listing 27. 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.

Get the top N from each group

Both VFP and SQL Server include the TOP n clause, which allows you to include in the result only the first n records that match a query's filter conditions. But TOP n doesn't work when what you really want is the TOP n for each group in the query.

Suppose a company wants to know its top five salespeople for each year in some period. In VFP, you need to combine SQL with Xbase code or use a trick to get the desired results. With SQL Server, you can do it with a single query.

The VFP solution

Collecting the basic data you need to solve this problem is straightforward. **Listing 28** (EmployeeSalesByYear.PRG in the materials for this session) shows a query that provides each employee's sales by year; **Figure 8** shows part of the results.

Listing 28. Getting total sales by employee by year is easy in VFP.

```
SELECT FirstName, LastName, ;
       YEAR(OrderDate) as OrderYear, ;
       SUM(UnitPrice*Quantity) AS TotalSales ;
  FROM Employees ;
    JOIN Orders ;
      ON Employees.EmployeeID = Orders.EmployeeID ;
    JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY 1, 2, 3 ;
  ORDER BY OrderYear, TotalSales DESC ;
  INTO CURSOR csrEmployeeSalesByYear
```

Firstname	Lastname	Orderyear	Totalsales
Margaret	Peacock	1996	53114.8000
Nancy	Davolio	1996	38789.0000
Laura	Callahan	1996	23161.4000
Andrew	Fuller	1996	22834.7000
Steven	Buchanan	1996	21965.2000
Janet	Leverling	1996	19231.8000
Robert	King	1996	18104.8000
Michael	Suyama	1996	17731.1000
Anne	Dodsworth	1996	11365.7000
Margaret	Peacock	1997	139477.7000
Janet	Leverling	1997	111788.6100

Figure 8. The query in **Listing 28** produces the total sales for each employee by year.

However, when you want to keep only the top five for each year, you need to either combine SQL code with some Xbase code or use a trick that can result in a significant slowdown with large datasets.

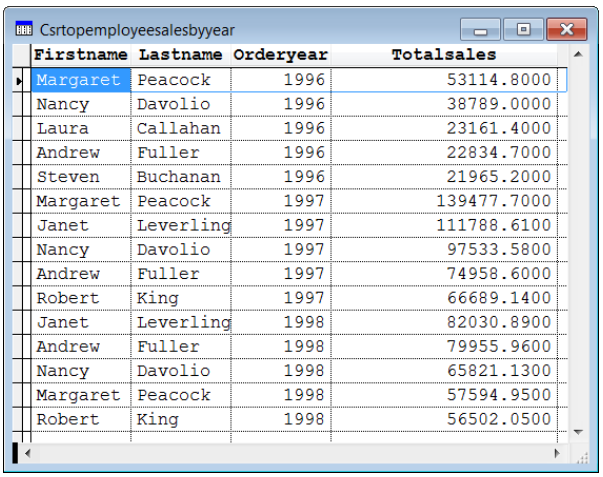
SQL plus Xbase

The mixed solution is easier to follow, so let's start with that one. The idea is to first select the raw data needed, in this case, the total sales by employee by year. Then we loop through on the grouping field, and select the top n (five, in this case) in each group and put them into a cursor. **Listing 29** (TopnEmployeeSalesByYear-Loop.PRG in the materials for this session) shows the code; **Figure 9** shows the result.

Listing 29. One way to find the top n in each group is to collect the data, then loop through it by group.

```
SELECT EmployeeID, ;
       YEAR(OrderDate) as OrderYear, ;
       SUM(UnitPrice*Quantity) AS TotalSales ;
  FROM Orders ;
    JOIN OrderDetails ;
    ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY 1, 2 ;
  INTO CURSOR csrEmpSalesByYear

CREATE CURSOR csrTopEmployeeSalesByYear ;
  (FirstName C(10), LastName C(20), ;
   OrderYear N(4), TotalSales Y)
```

```
SELECT distinct OrderYear ;
  FROM csrEmpSalesByYear ;
  INTO CURSOR csrYears

LOCAL nYear

SCAN
  nYear = csrYears.OrderYear

  INSERT INTO csrTopEmployeeSalesByYear ;
    SELECT TOP 5 ;
        FirstName, LastName, ;
        OrderYear, TotalSales ;
      FROM Employees ;
        JOIN csrEmpSalesByYear ;
          ON Employees.EmployeeID = csrEmpSalesByYear.EmployeeID ;
        WHERE csrEmpSalesByYear.OrderYear = m.nYear ;
      ORDER BY OrderYear, TotalSales DESC

ENDSCAN

USE IN csrYears
USE IN csrEmpSalesByYear
SELECT csrTopEmployeeSalesByYear
```


Csrtopemployeesalesbyyear

Firstname	Lastname	Orderyear	Totalsales
Margaret	Peacock	1996	53114.8000
Nancy	Davolio	1996	38789.0000
Laura	Callahan	1996	23161.4000
Andrew	Fuller	1996	22834.7000
Steven	Buchanan	1996	21965.2000
Margaret	Peacock	1997	139477.7000
Janet	Leverling	1997	111788.6100
Nancy	Davolio	1997	97533.5800
Andrew	Fuller	1997	74958.6000
Robert	King	1997	66689.1400
Janet	Leverling	1998	82030.8900
Andrew	Fuller	1998	79955.9600
Nancy	Davolio	1998	65821.1300
Margaret	Peacock	1998	57594.9500
Robert	King	1998	56502.0500

Figure 9. The query in **Listing 29** produces these results.

The first query is just a simpler version of **Listing 28**, omitting the Employees table and the ORDER BY clause; both of those will be used later. Next, we create a cursor to hold the final results. Then, we get a list of the years for which we have data. Finally, we loop through the cursor of years and, for each, grab the top five salespeople for that year, and put them into the result cursor, adding the employee's name and sorting as we insert.

You can actually consolidate this version a little by turning the first query into a derived table in the query inside the INSERT command. **Listing 30** (TopnEmployeeSalesByYear-Loop2.PRG in the materials for this session) shows the revised version. Note that you have

to get the list of years directly from the Orders table in this version. This version, of course, gives the same results.

Listing 30. The code in **Listing 29** can be reworked to use a derived table to compute the totals for each year.

```
CREATE CURSOR csrTopEmployeeSalesByYear ;
  (FirstName C(10), LastName C(20), ;
   OrderYear N(4), TotalSales Y)

SELECT distinct YEAR(OrderDate) AS OrderYear ;
  FROM Orders ;
  INTO CURSOR csrYears

LOCAL nYear

SCAN
  nYear = csrYears.OrderYear

  INSERT INTO csrTopEmployeeSalesByYear ;
    SELECT TOP 5 ;
      FirstName, LastName, ;
      OrderYear, TotalSales ;
     FROM Employees ;
      JOIN ( ;
       SELECT EmployeeID, ;
              YEAR(OrderDate) as OrderYear, ;
              SUM(UnitPrice * Quantity) ;
                AS TotalSales ;
        FROM Orders ;
         JOIN OrderDetails ;
          ON Orders.OrderID = OrderDetails.OrderID ;
         WHERE YEAR(OrderDate) = m.nYear ;
         GROUP BY 1, 2) csrEmpSalesByYear ;
     ON Employees.EmployeeID = csrEmpSalesByYear.EmployeeID ;
     ORDER BY OrderYear, TotalSales DESC

ENDSCAN

USE IN csrYears
SELECT csrTopEmployeeSalesByYear
```

SQL-only

The alternative VFP solution uses only SQL commands, but relies on a trick of sorts. Like the mixed solution, it starts with a query to collect the basic data needed. It then joins that data to itself in a way that results in multiple records for each employee/year combination and uses HAVING to keep only those that represent the top n records. Finally, it adds the employee name. **Listing 31** (TopNEmployeeSalesByYear-Trick.prg in the materials for this session) shows the code.

