Turning Data Sideways: Crosstabs and Pivot Tables

Tamar E. Granor
Tomorrows Solutions, LLC
Web: www.tomorrowssolutionsllc.com
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

In many applications, you need to take nice, normalized data and turn it "sideways," using the values in one or more columns to specify columns in the result. Most often, the process also involves aggregating data for each of those values. In Visual FoxPro, the result is called a cross-tab; SQL Server calls it a pivot. Once you've created a cross-tab or pivot, reporting can be difficult because the result may have many, many columns.

In this session, we'll see how to generate cross-tabs and pivots, as well as how to report on them, including exporting to Excel and creating graphs of the results.

Introduction

Normalizing data is one of the keys to working with relational databases. Store every data item once and only once and use relationships among the tables to connect disparate pieces of data. While normalization is great for storing and manipulating data, it's not unusual to need to denormalize data for reporting purposes. We do this, for example, by adding a customer's name and address to order data to produce a sales order we can send to the customer for confirmation.

One common need is to take data and turn it into column headers in order to see how two data items interact. For example, you might want to see the sales for each salesperson by year, or the number of students in each department receiving each grade. This kind of manipulation of data is called a *crosstab* (VFP's preferred term) or *pivot* (SQL Server's preferred term), and most database systems provide a way to create them.

Once you've created a crosstab or pivot, reporting on it can be challenging because you may not know ahead of time how many columns you have or the names of those columns. But it is possible to put such data into VFP reports, as well as to export it to Excel or to graph it with either VFP or Excel.

This paper first looks at how to create crosstabs and pivots in VFP and SQL Server, and then shows several ways to display that data to users. Except where noted, all VFP examples in this paper work with the Northwind database, while SQL Server examples use the AdventureWorks 2019 database (but should work with older versions of AdventureWorks).

Creating crosstabs in VFP

We'll look at several tools for creating crosstabs from Visual FoxPro data. To start exploring crosstabs, though, we'll start with something simpler. The sample Northwind company has employees only in the US and the UK. Suppose you want to know how many there are in each country. A simple query, shown in **Listing 1**, gives you the answer; there are 4 UK employees and 5 US employees.

Listing 1. It's easy to get one record for each value of interest. Here, each record indicates how many employees are in a given country.
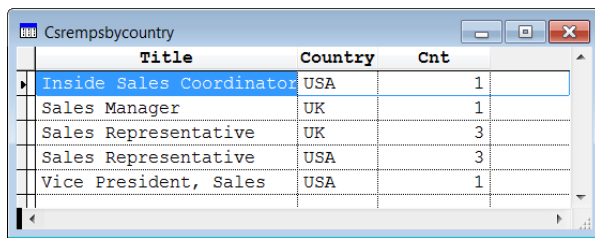
```
SELECT country, COUNT(*) ;
  FROM Employees ;
  GROUP BY Country ;
  INTO CURSOR csrEmpsByCountry
```

But what if you want to know how many there are in each job in each country? Again, that's not hard. **Listing 2** gives us one record for each combination of job title and country; results are shown in **Figure 1**.

Listing 2. Group on more fields to break the data down into smaller groups.

```
SELECT Title, Country, COUNT(*) ;
```

```
 FROM Employees ;
 GROUP BY Title, Country ;
 INTO CURSOR csrEmpsByCountry
```
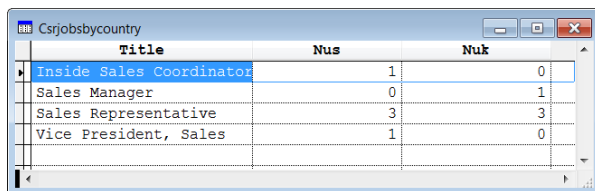

Figure 1. Grouping on Title and Country lets us see how many employees have each job in each country.

But what if what you really want is one record for each job title with a column for each country showing how many employees in that country have that job? That's a crosstab. For a simple case like this, you can get what you need with IIF(), as in **Listing 3** (included in the downloads for this session as JobsByCountry.PRG). **Figure 2** shows the results, with one row for each job title and one column for each country.

Listing 3. Here, combining SUM() with IIF() counts the number of employees in each country with each job.

```
SELECT Title, ;
       SUM(IIF(Country="US",1,0)) AS nUS,
       SUM(IIF(Country="UK",1,0)) AS nUK ;
  FROM Employees ;
  GROUP BY Title ;
  INTO CURSOR csrJobsByCountry
```


Figure 2. A crosstab uses data to determine what columns are in the result.

