Consolidate data from a field into a list

This task is hard in VFP, but SQL Server provides two ways to do it.

Tamar E. Granor, Ph.D.

Some SQL commands were added to FoxPro 2.0 and I fell in love with them as soon as I started playing around. Over the years, Visual FoxPro's SQL subset has grown, but there are still some tasks that are hard or impossible to do in VFP, but a lot easier in other SQL dialects. In my next few articles, 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.

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, for example, create a comma-separated list of all the values, 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 (probably 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.

VFP's SQL commands offers 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.

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

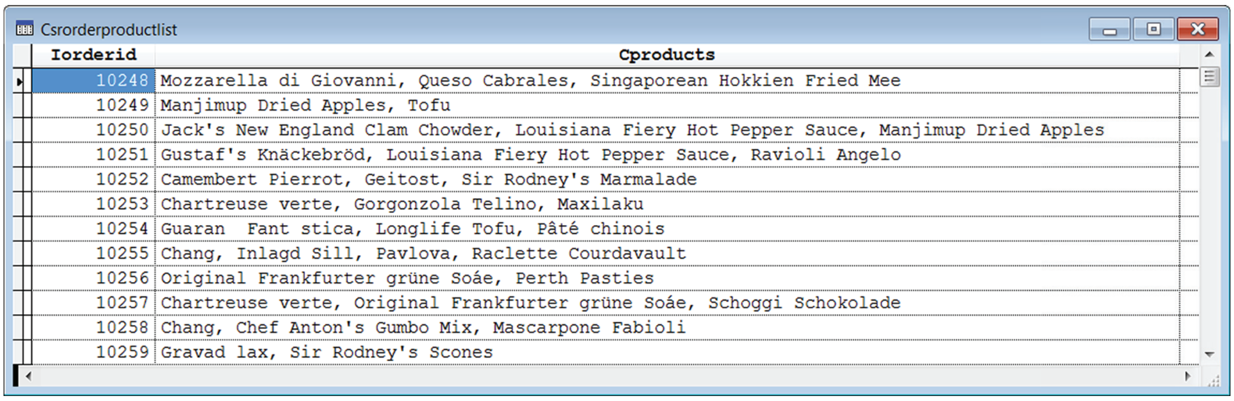
```
OPEN DATABASE FORCEPATH("Northwind", ;
  ADDBS(_samples) + "Northwind")

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
```


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 shows each order from the Northwind database with a comma-separated list of the products ordered.

```
  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 this month's downloads 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 each one.

We'll use the sample AdventureWorks (2008) database to demonstrate. 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 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 simply 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 article, 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. The derived table is required because using DISTINCT 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).

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().)

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). This query is included in this month's downloads as RollupOrdersForXML.SQL.

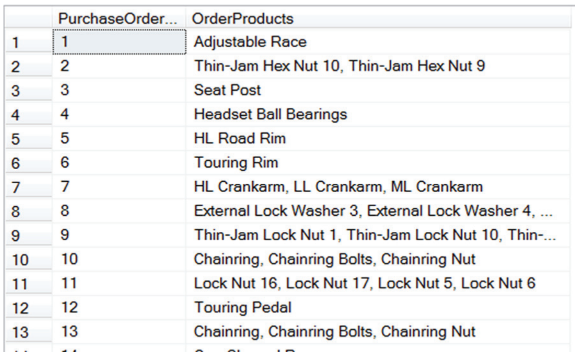
Listing 11. Combining the query from Listing 10 with code to include the purchase order number gives us the desired results.

```
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.Pur-
chaseOrderID
      = A.PurchaseOrderID) DistName
  ORDER BY Name
  FOR XML PATH('')), 1, 2, '') OrderProducts
 FROM Purchasing.PurchaseOrderDetail
 GROUP BY PurchaseOrderID
 ORDER BY 1
```

This solution is included in this month's downloads 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.



	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.

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 SSSMS, 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. 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 the problem 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 this month's downloads 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 testbed. 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.

Author Profile

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