Listing 31. This solution uses only SQL, but requires a tricky join condition.

```
SELECT EmployeeID, ;
       YEAR(OrderDate) as OrderYear, ;
       SUM(UnitPrice * Quantity) ;
         AS TotalSales ;
  FROM Orders ;
    JOIN OrderDetails ;
    ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY 1, 2 ;
  INTO CURSOR csrEmpSalesByYear

SELECT FirstName, LastName, ;
       OrderYear, TotalSales ;
  FROM Employees ;
    JOIN ( ;
      SELECT ESBY1.EmployeeID, ;
             ESBY1.OrderYear, ;
             ESBY1.TotalSales ;
        FROM csrEmpSalesByYear ESBY1 ;
          JOIN csrEmpSalesByYear ESBY2 ;
            ON ESBY1.OrderYear = ESBY2.OrderYear ;
           AND ESBY1.TotalSales <= ESBY2.TotalSales ;
        GROUP BY 1, 2,3 ;
        HAVING COUNT(*) <= 5) csrTop5;
      ON Employees.EmployeeID = csrTop5.EmployeeID ;
  ORDER BY OrderYear, TotalSales DESC ;
  INTO CURSOR csrTopEmployeeSalesByYear
```

The first query here is just a variant of **Listing 28**. The key portion of this approach is the derived table in the second query, in particular, the join condition between the two instances of csrEmpSalesByYear, shown in **Listing 32**. Records are matched up first by having the same year and then by having sales in the second instance be the same or more than sales in the first instance. This results in a single record for the employee from that year with the highest sales total, two records for the employee with the second highest sales total and so on.

Listing 32. The key to this solution is the unorthodox join condition between two instances of the same table.

```
FROM csrEmpSalesByYear ESBY1 ;
  JOIN csrEmpSalesByYear ESBY2 ;
    ON ESBY1.OrderYear = ESBY2.OrderYear ;
   AND ESBY1.TotalSales <= ESBY2.TotalSales
```

The GROUP BY and HAVING clauses then combine all the records for a given employee and year, and keeps only those where the number of records in the intermediate result is five or fewer (that is, where the count of records in the group is five or less), providing the top five salespeople for each year.

To make more sense of this solution, first consider the query in **Listing 33** (included in the materials for this session as TopNEmployeeSalesByYearBeforeGrouping.prg). It assumes we've already run the query to create the EmpSalesByYear cursor. It shows the results

(plus a couple of additional fields) from the derived table in **Listing 31** before the GROUP BY is applied. In the partial results shown in **Figure 10**, you can see one record for employee 4 in 1996, two for employee 1, three for employee 8 and so forth. The added columns Emp2ID and Emp2Sales show which row in ESBY2 resulted in this result row. So, for employee 4 in 1996, the only row that met the conditions was the one for employee 4 in 1996. For employee 1 in 1996, both employee 4 and itself met the conditions of total sales the same or more than his or her own.

Listing 33. This query demonstrates the intermediate results for the derived table in **Listing 31**.

```
SELECT ESBY1.EmployeeID, ;
       ESBY1.OrderYear, ;
       ESBY1.TotalSales , ;
       ESBY2.EmployeeID AS Emp2ID, ;
       ESBY2.TotalSales AS Emp2Sales ;
  FROM EmpSalesByYear ESBY1 ;
    JOIN EmpSalesByYear ESBY2 ;
      ON ESBY1.OrderYear = ESBY2.OrderYear ;
     AND ESBY1.TotalSales <= ESBY2.TotalSales ;
  ORDER BY ESBY1.OrderYear, ESBY1.TotalSales DESC ;
  INTO CURSOR csrIntermediate
```

Employeeid	Orderyear	Totalsales	Emp2id	Emp2sales
4	1996	53114.8000	4	53114.8000
1	1996	38789.0000	1	38789.0000
1	1996	38789.0000	4	53114.8000
8	1996	23161.4000	1	38789.0000
8	1996	23161.4000	4	53114.8000
8	1996	23161.4000	8	23161.4000
2	1996	22834.7000	1	38789.0000
2	1996	22834.7000	2	22834.7000
2	1996	22834.7000	4	53114.8000
2	1996	22834.7000	8	23161.4000
5	1996	21965.2000	1	38789.0000
5	1996	21965.2000	2	22834.7000
5	1996	21965.2000	4	53114.8000
5	1996	21965.2000	5	21965.2000
5	1996	21965.2000	8	23161.4000

Figure 10. The query in **Listing 33** unfolds the data that's grouped in the derived table.

The problem with this approach to the problem is that, as the size of the original data increases, it can get bogged down. So while this solution has a certain elegance, in the long run, a SQL plus Xbase solution is probably a better choice.

By the way, this example (the one in **Listing 31**) shows where CTEs (common table expressions, explained earlier in this paper) would be useful in VFP's SQL. We can't easily combine the two queries into one because the second query uses two instances of the EmpSalesByYear. If VFP supported CTEs, we could make the query that creates EmpSalesByYear into a CTE, and then use it twice in the main query.

The SQL Server solution

Solving the top n by group problem in SQL Server uses a couple of CTEs, but also uses another construct that's not available in VFP's version of SQL.

The OVER clause lets you apply a function to all or part of a result set; it's used in the field list. There are several variations, but the basic structure is shown in **Listing 34**.

Listing 34. The OVER clause lets you apply a function to all or some of the records in a query.

```
<function> OVER (<grouping and/or ordering>)
```

OVER lets you rank records, as well as applying aggregates to individual items in the field list. In SQL Server 2012 and later, OVER has additional features that let you compute complicated aggregates such as running totals and moving averages.

For the top n by group problem, we want to rank records within a group and then keep the top n. To do that, we can use the ROW_NUMBER() function, which , as its name suggests, returns the row number of a record within a group (or within the entire result set, if no grouping is specified).

For example, **Listing 35** (included in the materials for this session as EmployeeOrderNumber.sql) shows a query that lists AdventureWorks (2008) employees in the order they were hired, giving each an "employee order number." Here, the data is ordered by HireDate and then ROW_NUMBER() is applied to provide the position of each record. **Figure 11** shows partial results.

Listing 35. Using ROW_NUMBER() with OVER lets you give records a rank.

```
SELECT FirstName, LastName, HireDate,
  ROW_NUMBER() OVER (ORDER BY HireDate)
    AS EmployeeOrderNumber
  FROM HumanResources.Employee
    JOIN Person.Person
      ON Employee.BusinessEntityID = Person.BusinessEntityID
```

FirstName	LastName	HireDate	EmployeeOrderNum...
Guy	Gilbert	2000-07-31	1
Kevin	Brown	2001-02-26	2
Roberto	Tamburello	2001-12-12	3
Rob	Walters	2002-01-05	4
Thierry	D'Hers	2002-01-11	5
David	Bradley	2002-01-20	6
JoLynn	Dobney	2002-01-26	7
Ruth	Ellerbrock	2002-02-06	8
Gail	Erickson	2002-02-06	9
Barry	Johnson	2002-02-07	10
Jossef	Goldberg	2002-02-24	11
Terri	Duffy	2002-03-03	12
Sidney	Higa	2002-03-05	13
Taylor	Maxwell	2002-03-11	14

Figure 11. The query in **Listing 35** applies a rank to each employee by hire date.

But look at Ruth Ellerbock and Gail Erickson; they have the same hire date, but different values for EmployeeOrderNumber. Sometimes, that's what you want, but sometimes, you want such records to have the same value.

The ROW_NUMBER() funtion doesn't know anything about ties. However, the RANK() function is aware of ties and assigns them the same value, then skips the appropriate number of values. **Listing 36** (EmployeeRank.SQL in the materials for this session) shows the same query using RANK() instead of ROW_NUMBER(); **Figure 12** shows the first few records. This time, you can see that Ellerbock and Erickson have the same rank, 8, while Barry Johnson, who immediately follows them, still has a rank of 10.

Listing 36. The RANK() function is aware of ties, assigning them the same value.

```
SELECT FirstName, LastName, HireDate,
  RANK() OVER (ORDER BY HireDate)
    AS EmployeeOrderNumber
  FROM HumanResources.Employee
    JOIN Person.Person
      ON Employee.BusinessEntityID = Person.BusinessEntityID
```

FirstName	LastName	HireDate	EmployeeRank
Guy	Gilbert	2000-07-31	1
Kevin	Brown	2001-02-26	2
Roberto	Tamburello	2001-12-12	3
Rob	Walters	2002-01-05	4
Thierry	D'Hers	2002-01-11	5
David	Bradley	2002-01-20	6
JoLynn	Dobney	2002-01-26	7
Ruth	Ellerbrock	2002-02-06	8
Gail	Erickson	2002-02-06	8
Barry	Johnson	2002-02-07	10
Jossef	Goldberg	2002-02-24	11
Terri	Duffy	2002-03-03	12
Sidney	Higa	2002-03-05	13
Taylor	Maxwell	2002-03-11	14

Figure 12. Using RANK() assigns the same EmployeeOrderNumber to records with the same hire date.