This approach works fine when you know the exact number of columns you want and it's not too many. But more often, you don't know how many columns will be in the result, and you may not even know the set of values they're drawn from. That's where a crosstab tool comes in handy.

VFPXTab

A crosstab tool has been included since FoxPro 2.0, where it was called GenXTab. The version that comes with VFP is VFPXTab.PRG and it's found in the VFP home folder. The system variable _GENXTAB points to it; the Query Designer uses _GENXTAB when you create a crosstab query.

Comments at the top of VFPXTab.PRG explain its parameters and show how to call it, including the structure of the data needed for it to work. Basically, it expects to find a table

or cursor with three columns. By default, the data in the first column becomes the rows in the result, the data in the second column becomes the columns in the result, and the data in the third column is aggregated (summed, by default) to form the data in the result.

For example, suppose we want the total sales for each salesperson for each year with a row for each salesperson and a column for each year. We start with a query to collect the relevant data, shown in **Listing 4**; **Figure 3** shows partial results.

Listing 4. This query gives us one record for each employee for each year, showing total sales.

```
SELECT EmployeeID, ;
       YEAR(OrderDate) AS OrderYear, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
  FROM Orders ;
    JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY 1, 2 ;
  INTO CURSOR csrYearlyTotals
```
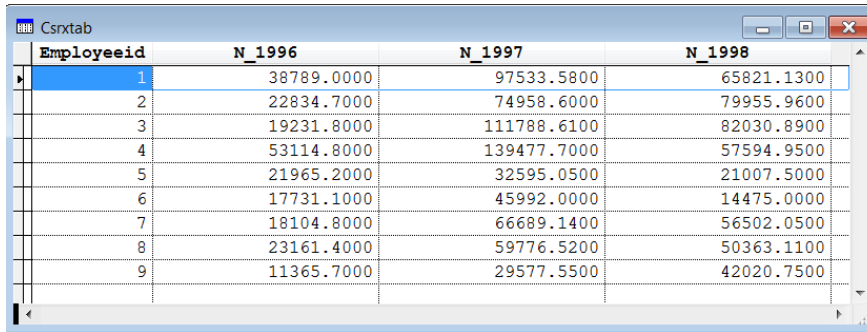
Employeeid	Orderyear	Ordertotal
1	1996	38789.0000
1	1997	97533.5800
1	1998	65821.1300
2	1996	22834.7000
2	1997	74958.6000
2	1998	79955.9600
3	1996	19231.8000
3	1997	111788.6100
3	1998	82030.8900
4	1996	53114.8000
4	1997	139477.7000
4	1998	57594.9500
5	1996	21965.2000
5	1997	32595.0500
5	1998	21007.5000
6	1996	17731.1000

Figure 3. The query in Listing 4 results in one record per salesperson per year.

To get the results we want, we instantiate the GenXTab class in VFPXTab.PRG and call its MakeXTab method, as in **Listing 5**. The class takes 10 parameters that configure its behavior. (Because I don't recommend using this crosstab tool, I'm not going to go into detail about the parameters. They're documented in the code.) The two parameters passed in the example indicate that the result should be stored in a cursor (rather than a table) named csrXTab. **Figure 4** shows the result.

Listing 5. The GenXTab class in VFPXTab accepts up to 10 parameters.

```
oXTab = NEWOBJECT("genxtab", _GENXTAB,'', "csrXTab", .t.)
oXTab.MakeXtab()
```

Figure 4. This crosstab has one row per employee and one column per year.

A complete program collecting the data and running the crosstab, as well as timing the crosstab process, is included in the downloads for this session as SalesPersonAnnualSales.PRG. The downloads also include SalesPersonMonthlySales.PRG, which generates a crosstab with one row per salesperson and a column for each month of 1997.

As the size of the data set supplied to VFPXTab grows, generating a crosstab gets much slower. In **Listing 6**, the result has one row for each date in the data set and a column for each salesperson; **Figure 5** shows partial results. This example is included in the downloads for this session as SalesPersonDailySales.PRG.