You can't say that either ROW_NUMBER() or RANK() is right; which one you want depends on the situation. In fact, there's a third related function, DENSE_RANK() that behaves like RANK(), giving ties the same value, but then continues numbering in order. That is, if we used DENSE_RANK() in this example, Barry Johnson would have a rank of 9, rather than 10.

Partitioning with OVER

In addition to specifying ordering, OVER also allows us to divide the data into groups before applying the function, using the PARTITION BY clause. The query in **Listing 37** (included in the materials for this session as EmployeeRankByDept.sql) assigns employee ranks within each department rather than for the company as a whole by using both PARTITION BY and ORDER BY. **Figure 13** shows partial results; note that the numbering begins again for each department and that ties are assigned the same rank.

Listing 37. Combining PARTITION BY and ORDER BY in the OVER clause lets you apply ranks within a group.

```
SELECT FirstName, LastName, StartDate,
       Department.Name,
       RANK() OVER
        (PARTITION BY Department.DepartmentID
         ORDER BY StartDate)
        AS EmployeeRank
FROM HumanResources.Employee
 JOIN HumanResources.EmployeeDepartmentHistory
   ON Employee.BusinessEntityID = EmployeeDepartmentHistory.BusinessEntityID
 JOIN HumanResources.Department
   ON EmployeeDepartmentHistory.DepartmentID = Department.DepartmentID
 JOIN Person.Person
   ON Employee.BusinessEntityID = Person.BusinessEntityID
 WHERE EndDate IS null
```

FirstName	LastName	StartDate	Name	EmployeeRank
Roberto	Tamburello	2001-12-12	Engineering	1
Gail	Erickson	2002-02-06	Engineering	2
Jossef	Goldberg	2002-02-24	Engineering	3
Terri	Duffy	2002-03-03	Engineering	4
Michael	Sullivan	2005-01-30	Engineering	5
Sharon	Salavaria	2005-02-18	Engineering	6
Thierry	D'Hers	2002-01-11	Tool Design	1
Rob	Walters	2004-07-01	Tool Design	2
Ovidiu	Cracium	2005-01-05	Tool Design	3
Janice	Galvin	2005-01-23	Tool Design	4
Stephen	Jiang	2005-02-04	Sales	1
Brian	Welcker	2005-03-18	Sales	2
Michael	Blythe	2005-07-01	Sales	3
Linda	Mitchell	2005-07-01	Sales	3
Jillian	Carson	2005-07-01	Sales	3

Figure 13. Here, employees are numbered within their current department, based on when they started in that department.

This example should provide a hint as to how we'll solve the top n by group problem, since we now have a way to number things by group. All we need to do is filter so we only keep those whose rank within the group is in the range of interest. However, it's not possible to filter on the computed field EmployeeOrderNumber in the same query. Instead, we turn that query into a CTE and filter in the main query, as in **Listing 38** (LongestStandingEmployeesByDept.sql in the materials for this session).

Listing 38. Once we have the rank for an item within its group, we just need to filter to get the top n items by group.

```
WITH EmpRanksByDepartment AS
(SELECT FirstName, LastName, StartDate,
        Department.Name AS Department,
        RANK() OVER
         (PARTITION BY Department.DepartmentID
          ORDER BY StartDate)
         AS EmployeeRank
 FROM HumanResources.Employee
 JOIN HumanResources.EmployeeDepartmentHistory
   ON Employee.BusinessEntityID = EmployeeDepartmentHistory.BusinessEntityID
 JOIN HumanResources.Department
   ON EmployeeDepartmentHistory.DepartmentID = Department.DepartmentID
 JOIN Person.Person
   ON Employee.BusinessEntityID = Person.BusinessEntityID
 WHERE EndDate IS NULL)

 SELECT FirstName, LastName, StartDate, Department
  FROM EmpRanksByDepartment
    WHERE EmployeeRank <= 3
    ORDER BY Department, StartDate
```

Figure 14 shows part of the result. Note that there are many more than three records for the Sales department because a whole group of people started on the same day. If you really want only three per department and don't care which records you omit from a last-place tie, use RECORD_NUMBER() instead of RANK().

Diane	Margheim	2003-01-30	Research and Developm...
Gigi	Matthew	2003-02-17	Research and Developm...
Dylan	Miller	2003-03-12	Research and Developm...
Stephen	Jiang	2005-02-04	Sales
Brian	Welcker	2005-03-18	Sales
Michael	Blythe	2005-07-01	Sales
Linda	Mitchell	2005-07-01	Sales
Jillian	Carson	2005-07-01	Sales
Garrett	Vargas	2005-07-01	Sales
Tsvi	Reiter	2005-07-01	Sales
Pamela	Ansman-Wolfe	2005-07-01	Sales
Shu	Ito	2005-07-01	Sales
José	Saraiva	2005-07-01	Sales
David	Campbell	2005-07-01	Sales
Susan	Eaton	2003-01-08	Shipping and Receiving

Figure 14. The query in **Listing 38** provides the three longest-standing employees in each department. When there are ties, it may produce more than three results.

Applying the same principle to finding the top five salespeople by year at AdventureWorks (to match our VFP example) is a little more complicated because we have to compute sales totals first. To make that work, we first use a CTE to compute those totals and then a second CTE based on that result to add the ranks. (Note the comma between the two CTEs.) **Listing 39** (TopSalesPeopleByYear.sql in the materials for this session) shows the complete query.

Listing 39. Finding the top five salepeople by year requires cascading CTEs, plus the OVER clause.

```
WITH TotalSalesBySalesPerson AS
(SELECT BusinessEntityID,
        YEAR(OrderDate) AS nYear,
        SUM(SubTotal) AS TotalSales
 FROM Sales.SalesPerson
  JOIN Sales.SalesOrderHeader
    ON SalesPerson.BusinessEntityID = SalesOrderHeader.SalesPersonID
GROUP BY BusinessEntityID, YEAR(OrderDate)),

RankSalesPerson AS
(SELECT BusinessEntityID, nYear, TotalSales,
  RANK() OVER
   (PARTITION BY nYear
    ORDER BY TotalSales DESC) AS nRank
  FROM TotalSalesBySalesPerson)

SELECT FirstName, LastName, nYear, TotalSales
  FROM RankSalesPerson
    JOIN Person.Person
      ON RankSalesPerson.BusinessEntityID = Person.BusinessEntityID
  WHERE nRank <= 5
```

The first CTE, TotalSalesBySalesPerson, contains the ID for the salesperson, the year and that person‘s total sales for the year. The second CTE, RankSalesPerson, adds rank within the group to the data from TotalSalesByPerson. Finally, the main query keeps only the top five in each and adds the actual name of the person. **Figure 15** shows partial results.

FirstName	LastName	nYear	TotalSales
Tsvi	Reiter	2005	1380707.4422
Jillian	Carson	2005	1247434.4374
Linda	Mitchell	2005	1143819.6543
José	Saraiva	2005	1038949.5399
Shu	Ito	2005	887498.8258
Jillian	Carson	2006	3803368.3941
Linda	Mitchell	2006	3234995.6953
Michael	Blythe	2006	3077197.9242
Jae	Pak	2006	2522835.9368
Tsvi	Reiter	2006	2478985.1202
Jae	Pak	2007	4172459.4445
Linda	Mitchell	2007	4102250.1622

Figure 15. These partial results show the top five salespeople by year.

It's worth noting the very cool feature demonstrated by this query. Not only can a query have multiple CTEs, but CTEs later in the list can be based on previous CTEs. So RankSalesPerson uses TotalSalesBySalesPerson in its FROM list.

The OVER clause has other uses, such as helping to de-dupe a list. In SQL 2012 and later, it's even more useful, with the ability to apply the function to a group of records based not only on an expression, but based on position within a group.

Summarize aggregated data

As earlier sections of this paper show, SQL SELECT's GROUP BY clause makes it easy to aggregate data in a query. Just include the fields that specify the groups and some fields using the aggregate functions (COUNT, SUM, AVG, MIN, MAX in VFP; SQL Server has those and a few more).