Listing 6. Here, the result has one row per day with one column per salesperson.

```
SELECT OrderDate, ;
       EmployeeID, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
  FROM Orders ;
    JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY 1, 2 ;
  INTO CURSOR csrDailyTotals

LOCAL oXTab

oXTab = NEWOBJECT("genxtab", _GENXTAB,'', "csrXTab", .t.)
oXTab.MakeXtab()
```


Figure 5. Partial results for a crosstab of daily sales by salesperson. There are more rows in the full result.

In my tests, the crosstab with one row per salesperson and one column per year varies from about 3/100th of a second to about 5/100th of a second, but the daily sales example takes about 3/10th of a second. (In each case, I'm timing only crosstab generation, not the query that assembles the data.)

FastXTab

Fortunately, there's a faster way to get crosstabs. FastXTab was created by Alexander Golovlev specifically to address VFPXTab's speed issues. It's can be downloaded from Level Extreme at https://www.levelextreme.com/ShowHeaderDownloadOneItem.aspx?ID=9944. FastXTab replaces VFPXTab's long list of parameters (which were a throwback to the program's non-OOP origins) with properties. **Table 1** shows the key properties.

Table 1. FastXTab uses properties for configuration.

Property	Default value	Purpose
cOutFile	"xtabquery"	Alias of the output table or cursor
lCursorOnly	.F.	Indicates whether the results are placed in a table (.F.) or cursor (.T.)
lCloseTable	.T.	Indicates whether the source table should be closed after creating the crosstab
nRowField	1	Indicates which column of the source table provides the rows
nColField	2	Indicates which column of the source table provides the columns
nDataField	3	Indicates which column of the source table provides the data to be aggregated
lTotalRows	.F.	Indicates whether the result should include an extra row totaling the data in the other rows
lBrowseAfter	.F.	Indicates whether the result should be opened in a Browse window

The use of properties makes code that calls FastXTab easier to read than code that calls VFPXTab. **Listing 7** shows the FastXTab equivalent of **Listing 5**; it's based on the same query (shown in **Listing 4**) and, of course, produces the same results. The complete code, including timing test is included as SalesPersonAnnualSalesFast.prg in the downloads for this session.

Listing 7. With FastXTab, the options you choose are easy to understand because you specify them with properties.

```
oXTab = NEWOBJECT("fastxtab", "fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab.prg"
   .cOutFile = "csrXtab"
   .lCursorOnly = .T.
   .lCloseTable = .T.
   .RunXtab()
ENDWITH
```

For this example, FastXTab is only marginally faster than VFPXTab. (In my tests, the slowest run using FastXTab took the same time as the fastest run using VFPXTab.) But for the daily sales example, you can see the major improvement FastXTab offers. Using the same original query as in **Listing 6** and the call to FastXTab shown in **Listing 7**, on my computer, the crosstab is computed in between 2/100th of a second and 3/100th of a second, that is, about two orders of magnitude faster than with VFPXTab. The code, including timing test, is included in the downloads for this session as SalesPersonDailySalesFast.prg.

FastXTab 1.6

The additional speed and readability are enough to make me recommend using FastXTab rather than VFPXTab. But, as the infomercials say, "wait … there's more."

Community member Vilhelm-Ion Praisach has extended FastXTab considerably, both for usability and to provide additional capabilities. It's included in the downloads for this session as FastXTab16.RAR. Since I first presented this session in 2017, Praisach's version has been added to VFPX, so the latest can always be downloaded from https://github.com/VFPX/FastXTab.

There's documentation on that VFPX page, but you can find Praisach's original documentation at http://praisachion.blogspot.com/2015/02/fastxtab-version-16.html. The documentation includes lots of examples; I've broken those examples out into individual PRGs, which are included as FastXTab16Demos.ZIP in the downloads for this session.

Among the things supported in Praisach's version are specifying the relevant fields by name rather than column number; specifying multiple data columns; specifying the function to apply to a given data column, including with a custom expression; and filtering the data source.

All the properties supported by the original FastXTab remain, but there are quite a few new properties to provide new capabilities. **Table 2** shows some of them.

Table 2. FastXTab 1.6 includes many new properties to support new capabilities.

Property	Default value	Purpose
cRowField	""	Expression from the source table that provides the rows
cColField	""	Expression from the source table that provides the columns
cDataField	""	Name of the column in the source table that provides the data to be aggregated
cPageField	""	Expression from the source table that provides "pages"
nFunctionType	1	Indicates how to aggregate the data; see Table 3.
cFunctionExp	""	Expression to use for custom aggregation
cCondition	""	Filter to apply to source data before aggregating
cHaving	""	Filter to apply to source data after aggregating
nMultiDataField	1	Indicates how many data fields are specified
anDataField		Array of data fields specified by column position
acDataField		Array of data fields specified by name
anFunctionType		Array of aggregation function choices
acFunctionExp		Array of custom aggregation expressions

Calling FastXTab 1.6 is the same as calling the original FastXTab (which I'll refer to as FastXTab 1.0), except that you need to point to the FastXTab 1.6 version of FastXTab.PRG. The downloads for this session include SalesPersonAnnualSalesFast16.PRG, which is identical to SalesPersonAnnualSalesFast.PRG, except for the path to the class library. In my tests, it runs a little faster than the FastXTab 1.0 version.

But FastXTab 1.6 lets you do much more. Suppose you want to know how many sales each salesperson had in each year. Specifying 2 for the nFunctionType property indicates Count; **Table 3** shows the options for this property.

Table 3. FastXTab 1.6 offers six ways of aggregating the data.

nFunctionType	Meaning
1	Sum
2	Count
3	Average
4	Min
5	Max
6	Custom expression specified in cFunctionExp (or acFunctionExp for the relevant column)
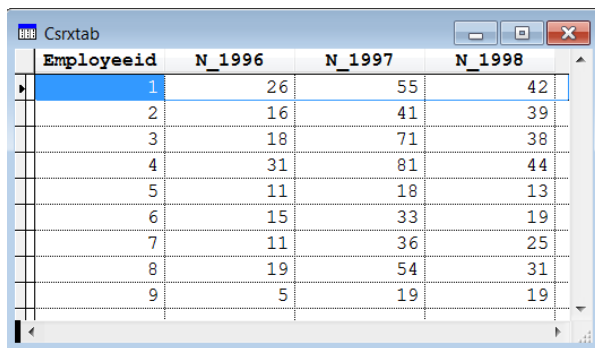
The code in **Listing 8** (included as SalesPersonAnnualSalesCountFast16.prg in the downloads for this session) shows another cool feature of FastXTab 1.6, the ability to use an expression to specify the rows or columns rather than just a field name. To get the correct results here, we need to see every order, but we want them counted by year. So the query that gathers the data keeps the original OrderDate column, but the cColField property specifies "YEAR(OrderDate)," so that the result has one column per year. (In fact, the data collection query is unnecessary here. FastXTab could be run directly against the Orders table.) **Figure 6** shows the results.

Listing 8. FastXTab 1.6 lets you specify the aggregation function to apply, as well as specify expressions for rows and columns.

```
SELECT EmployeeID, OrderDate, OrderID ;
    FROM Orders ;
    INTO CURSOR csrOrders

LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab.prg"
    .nFunctionType = 2 && Count
    .cRowField = "EmployeeID"
    .cColField = "YEAR(OrderDate)"
    .cDataField = "OrderID"
    .cOutFile = "csrXtab"
    .lCursorOnly = .T.
    .lCloseTable = .T.
    .RunXtab()
ENDWITH
```


Figure 6. Here, the nFunctionType property was set to 2 to count the number of sales for each employee each year.

What if you want to know not just total sales or the number of sales for each salesperson for each year, but the total sales, the average sale and the number of sales? With FastXTab 1.6, you can specify multiple data columns and apply a different aggregation function to each. To specify multiple data columns, set nMultiDataField to the number of data columns, and then populate either anDataField or acDataField with the list of data columns. If you want different aggregation for different columns, fill anFunctionType as well. In **Listing 9** (included in the downloads for this session as SalesPersonAnnualSumAvgCnt.prg), nMultiDataField is set to 3. The first two columns use the same field from the source, OrderTotal, but each applies a different function. **Figure 7** shows the results. There are three columns for each year, one showing the total, one the average, and one the count.