For example, the query in **Listing 40** (TotalsByCountryCity.PRG in the materials for this session) fills a cursor with sales for each city for each month; **Figure 16** shows partial results.

Listing 40. This query computes total sales for each combination of country, city, year and month.

```
SELECT Country, City, ;
       YEAR(OrderDate) AS OrderYear, MONTH(OrderDate) AS OrderMonth, ;
       SUM(Quantity * OrderDetails.UnitPrice) AS nTotal ;
       AVG(Quantity * OrderDetails.UnitPrice) AS nAvg, ;
       COUNT(*) AS nCount ;
  FROM Customers ;
    JOIN Orders ;
      ON Customers.CustomerID = Orders.CustomerID ;
    JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY OrderYear, OrderMonth, ;
           Country, City ;
  ORDER BY Country, City, ;
           OrderYear, OrderMonth ;
```

```
  INTO CURSOR csrCtyTotals
```

Country	City	Orderyear	Ordermonth	Ntotal	Navg	Ncount
Argentina	Buenos Aires	1997	1	319.2000	159.6000	2
Argentina	Buenos Aires	1997	2	443.4000	221.7000	2
Argentina	Buenos Aires	1997	4	225.5000	75.1667	3
Argentina	Buenos Aires	1997	5	110.0000	110.0000	1
Argentina	Buenos Aires	1997	10	706.0000	235.3333	3
Argentina	Buenos Aires	1997	12	12.5000	12.5000	1
Argentina	Buenos Aires	1998	1	1409.0000	352.2500	4
Argentina	Buenos Aires	1998	2	866.7000	173.3400	5
Argentina	Buenos Aires	1998	3	3645.8000	405.0889	9
Argentina	Buenos Aires	1998	4	381.0000	95.2500	4
Austria	Graz	1996	7	4483.4000	640.4857	7
Austria	Graz	1996	11	7511.8000	938.9750	8
Austria	Graz	1996	12	5175.2000	575.0222	9
Austria	Graz	1997	1	9515.4000	1189.4250	8

Figure 16. The query in **Listing 40** computes the total sales for each city in each month.

You can do an analogous query using the SQL Server AdventureWorks 2008 database, though it involves a lot more tables because the AdventureWorks database covers a wider range of data than just sales. **Listing 41** (SalesByCountryCity.SQL in the materials for this session) shows the corresponding SQL Server query.

Listing 41. Aggregating the data with SQL Server's AdventureWorks 2008 database is more verbose, but contains the same elements.

```
SELECT Person.CountryRegion.Name AS Country,
       Person.Address.City,
       YEAR(OrderDate) AS nYear,
       MONTH(OrderDate) AS nMonth,
       SUM(SubTotal) AS TotalSales
       AVG(SubTotal) AS AvgSale,
       COUNT(*) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
  GROUP BY CountryRegion.Name, Address.City,
           YEAR(OrderDate), MONTH(OrderDate))
```

The rules for grouping are pretty simple. The field list contains two types of fields, those to group on, and those that are being aggregated. In the VFP example, the fields to group on are Country, City, OrderYear and OrderMonth, and the aggregated fields are nTotal, nAvg and nCount. The SQL Server query has the same list, but some of the field names are different. (Before VFP 8, you could include fields in the list that were neither grouped or nor aggregated, but doing so could give you misleading results. This article on my website explains the problem in detail: http://tinyurl.com/leydyqw.)

Computing group totals

What the basic query doesn't give you, though, is aggregation (that is, summaries) at any level except the one you specify. That is, while you get the total, average and count for a specific city in a specific month, you don't get them for that city for the whole year, or for that month for a whole country, and so on. **Figure 17** shows what we're looking for. At the end of each year, a new record shows the total, average and count for that year. At the end of each city, another record shows the city's total, average and count and at the end of each country, yet another record has country-wide results.

Country	City	Orderyear	Ordermonth	Ntotal	Navg	Ncount
Austria	.NULL.	.NULL.	.NULL.	139496.6300	877.3373	159
Belgium	Bruxelles	1997	5	946.0000	315.3333	3
Belgium	Bruxelles	1997	8	1434.0000	717.0000	2
Belgium	Bruxelles	1997	12	3304.0000	1101.3333	3
Belgium	Bruxelles	1997	.NULL.	5684.0000	710.5000	8
Belgium	Bruxelles	1998	2	2950.5000	983.5000	3
Belgium	Bruxelles	1998	3	1500.7000	375.1750	4
Belgium	Bruxelles	1998	4	295.3800	147.6900	2
Belgium	Bruxelles	1998	.NULL.	4746.5800	527.3978	9
Belgium	Bruxelles	.NULL.	.NULL.	10430.5800	613.5635	17
Belgium	Charleroi	1996	7	3730.0000	1243.3333	3
Belgium	Charleroi	1996	9	2708.8000	902.9333	3
Belgium	Charleroi	1996	.NULL.	6438.8000	1073.1333	6
Belgium	Charleroi	1997	2	3891.0000	778.2000	5
Belgium	Charleroi	1997	3	2484.1000	496.8200	5
Belgium	Charleroi	1997	12	28.0000	28.0000	1
Belgium	Charleroi	1997	.NULL.	6403.1000	582.1000	11
Belgium	Charleroi	1998	1	5693.0000	813.2857	7
Belgium	Charleroi	1998	2	1209.0000	302.2500	4
Belgium	Charleroi	1998	3	2455.0000	613.7500	4
Belgium	Charleroi	1998	4	2505.5000	357.9286	7
Belgium	Charleroi	1998	.NULL.	11862.5000	539.2045	22
Belgium	Charleroi	.NULL.	.NULL.	24704.4000	633.4462	39
Belgium	.NULL.	.NULL.	.NULL.	35134.9800	163.4185	215

Figure 17. It can be useful to have group totals in the same cursor as the original data.

In VFP, there are three ways to get that data. One is to create a report and use totals and report variables to compute and report that data, but of course, then you only have the data as output, not in a VFP cursor.

The second choice is to use Xbase code to compute them based on the initial cursor. **Listing 42** (WithGroupTotalsXbase.PRG in the materials for this session) shows how to do this; it assumes you've already run the query in **Listing 40**. It keeps running totals and counts for

each level: year, city, country and overall. Then, when one of those changes, it inserts the appropriate record.

Listing 42. You can add subgroup aggregates by looping through the cursor.

```
LOCAL nYearTotal, nCityTotal, nCountryTotal, nGrandTotal
LOCAL nYearCnt, nCityCnt, nCountryCount, nGrandCount
LOCAL nCurYear, cCurCity, cCurCountry

* Create a new cursor to hold the results
SELECT * ;
  FROM csrCtyTotals ;
  WHERE .F. ;
  INTO CURSOR csrWithGroupTotals READWRITE

SELECT csrCtyTotals
STORE 0 TO nYearTotal, nCityTotal, nCountryTotal, nGrandTotal
STORE 0 TO nYearCount, nCityCount, nCountryCount, nGrandCount
nCurYear = csrCtyTotals.OrderYear
cCurCity = csrCtyTotals.City
cCurCountry = csrCtyTotals.Country

SCAN
  * First check for end of year,
  * but could be same year and change of city
  * or country.
  IF csrCtyTotals.OrderYear <> m.nCurYear OR ;
     NOT (csrCtyTotals.City == m.cCurCity) OR;
     NOT (csrCtyTotals.Country == m.cCurCountry)
    INSERT INTO csrWithGroupTotals ;
      VALUES (m.cCurCountry, m.cCurCity, ;
              m.nCurYear, .null., ;
              m.nYearTotal, ;
              m.nYearTotal/m.nYearCount, ;
              m.nYearCount)
    m.nCurYear = csrCtyTotals.OrderYear
    STORE 0 TO m.nYearTotal, m.nYearCount

    * Now check for change of city
    IF NOT (csrCtyTotals.City == m.cCurCity) ;
       OR NOT (csrCtyTotals.Country == m.cCurCountry)
        INSERT INTO csrWithGroupTotals ;
            VALUES (m.cCurCountry, ;
                    m.cCurCity, ;
                    .null., .null., ;
                    m.nCityTotal, ;
                    m.nCityTotal/m.nCityCount, ;
                    m.nCityCount)
          m.cCurCity = csrCtyTotals.City
          STORE 0 TO m.nCityTotal, m.nCityCount

          * Now check for change of country
          IF NOT (csrCtyTotals.Country == m.cCurCountry)
            INSERT INTO csrWithGroupTotals ;
```

```
              VALUES (m.cCurCountry, .null., ,
                     .null., .null., ;
                      m.nCountryTotal, ;
                      m.nCountryTotal/m.nCountryCount, ;
                      m.nCountryCount)
            m.cCurCountry = csrCtyTotals.Country
            STORE 0 TO m.nCountryTotal, m.CountryCount
          ENDIF
      ENDIF
  ENDIF

  * Now handle current record
  INSERT INTO csrWithGroupTotals ;
    VALUES (csrCtyTotals.Country, ;
            csrCtyTotals.City, ;
            csrCtyTotals.OrderYear, ;
            csrCtyTotals.OrderMonth, ;
            csrCtyTotals.nTotal, ;
            csrCtyTotals.nAvg, ;
            csrCtyTotals.nCount)
  nYearTotal = m.nYearTotal + csrCtyTotals.nTotal
  nYearCount = m.nYearCount + csrCtyTotals.nCount
  nCityTotal = m.nCityTotal + csrCtyTotals.nTotal
  nCityCount = m.nCityCount + csrCtyTotals.nCount
  nCountryTotal = m.nCountryTotal + csrCtyTotals.nTotal
  nCountryCount = m.nCountryCount + csrCtyTotals.nCount
  nGrandTotal = m.nGrandTotal + csrCtyTotals.nTotal
  nGrandCount = m.nGrandCount + csrCtyTotals.nCount

ENDSCAN

* Now insert grand totals
INSERT INTO csrWithGroupTotals ;
  VALUES (.null., .null., .null., .null., ;
          m.nGrandTotal, ;
          m.nGrandTotal/m.nGrandCount, ;
          m.nGrandCount)
```

The third choice is to do a series of queries, each grouping on different levels and then consolidate the results. **Listing 43** shows this version of the code; as in the previous example, it assumes you've already run the query that creates csrCtyTotals. This code creates a cursor with each city's annual totals, one with each city's overall totals, one with each country's overall totals, and one containing the grand total. Then it uses UNION to combine all the results into a single cursor. It's included in the materials for this session as WithGroupTotalsSQL.PRG.

Listing 43. You can add the yearly, city-wide and country-wide totals using SQL, as well.

```
* Now year totals by city
SELECT Country, City, OrderYear, ;
       999 as OrderMonth, ;
       SUM(nTotal) AS nTotal, ;
       SUM(nTotal)/SUM(nCount) AS nAvg, ;
```

```
       SUM(nCount) AS nCount ;
  FROM csrCtyTotals ;
  GROUP BY Country, City, OrderYear ;
  INTO CURSOR csrYearTotals

* Now city totals
SELECT Country, City, ;
       99999 AS OrderYear, ;
       999 as OrderMonth, ;
       SUM(nTotal) AS nTotal, ;
       SUM(nTotal)/SUM(nCount) AS nAvg, ;
       SUM(nCount) AS nCount ;
  FROM csrCtyTotals ;
  GROUP BY Country, City ;
  INTO CURSOR csrCityTotals

* Now country totals
SELECT Country, ;
       REPLICATE('Z', 15) AS City, ;
       99999 AS OrderYear, ;
       999 as OrderMonth, ;
       SUM(nTotal) AS nTotal, ;
       SUM(nTotal)/SUM(nCount) AS nAvg, ;
       SUM(nCount) AS nCount ;
  FROM csrCtyTotals ;
  GROUP BY Country ;
  INTO CURSOR csrCountryTotals

* Now grand total
SELECT REPLICATE('Z', 15) AS Country, ;
       REPLICATE('Z', 15) AS City, ;
       99999 AS OrderYear, ;
       999 as OrderMonth, ;
       SUM(nTotal) AS nTotal, ;
       SUM(nTotal)/SUM(nCount) AS nAvg, ;
       SUM(nCount) AS nCount ;
  FROM csrCtyTotals ;
  INTO CURSOR csrGrandTotal

* Create one cursor
SELECT * ;
  FROM csrCtyTotals ;
UNION ALL ;
SELECT * ;
  FROM csrYearTotals ;
UNION ALL ;
SELECT * ;
  FROM csrCityTotals ;
UNION ALL ;
SELECT * ;
  FROM csrCountryTotals ;
UNION ALL ;
SELECT * ;
  FROM csrGrandTotal ;
  ORDER BY Country, City, ;
```

```
          OrderYear, OrderMonth ;
  INTO CURSOR csrWithGroupTotals READWRITE

UPDATE csrWithGroupTotals ;
  SET OrderMonth = .null. ;
  WHERE OrderMonth = 999

UPDATE csrWithGroupTotals ;
  SET OrderYear = .null. ;
  WHERE OrderYear = 99999

UPDATE csrWithGroupTotals ;
  SET City = .null. ;
  WHERE City = REPLICATE('Z', 15)

UPDATE csrWithGroupTotals ;
  SET Country = .null. ;
  WHERE Country = REPLICATE('Z', 15)
```

There's one trick in this code. If we put null into the fields that are irrelevant for a given total, when we sort the result, the totals appear above rather than below the records they represent. Instead, we put an impossible value that sorts to the bottom initially, then change it to null after ordering the data.

Introducing ROLLUP

Of course, the reason for showing all this code is that SQL Server makes it much easier. The ROLLUP clause lets you compute these summaries as part of the original query.

ROLLUP appears in the GROUP BY clause, looking like a function around the fields you apply it to. **Listing 44** shows the SQL Server equivalent of **Listing 42** and **Listing 43**; the code is included in the materials for this session as SalesByCountryCityRollup.SQL. **Figure 18** shows partial results.

Listing 44. SQL Server's ROLLUP clause computes the subgroup aggregates as part of the query.

```
SELECT Person.CountryRegion.Name AS Country, Person.Address.City,
       YEAR(OrderDate) AS nYear, MONTH(OrderDate) AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
```

```
    ON Customer.CustomerID = SalesOrderHeader.CustomerID
  JOIN Sales.SalesOrderDetail
    ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
GROUP BY ROLLUP(CountryRegion.Name, Address.City,
                YEAR(OrderDate), MONTH(OrderDate))
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
Australia	Caloundra	2007	NULL	203478.1...	1225.7718	166
Australia	Caloundra	2008	1	19240.68	962.034	20
Australia	Caloundra	2008	2	31246.21	801.1848	39
Australia	Caloundra	2008	3	50989.52	1456.8434	35
Australia	Caloundra	2008	4	50096.48	1192.7733	42
Australia	Caloundra	2008	5	34395.28	1433.1366	24
Australia	Caloundra	2008	6	25243.12	901.54	28
Australia	Caloundra	2008	7	494.28	61.785	8
Australia	Caloundra	2008	NULL	211705.57	1080.1304	196
Australia	Caloundra	NU...	NULL	527130.8...	1301.5576	405
Australia	Cloverdale	2005	8	3399.99	3399.99	1
Australia	Cloverdale	2005	10	3374.99	3374.99	1
Australia	Cloverdale	2005	11	3578.27	3578.27	1
Australia	Cloverdale	2005	12	6953.26	3476.63	2
Australia	Cloverdale	2005	NULL	17306.51	3461.302	5

Figure 18. In SQL Server, it's easy to compute aggregates for subgroups.

The order of the fields in the ROLLUP clause matters. The last one listed is summarized first. In **Figure 18**, you can see that the first level of summary is the whole year for a given city and country, because the month column is listed last. If you change the order in the ROLLUP clause to put the city last, as in **Listing 45**, the first summary level is a single month (and year), across all cities in a country; **Figure 19** shows partial results.

Listing 45. The order of the fields in the ROLLUP clause matters. Changing the order changes what summaries you get.