Listing 9. FastXTab 1.6 lets you specify multiple data columns.

```
SELECT EmployeeID, ;
      Orders.OrderID, ;
      OrderDate, ;
      SUM(Quantity*UnitPrice) AS OrderTotal ;
```
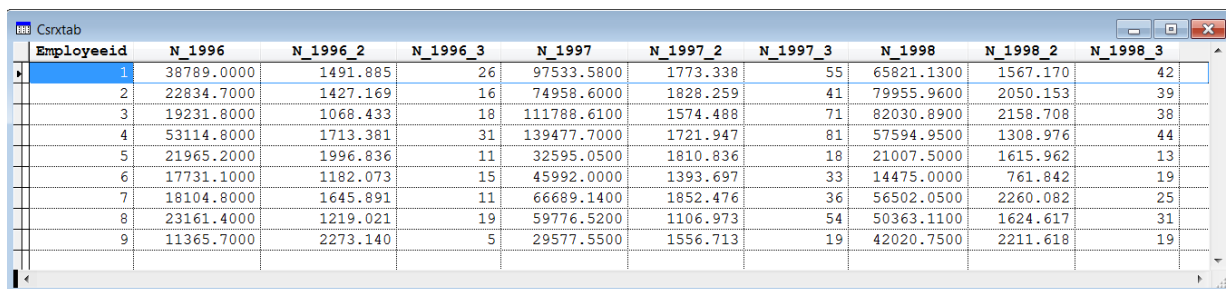
```
    FROM Orders ;
      JOIN OrderDetails ;
        ON Orders.OrderID = OrderDetails.OrderID ;
    GROUP BY 1, 2, 3 ;
    INTO CURSOR csrOrderTotals

LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
    .cRowField = 'EmployeeID'
    .cColField = 'YEAR(OrderDate)'
    .nMultiDataField = 3
    .acDataField[1] = 'OrderTotal'
    .anFunctionType[1] = 1
    .acDataField[2] = 'OrderTotal'
    .anFunctionType[2] = 3
    .anDataField[3] = 'OrderID'
    .anFunctionType[3] = 2
    .cOutFile = "csrXtab"
    .lCursorOnly = .T.
    .lCloseTable = .F.
    .RunXtab()
ENDWITH
```

Employeeid	N_1996	N_1996_2	N_1996_3	N_1997	N_1997_2	N_1997_3	N_1998	N_1998_2	N_1998_3
1	38789.0000	1491.885	26	97533.5800	1773.338	55	65821.1300	1567.170	42
2	22834.7000	1427.169	16	74958.6000	1828.259	41	79955.9600	2050.153	39
3	19231.8000	1068.433	18	111788.6100	1574.488	71	82030.8900	2158.708	38
4	53114.8000	1713.381	31	139477.7000	1721.947	81	57594.9500	1308.976	44
5	21965.2000	1996.836	11	32595.0500	1810.836	18	21007.5000	1615.962	13
6	17731.1000	1182.073	15	45992.0000	1393.697	33	14475.0000	761.842	19
7	18104.8000	1645.891	11	66689.1400	1852.476	36	56502.0500	2260.082	25
8	23161.4000	1219.021	19	59776.5200	1106.973	54	50363.1100	1624.617	31
9	11365.7000	2273.140	5	29577.5500	1556.713	19	42020.7500	2211.618	19

Figure 7.Using the nMultiDataField property, you can get multiple data columns for each value of the specified column field. Here, there are three columns for each year.

The previous examples showed that you can use an expression to specify the column field. In fact, you can use an expression for the row field, too and that expression can be fairly complex (in either case). **Listing 10** inverts the problem we've been looking at, putting employees in columns and time period in rows. In this case, each row specifies a quarter, using an expression that gives a result like 1998_Q1 (for the first quarter of 1998). **Figure 8** shows partial results. This query is included in the downloads for this session as SalesPersonColsQuarterly.PRG.

Listing 10. The expressions used to specify row and columns can be complex.

```
SELECT EmployeeID, ;
       Orders.OrderID, ;
       OrderDate, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
    FROM Orders ;
      JOIN OrderDetails ;
```

```
      ON Orders.OrderID = OrderDetails.OrderID ;
   GROUP BY 1, 2, 3 ;
   INTO CURSOR csrOrderTotals

LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
   .cRowField = 'PADL(YEAR(OrderDate),4) + "_Q" + PADL(QUARTER(OrderDate),1)'
   .cColField = 'EmployeeID'
   .nMultiDataField = 3
   .acDataField[1] = 'OrderTotal'
   .anFunctionType[1] = 1
   .acDataField[2] = 'OrderTotal'
   .anFunctionType[2] = 3
   .anDataField[3] = 'OrderID'
   .anFunctionType[3] = 2
   .cOutFile = "csrXtab"
   .lCursorOnly = .T.
   .lCloseTable = .F.
   .RunXtab()
ENDWITH
```