```
GROUP BY ROLLUP(CountryRegion.Name, YEAR(OrderDate),
                MONTH(OrderDate), Address.City)
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
Australia	Townsville	2008	7	66.11	13.222	5
Australia	Warrnambool	2008	7	660.20	73.3555	9
Australia	Wollongong	2008	7	367.84	91.96	4
Australia	NULL	2008	7	23555.63	63.1518	373
Australia	NULL	2008	NULL	6786807.30	1004.4113	6757
Australia	NULL	NU...	NULL	16322659...	1223.1292	13345
Canada	Haney	2005	7	3578.27	3578.27	1
Canada	Metchosin	2005	7	3578.27	3578.27	1
Canada	N. Vancouver	2005	7	3578.27	3578.27	1
Canada	Newton	2005	7	3374.99	3374.99	1
Canada	Royal Oak	2005	7	3578.27	3578.27	1
Canada	Shawnee	2005	7	4277.3682	2138.6841	2
Canada	NULL	2005	7	21965.4382	3137.9197	7
Canada	Burnaby	2005	8	3578.27	3578.27	1

Figure 19. When you change the order of fields in the ROLLUP clause, you get a different set of summaries.

The ROLLUP clause doesn't have to surround all the fields in the GROUP BY, only the ones for which you want summaries. So, if you don't need a grand total in the previous example,

you can put CountryRegion.Name before the ROLLUP clause, as in **Listing 46**. Similarly, if you want summaries only for each city and year, put both CountryRegion.Name and Address.City before the ROLLUP clause. You can also put fields after the ROLLUP clause, but in my testing, the results aren't terribly useful.

Listing 46. Not all fields have to be included in ROLLUP, just those that should be summarized. With this GROUP BY clause, the results won't include grand totals because we're not rolling up the country.

```
GROUP BY CountryRegion.Name,
         ROLLUP(Address.City,
                YEAR(OrderDate),
                MONTH(OrderDate))
```

Note also that when ROLLUP is involved, you use the source field names, not the result field names in the GROUP BY clause.

ROLLUP with cross-products

You can use two ROLLUP clauses in the same GROUP BY. Doing so gives you the cross-product of the two groups. That is, you get the results you'd get from either ROLLUP, but you also get combinations of the two.

For example, if you change the GROUP BY clause in **Listing 44** to the one shown in **Listing 47**, you get all the rows you had before, but you also get summaries for each country for each month and year, as well as overall summaries for each month and for each year. **Figure 20** shows part of the results. The complete query is included in the materials for this session as SalesByCountryCityRollupXProd.SQL.

Listing 47. You can use two ROLLUP clauses to generate the cross-product of the two sets of fields.

```
GROUP BY ROLLUP(YEAR(OrderDate), MONTH(OrderDate)),
         ROLLUP(CountryRegion.Name, Address.City)
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
United King...	NULL	2008	6	777041.13	1137.6883	683
United King...	NULL	2008	7	11942.84	60.6235	197
United King...	NULL	2008	NULL	3403936.78	951.0859	3579
United States	NULL	2008	1	1241090.59	783.0224	1585
United States	NULL	2008	2	1417644.54	857.6192	1653
United States	NULL	2008	3	1407198.05	831.6773	1692
United States	NULL	2008	4	1457717.98	807.1528	1806
United States	NULL	2008	5	2066050.29	979.1707	2110
United States	NULL	2008	6	1926201.99	965.031	1996
United States	NULL	2008	7	52011.40	63.2742	822
United States	NULL	2008	NULL	9567914.84	820.2944	11664
NULL	NULL	2008	NULL	27419405...	848.9505	32298
NULL	NULL	2006	9	350466.9...	1770.0353	198
NULL	NULL	2007	4	507965.2...	1716.0988	296
NULL	NULL	2005	10	513329.474	3188.3818	161

Figure 20. Using the GROUP BY clause in **Listing 47** with the earlier query provides summaries for not just each city by year, each city overall, and each country, but also for each country by month and by year, and for each month and each year.

As with a single ROLLUP clause, the order in which you list the ROLLUP clauses and the order of the fields within them determines both what summaries you get and, if you don't use an ORDER BY clause, the order of the records in the result.

Adding descriptions to summaries

In all the examples so far, the null value indicates which field is being summarized. But you can put descriptive data in those fields instead.

Wrap the columns being rolled up with ISNULL() and specify the string you want in the summary rows as the alternate. (ISNULL() in SQL Server behaves like VFP's NVL() function, returning the first parameter unless it's null, in which case it returns the second parameter.) **Listing 48** (SalesByCountryCityRollupWDesc.SQL in the materials for this session) shows the same query as **Listing 44**, except that each of the non-aggregated fields includes a description to use when it's summarized. Doing so requires changing the year and month columns to character, of course. **Figure 21** shows a chunk of the results.

Listing 48. Rather than having null indicate a summary row, use the description you want.

```
SELECT ISNULL(Person.CountryRegion.Name, 'All countries') AS Country,
       ISNULL(Person.Address.City, 'All cities') AS City,
       ISNULL(STR(YEAR(OrderDate)), 'All years') AS OrderYear,
       ISNULL(STR(MONTH(OrderDate)), 'All months') AS OrderMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
  GROUP BY ROLLUP(CountryRegion.Name, Address.City,
                  YEAR(OrderDate), MONTH(OrderDate))
```

Country	City	OrderY...	OrderMo...	TotalSales	AvgSale	NumSales
Australia	Wollongong	2008	2	29543.26	1477.163	20
Australia	Wollongong	2008	3	46330.36	1494.5277	31
Australia	Wollongong	2008	4	42357.88	1033.119	41
Australia	Wollongong	2008	5	39493.51	1128.386	35
Australia	Wollongong	2008	6	64000.30	1280.006	50
Australia	Wollongong	2008	7	367.84	91.96	4
Australia	Wollongong	2008	All months	268554.13	1272.7683	211
Australia	Wollongong	All years	All months	618257.4...	1504.2761	411
Australia	All cities	All years	All months	16322659...	1223.1292	13345
Canada	Burnaby	2005	8	3578.27	3578.27	1
Canada	Burnaby	2005	All months	3578.27	3578.27	1
Canada	Burnaby	2006	3	3578.27	3578.27	1
Canada	Burnaby	2006	5	3578.27	3578.27	1

Figure 21. Including descriptions instead of null makes it easier to understand the summary lines.

Introducing CUBE

ROLLUP is limited to summarizing based only on the hierarchy you specify. For example, the query in **Listing 44** doesn't give summaries for each country for each year. While you can get that result with ROLLUP, you have to give up some other summaries to do so.

If you want to summarize based on every possible combination of values, use CUBE rather than ROLLUP. The query in **Listing 49** is identical to the one in **Listing 44**, except that the GROUP BY clause specifies CUBE rather than ROLLUP. **Figure 22** shows part of the results. The items at the top of the grid include summaries you wouldn't get with ROLLUP, such as the summary for all locations in all Decembers about halfway down and the summary for Australia for all of 2005 in the last row shown. This query is included in the materials for this session as SalesByCountryCityCubeNoOrder.sql.

Listing 49. Use the CUBE clause to get summaries for all combinations of values.

```
SELECT Person.CountryRegion.Name AS Country, Person.Address.City,
       YEAR(OrderDate) AS nYear, MONTH(OrderDate) AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
  GROUP BY CUBE(CountryRegion.Name, Address.City,
                YEAR(OrderDate), MONTH(OrderDate))
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
United Kingdom	Woolston	2007	12	16936.33	1129.0886	15
NULL	Woolston	2007	12	16936.33	1129.0886	15
United States	Yakima	2007	12	22768.39	875.7073	26
NULL	Yakima	2007	12	22768.39	875.7073	26
United Kingdom	York	2007	12	50720.73	1334.756	38
NULL	York	2007	12	50720.73	1334.756	38
NULL	NULL	2007	12	4899275...	927.3661	5283
NULL	NULL	NULL	12	6233117...	1065.6723	5849
NULL	NULL	NULL	NULL	5927376...	980.3961	60459
Australia	NULL	2005	7	209652....	3381.4984	62
Australia	NULL	2005	8	222538....	3272.6219	68
Australia	NULL	2005	9	173993....	3346.029	52
Australia	NULL	2005	10	217993....	3353.7443	65
Australia	NULL	2005	11	210683....	3344.1835	63
Australia	NULL	2005	12	274185....	3264.1136	84

Figure 22. When you specify CUBE, every possible combination of values is summarized.

However, some of the results of this query are misleading. The first few rows in **Figure 22** should give you a clue as to the problem. We're summarizing by name of a city for a month. What if we have multiple cities with the same name? In fact, this data set contains several repeated city names, among them Birmingham. **Figure 23** shows that when both Birminghams have data for a given month, we get a total for that month that covers both cities, which is meaningless.