C_1996_q3	N_1	N_1_2	N_1_3	N_2	N_2_2	N_2_3	N_3	N_3_2	N
1996_Q3	14909.4000	1355.400	11	5940.8000	742.600	8	8317.4000	1188.200	
1996_Q4	23879.6000	1591.973	15	16893.9000	2111.738	8	10914.4000	992.218	
1997_Q1	15330.1000	1533.010	10	7639.3000	848.811	9	29658.6000	1560.979	
1997_Q2	15520.9000	1552.090	10	25667.5000	2566.750	10	34808.7500	2175.547	
1997_Q3	33578.4800	1865.471	18	19999.7500	1818.159	11	10586.4500	1058.645	
1997_Q4	33104.1000	1947.300	17	21652.0500	1968.368	11	36734.8100	1412.877	
1998_Q1	45146.4800	1556.775	29	45101.4100	2373.758	19	67731.9400	2418.998	
1998_Q2	20674.6500	1590.358	13	34854.5500	1742.728	20	14298.9500	1429.895	

Figure 8. Each row here represents a quarter and each set of three columns represents an employee.

Both versions of FastXTab support "pages," the ability to group rows. (Imagine each "page" as being a tab in a workbook.) With FastXTab 1.0, you specify the column that determines "pages" with the nPageField property. FastXTab 1.6's cPageField property lets you specify a field name or an expression if you prefer, just as you do for rows and columns. **Listing 11** (included in the downloads for this session as ProductsSold.PRG) shows how many units of each category were sold and shipped to each country each year; **Figure 9** shows partial results. The most obvious use for this capability is making grouping in a report simpler.

Listing 11. Use cPageField to specify an expression to "page" by in the crosstab.

```
SELECT ProductName, CategoryName, ;
      OrderDate, ShipCountry,  ;
      SUM(Quantity) AS NumSold ;
  FROM Orders ;
    JOIN OrderDetails OD ;
      ON Orders.OrderID = OD.OrderID ;
    JOIN Products ;
      ON OD.ProductID = Products.ProductID ;
    JOIN Categories ;
```

```
      ON Products.CategoryID = Categories.CategoryID ;
  GROUP BY 1, 2, 3, 4 ;
  INTO CURSOR csrProductsSold

LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
  .cPageField = 'ShipCountry'
  .cRowField = 'CategoryName'
  .cColField = 'YEAR(OrderDate)'
  .cDataField = 'NumSold'
  .cOutFile = "csrXtab"
  .lCursorOnly = .T.
  .lCloseTable = .F.
  .RunXtab()
ENDWITH
```

Shipcountry	Categoryname	N_1996	N_1997	N_1998
Argentina	Beverages	0	3	79
Argentina	Condiments	0	10	35
Argentina	Confections	0	29	28
Argentina	Dairy Products	0	3	51
Argentina	Grains/Cereals	0	0	20
Argentina	Produce	0	19	14
Argentina	Seafood	0	30	18
Austria	Beverages	188	335	459
Austria	Condiments	184	410	126
Austria	Confections	65	393	117
Austria	Dairy Products	212	430	385
Austria	Grains/Cereals	60	220	300
Austria	Meat/Poultry	14	283	65
Austria	Produce	99	201	88
Austria	Seafood	127	75	331
Belgium	Beverages	12	92	168
Belgium	Condiments	0	60	87
Belgium	Confections	40	123	107
Belgium	Dairy Products	65	110	120
Belgium	Grains/Cereals	0	67	78
Belgium	Meat/Poultry	40	14	35
Belgium	Produce	28	50	20

Figure 9. Here, each row represents one category for one country and each data column represents a year. The data is the number of units of that category shipped to that country in the specified year.

When you set nFunctionType to 6 (or set anFunctionType for a particular column to 6), you can specify a custom aggregation calculation. In **Listing 12**, we calculate the ratio of shipping cost (Freight) to the order total for each month for each customer. **Figure 10** shows partial results; the code is included in the downloads for this session as FreightRatio.PRG.