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
United Kingdom	Birmingham	NULL	6	9782.32	543.4622	18
NULL	Birmingham	NULL	6	9782.32	543.4622	18
United Kingdom	Birmingham	NULL	7	8189.09	1169.87	7
United States	Birmingham	NULL	7	35.00	35.00	1
NULL	Birmingham	NULL	7	8224.09	1028.0112	8
United Kingdom	Birmingham	NULL	8	1714.0957	428.5239	4
NULL	Birmingham	NULL	8	1714.0957	428.5239	4
United Kingdom	Birmingham	NULL	9	7298.8825	1042.6975	7
NULL	Birmingham	NULL	9	7298.8825	1042.6975	7
United Kingdom	Birmingham	NULL	10	13124.3...	1009.5621	13
NULL	Birmingham	NULL	10	13124.3...	1009.5621	13
United Kingdom	Birmingham	NULL	11	6083.63	675.9588	9
United States	Birmingham	NULL	11	2.29	2.29	1
NULL	Birmingham	NULL	11	6085.92	608.592	10
United Kingdom	Birmingham	NULL	12	17644.17	1604.0154	11

Figure 23. Some of the summarized results can be misleading if fields are dependent on each other. Here, we get totals for a given month for both Birminghams.

The way to avoid the problem is to group fields together if their data is linked. You do that by putting parentheses around the fields to be grouped. **Listing 50** shows the same query, but with the Country and City fields grouped together. (It also has an ORDER BY clause to sort the results into a useful order.) It's included in the materials for this session as SalesByCountryCityCubeCombined.sql. **Figure 24** shows partial results; note that there are no totals where Name is null, but City is not.

Listing 50. Group fields with parentheses in the CUBE clause to have them treated as a single dimension.

```
SELECT Person.CountryRegion.Name AS Country, Person.Address.City,
       YEAR(OrderDate) AS nYear, MONTH(OrderDate) AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
  GROUP BY CUBE((CountryRegion.Name, Address.City),
                YEAR(OrderDate),
                MONTH(OrderDate))
  ORDER BY Country, City, nYear, nMonth
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
NULL	NULL	2008	6	5534993.75	996.7573	5553
NULL	NULL	2008	7	137184.87	62.0465	2211
Australia	Bendigo	NULL	NULL	555431.9893	1402.606	396
Australia	Bendigo	NULL	1	45717.8121	1576.4762	29
Australia	Bendigo	NULL	2	35863.8521	1434.554	25
Australia	Bendigo	NULL	3	47283.4582	1432.832	33
Australia	Bendigo	NULL	4	63837.0056	1329.9376	48
Australia	Bendigo	NULL	5	61997.1378	1265.2477	49
Australia	Bendigo	NULL	6	63823.672	1556.6749	41
Australia	Bendigo	NULL	7	60588.16	1731.0902	35
Australia	Bendigo	NULL	8	28585.1242	1242.8314	23
Australia	Bendigo	NULL	9	23499.03	1382.2958	17
Australia	Bendigo	NULL	10	52875.3221	1429.0627	37
Australia	Bendigo	NULL	11	34616.54	1081.7668	32
Australia	Bendigo	NULL	12	36744.8752	1360.9213	27
Australia	Bendigo	2005	NULL	41972.84	3497.7366	12
Australia	Bendigo	2005	7	20909.78	3484.9633	6
Australia	Bendigo	2005	9	3578.27	3578.27	1
Australia	Bendigo	2005	10	6953.26	3476.63	2

Figure 24. With country and city grouped, the results don't have totals for a city without the associated country.

If you don't want summaries for each month across the years (that is, for example, for all Aprils), you can group year and month in the CUBE clause, as well, as in **Listing 51**. A query

that uses this CUBE clause is included in the materials for this session as SalesByCountryCityCubeCombinedBoth.sql.

Listing 51. You can have multiple groups of fields within the CUBE clause.

```
GROUP BY CUBE((CountryRegion.Name, Address.City),
              (YEAR(OrderDate), MONTH(OrderDate)))
```

Fine tuning the set of summaries

ROLLUP and CUBE take care of very common scenarios, but each is restricted in which set of summaries you can get, and each includes the basic aggregated data in the result. What if you want a different set of summaries? What if you want just the summaries without the basic aggregated data?

In our example, suppose you want to see the summary for each month across all years and locations, the summary for each year across all months and locations, and the summary for each location across all months and years? You could get those results by doing a separate query for each and then combining them with UNION ALL, as in **Listing 52** (SummariesUnion.SQL in the materials for this session); **Figure 25** shows partial results.

Listing 52. You can retrieve just the summaries using UNION ALL.

```
SELECT Person.CountryRegion.Name AS Country, Person.Address.City,
       null AS nYear, null AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
    GROUP BY Person.CountryRegion.Name, City
UNION ALL
SELECT NULL AS Country, NULL City,
       NULL AS nYear, MONTH(OrderDate) AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.SalesOrderHeader
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
```

```
  GROUP BY MONTH(OrderDate)
UNION ALL
SELECT NULL AS Country, NULL AS City,
       YEAR(OrderDate) AS nYear, NULL AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.SalesOrderHeader
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
    GROUP BY YEAR(OrderDate)
    ORDER BY Country, City, nYear, nMonth
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
NULL	NULL	NULL	7	228486423.7421	27584.9841	8283
NULL	NULL	NULL	8	346541500.3052	29440.2769	11771
NULL	NULL	NULL	9	314385828.5747	29177.339	10775
NULL	NULL	NULL	10	166185042.8543	20133.8796	8254
NULL	NULL	NULL	11	271459474.9601	24909.1094	10898
NULL	NULL	NULL	12	235620076.8434	20664.8023	11402
NULL	NULL	2005	NULL	141944504.3041	27556.6888	5151
NULL	NULL	2006	NULL	779150362.1894	40259.9267	19353
NULL	NULL	2007	NULL	1169638118.0044	22827.9976	51237
NULL	NULL	2008	NULL	505737472.1795	11096.5743	45576
Australia	Bendigo	NULL	NULL	555431.9893	1402.606	396
Australia	Brisbane	NULL	NULL	546014.4579	1403.6361	389
Australia	Caloundra	NULL	NULL	527130.8302	1301.5576	405
Australia	Cloverdale	NULL	NULL	384307.4948	1311.6296	293

Figure 25. Sometimes, you want only the summaries, not the original aggregations.

That's a lot of code. SQL Server offers an alternative way to do this, using a feature called grouping sets. They let you fine tune which summaries you get. With grouping sets, you explicitly tell the query which combinations to summarize. The grouping sets equivalent of the UNIONed query in **Listing 52** is shown in **Listing 53** (included in the materials for this session as SummariesGroupingSets.SQL).

Listing 53. GROUPING SETS let you ask for the specific set of summaries you want.

```
SELECT Person.CountryRegion.Name AS Country, Person.Address.City,
       YEAR(OrderDate) AS nYear, MONTH(OrderDate) AS nMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
```

```
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
      ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
  GROUP BY GROUPING SETS ((CountryRegion.Name, Address.City),
                          (YEAR(OrderDate)),
                          (MONTH(OrderDate)))
  ORDER BY Country, City, YEAR(OrderDate), MONTH(OrderDate)
```

The GROUP BY clause indicates three grouping sets here, each enclosed in parentheses: (CountryRegion.Name, Address.City) which says to show totals for each city and country combination, across all years and months; (YEAR(OrderDate)), which asks for totals for each year, across all locations and months; and (MONTH(OrderDate)), which requests totals for each month, across all locations and years. The parentheses are required in the first case, to show that city and country are to be treated as a set. While they're not required for the other two items, they do make clear that each is to be handled separately.

ROLLUP and CUBE are actually special cases of grouping sets. You can use grouping sets to get the same results, though it actually makes the code longer. **Listing 54** shows the GROUP BY clause for the grouping sets equivalent of the ROLLUP query in **Listing 44**. (The complete version of this query is included in the materials for this session as GroupingSetsRollupEquiv.sql.)

Listing 54. You can use GROUPING SETS instead of ROLLUP, but it calls for more code in the GROUP BY clause.

```
GROUP BY GROUPING SETS (
  (CountryRegion.Name, Address.City, YEAR(OrderDate), MONTH(OrderDate)),
  (CountryRegion.Name, Address.City, YEAR(OrderDate)),
  (CountryRegion.Name, Address.City),
  (CountryRegion.Name),
  ())
```

There are five grouping sets shown. The first set, which includes all four non-aggregated fields is the equivalent of simply doing GROUP BY with that list. It does the aggregation, but no summaries.