Listing 12. You can set nFunctionType to 6 and cFunctionExp to a custom aggregation calculation. Here, it's the ratio of shipping cost to order total.

```
SELECT CustomerID, ;
       Orders.OrderID, ;
       OrderDate, ;
```
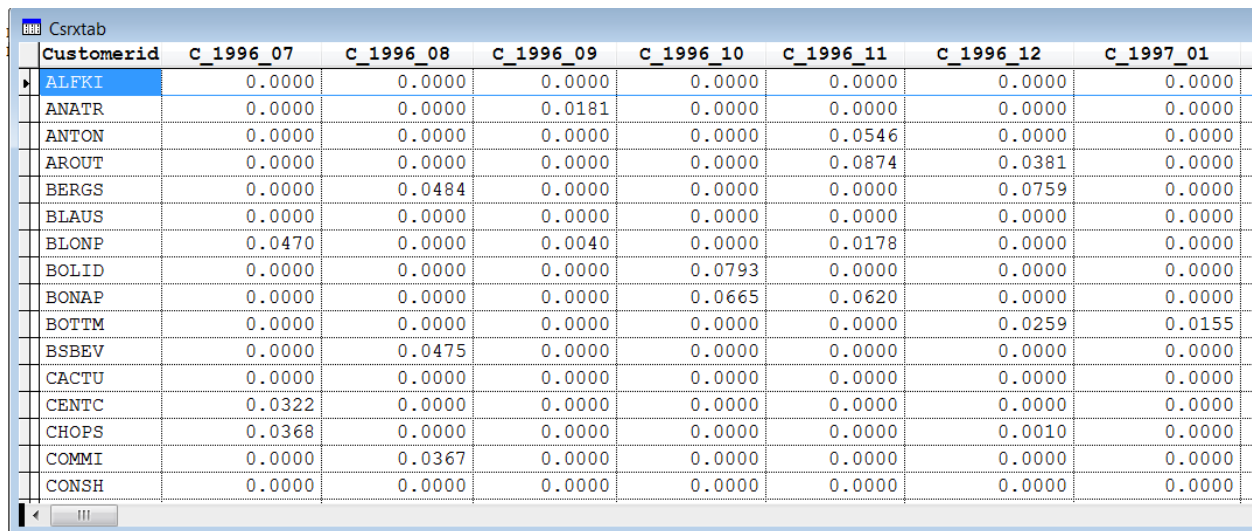
```
        SUM(Quantity*UnitPrice) AS OrderTotal,;
        Freight ;
  FROM Orders ;
    JOIN OrderDetails OD ;
      ON Orders.OrderID = OD.OrderID ;
  GROUP BY 1, 2, 3, 5 ;
  INTO CURSOR csrOrderTotals

LOCAL nStart, nEnd
LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
    .cRowField = 'CustomerID'
    .cColField = 'PADL(YEAR(OrderDate),4)' + ' + "_" + PADL(MONTH(OrderDate),2,"0")'
    .nFunctionType = 6
    .cFunctionExp = 'SUM(Freight)/SUM(OrderTotal)'
  .cOutFile = "csrXtab"
  .lCursorOnly = .T.
  .lCloseTable = .T.
  .RunXtab()
ENDWITH
```

Customerid	C_1996_07	C_1996_08	C_1996_09	C_1996_10	C_1996_11	C_1996_12	C_1997_01
ALFKI	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ANATR	0.0000	0.0000	0.0181	0.0000	0.0000	0.0000	0.0000
ANTON	0.0000	0.0000	0.0000	0.0000	0.0546	0.0000	0.0000
AROUT	0.0000	0.0000	0.0000	0.0000	0.0874	0.0381	0.0000
BERGS	0.0000	0.0484	0.0000	0.0000	0.0000	0.0759	0.0000
BLAUS	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
BLONP	0.0470	0.0000	0.0040	0.0000	0.0178	0.0000	0.0000
BOLID	0.0000	0.0000	0.0000	0.0793	0.0000	0.0000	0.0000
BONAP	0.0000	0.0000	0.0000	0.0665	0.0620	0.0000	0.0000
BOTTM	0.0000	0.0000	0.0000	0.0000	0.0000	0.0259	0.0155
BSBEV	0.0000	0.0475	0.0000	0.0000	0.0000	0.0000	0.0000
CACTU	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
CENTC	0.0322	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
CHOPS	0.0368	0.0000	0.0000	0.0000	0.0000	0.0010	0.0000
COMMI	0.0000	0.0367	0.0000	0.0000	0.0000	0.0000	0.0000
CONSH	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Figure 10. The data here is the ratio of freight cost to order total for a customer for a month.