Each grouping set after that contains one fewer field than the preceding one, until the last contains no field, indicating that the summary should be computed over the entire data set. Looking at this GROUP BY clause actually helps to clarify what ROLLUP does. It aggregates on all the fields listed, then one by one, removes fields from the right and aggregates again.

For the equivalent of CUBE, the GROUPING SETS list is even more unwieldy, but again it sheds light on what's going on when you use CUBE. **Listing 55** shows the GROUP BY clause for a query (GroupingSetsCubeCombinedEquiv.sql in the materials for this session) that produces the same results as **Listing 50**.

Listing 55. Replacing CUBE with GROUPING SETS lets you see all the cases that CUBE handles.

```
GROUP BY GROUPING SETS(
  (CountryRegion.Name, Address.City, YEAR(OrderDate), MONTH(OrderDate)),
  (CountryRegion.Name, Address.City, YEAR(OrderDate)),
  (CountryRegion.Name, Address.City, MONTH(OrderDate)),
  (CountryRegion.Name, Address.City),
  (YEAR(OrderDate), MONTH(OrderDate)),
  (YEAR(OrderDate)),
  (MONTH(OrderDate)),
  ())
```

Note that unlike the CUBE query, you don't have to (in fact, can't) enclose the country/city pair in parentheses here. You just omit any grouping sets that include one without the other.

Of course, there's no reason to write out the long version when you can use ROLLUP or CUBE. But when you need something else, having grouping sets available is a big help.

As **Listing 53** demonstrates, grouping sets also let you get summaries without including the basic aggregated data. Just omit the grouping set that lists all the fields on which to aggregate. Be aware, though, that as with any other GROUP BY clause, every field in the field list that doesn't include an aggregate function must appear somewhere in the list of grouping sets.

Listing 56 shows the GROUP BY clause for a query that's equivalent to **Listing 50**, but without the first grouping set, so that only the summaries are included. **Figure 26** shows partial results; if you compare to **Figure 24**, you can see that the rows where nothing is null have been eliminated. This query is included as GroupingSetsWithoutAggregates.sql in the materials for this session.

Listing 56. By omitting the grouping set that includes all non-aggregated fields, you can get just the summaries you want without the base aggregated data.

```
GROUP BY GROUPING SETS(
  (CountryRegion.Name, Address.City, YEAR(OrderDate)),
  (CountryRegion.Name, Address.City, MONTH(OrderDate)),
  (CountryRegion.Name, Address.City),
  (YEAR(OrderDate), MONTH(OrderDate)),
  (YEAR(OrderDate)),
  (MONTH(OrderDate)),
  ())
```

Country	City	nYear	nMonth	TotalSales	AvgSale	NumSales
NULL	NULL	2008	6	5534993.75	996.7573	5553
NULL	NULL	2008	7	137184.87	62.0465	2211
Australia	Bendigo	NULL	NULL	555431.9893	1402.606	396
Australia	Bendigo	NULL	1	45717.8121	1576.4762	29
Australia	Bendigo	NULL	2	35863.8521	1434.554	25
Australia	Bendigo	NULL	3	47283.4582	1432.832	33
Australia	Bendigo	NULL	4	63837.0056	1329.9376	48
Australia	Bendigo	NULL	5	61997.1378	1265.2477	49
Australia	Bendigo	NULL	6	63823.672	1556.6749	41
Australia	Bendigo	NULL	7	60588.16	1731.0902	35
Australia	Bendigo	NULL	8	28585.1242	1242.8314	23
Australia	Bendigo	NULL	9	23499.03	1382.2958	17
Australia	Bendigo	NULL	10	52875.3221	1429.0627	37
Australia	Bendigo	NULL	11	34616.54	1081.7668	32
Australia	Bendigo	NULL	12	36744.8752	1360.9213	27
Australia	Bendigo	2005	NULL	41972.84	3497.7366	12
Australia	Bendigo	2006	NULL	68205.8315	2435.9225	28
Australia	Bendigo	2007	NULL	211159.4078	1311.5491	161

Figure 26. When you exclude the grouping set that contains all aggregated fields, the result contains only the summaries.

Make it pretty

As with the ROLLUP clause, for both CUBE and GROUPING SETS, you can make the results easier to understand by using ISNULL() to replace the nulls with meaningful descriptions.

Listing 57 shows the query from **Listing 50** with the descriptions added. **Figure 27** shows partial results. The query is included in the materials for this session as SalesByCountryCityCubeCombinedWDesc.sql.

Listing 57. You can replace the nulls that indicate summary records with descriptions.

```
SELECT ISNULL(Person.CountryRegion.Name, 'All countries') AS Country,
       ISNULL(Person.Address.City, 'All cities') AS City,
       ISNULL(STR(YEAR(OrderDate)), 'All years') AS cYear,
       ISNULL(STR(MONTH(OrderDate)), 'All months') AS cMonth,
       SUM(SubTotal) AS TotalSales,
       AVG(SubTotal) AS AvgSale,
       COUNT(SubTotal) AS NumSales
  FROM Sales.Customer
    JOIN Person.Person
      ON Customer.PersonID = Person.BusinessEntityID
    JOIN Person.BusinessEntityAddress
      ON Person.BusinessEntityID = BusinessEntityAddress.BusinessEntityID
    JOIN Person.Address
      ON BusinessEntityAddress.AddressID = Address.AddressID
    JOIN Person.StateProvince
      ON Address.StateProvinceID = StateProvince.StateProvinceID
    JOIN Person.CountryRegion
      ON StateProvince.CountryRegionCode = CountryRegion.CountryRegionCode
    JOIN Sales.SalesOrderHeader
      ON Customer.CustomerID = SalesOrderHeader.CustomerID
    JOIN Sales.SalesOrderDetail
```

```
    ON SalesOrderHeader.SalesOrderID = SalesOrderDetail.SalesOrderID
GROUP BY CUBE((CountryRegion.Name, Address.City),
              YEAR(OrderDate),
              MONTH(OrderDate))
ORDER BY Country, City, cYear, cMonth
```

Country	City	cYear	cMonth	TotalSales	AvgSale	NumSales
All countries	All cities	All years	9	3649551.8024	862.5742	4231
All countries	All cities	All years	10	4014666.7773	884.6775	4538
All countries	All cities	All years	11	4190634.2945	923.0472	4540
All countries	All cities	All years	12	6233117.4491	1065.6723	5849
All countries	All cities	All years	All months	59273769.3...	980.3961	60459
Australia	Bendigo	2005	7	20909.78	3484.9633	6
Australia	Bendigo	2005	9	3578.27	3578.27	1
Australia	Bendigo	2005	10	6953.26	3476.63	2
Australia	Bendigo	2005	11	3578.27	3578.27	1
Australia	Bendigo	2005	12	6953.26	3476.63	2
Australia	Bendigo	2005	All months	41972.84	3497.7366	12
Australia	Bendigo	2006	1	10734.81	3578.27	3
Australia	Bendigo	2006	2	10734.81	3578.27	3
Australia	Bendigo	2006	3	3578.27	3578.27	1
Australia	Bendigo	2006	4	3578.27	3578.27	1

Figure 27. You can use ISNULL() to substitute descriptions for nulls, and make the results easier to comprehend.

What about VFP?

I showed how to do the equivalent of ROLLUP in VFP. The second approach shown there, using a separate query for each summary you want, and then combining the results with UNION, works for CUBE and GROUPING SETS, as well. Of course, the resulting code is fairly opaque. That's why having these shortcuts in SQL Server is so nice.

Keep on learning

While I read articles and examples of each of these features to learn them, it was trying different variations that really helped me understand them. I strongly recommend you start with the examples here and then try building analogous code against your own data, or modifying this code to see the results.

Beyond that, the features in this paper are only a subset of those T-SQL offers that aren't part of VFP's SQL. If you're really trying to learn more T-SQL, find a SQL Q&A forum and start reading. I've learned a lot reading the one at www.tek-tips.com; http://www.sqlservercentral.com/ has articles and Q&A forums. There are lots of others, as well.