FastXTab 1.6 lets you filter the data both before and after it's aggregated. Use cCondition to filter before aggregating, and cHaving to filter afterward.
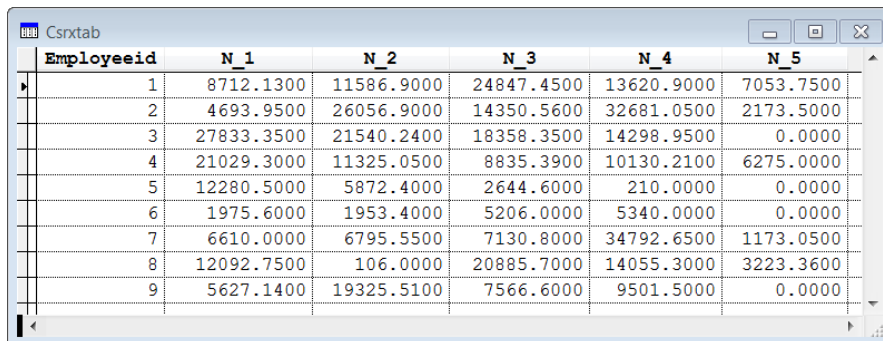
In **Listing 13**, we retrieve data on all orders, but use only those from one year in the crosstab, which shows totals for each salesperson for each month. You might use this approach in a loop to generate a crosstab for each year. **Figure 11** shows the results. The code is included in the downloads for this session as SalesPersonMonthlyFilter.PRG.

Listing 13. The cCondition property filters data out of the cursor or table supplied to FastXTab 1.6. Here, we keep only one year's data.

```
SELECT EmployeeID, ;
       OrderDate, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
  FROM Orders ;
    JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
  GROUP BY 1, 2 ;
  INTO CURSOR csrMonthlyTotals

LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
    .cRowField = 'EmployeeID'
    .cColField = 'MONTH(OrderDate)'
    .cDataField = 'OrderTotal'
  .cCondition = 'YEAR(OrderDate) = 1998'
  .cOutFile = "csrXtab"
  .lCursorOnly = .T.
  .lCloseTable = .T.
  .RunXtab()
ENDWITH
```

Employeeid	N_1	N_2	N_3	N_4	N_5
1	8712.1300	11586.9000	24847.4500	13620.9000	7053.7500
2	4693.9500	26056.9000	14350.5600	32681.0500	2173.5000
3	27833.3500	21540.2400	18358.3500	14298.9500	0.0000
4	21029.3000	11325.0500	8835.3900	10130.2100	6275.0000
5	12280.5000	5872.4000	2644.6000	210.0000	0.0000
6	1975.6000	1953.4000	5206.0000	5340.0000	0.0000
7	6610.0000	6795.5500	7130.8000	34792.6500	1173.0500
8	12092.7500	106.0000	20885.7000	14055.3000	3223.3600
9	5627.1400	19325.5100	7566.6000	9501.5000	0.0000

Figure 11. This crosstab shows sales for each employee by month for the year specified in the cCondition property.

To demonstrate cHaving, we'll extend the freight ratio example. Perhaps you're interested in seeing only cases where customers seem to be spending too much on freight charges. Add the line in **Listing 14** inside the WITH clause in **Listing 12** to see only cases where a customer's total monthly freight charges were more than 5% of the total orders. This version of the example is included as FreightRatioFiltered.PRG in the downloads for this session.

Listing 14. The cHaving property filters after aggregation. Here, it keeps only data where the ratio of freight to order total (for the month) is more than 5%.

```
    .cHaving = 'SUM(Freight)/SUM(OrderTotal) >= 0.05'
```

Which one to use?

It should be obvious that I recommend FastXTab 1.6 over VFPXTab or FastXTab 1.0. FastXTab 1.6 has some additional capabilities not discussed in this paper. If you use crosstabs (or if you now see how you can use them), I strongly recommend spending some time not only with the examples above, but with the ones that Vilhelm-Ion Praisach provides.

Generating crosstabs (pivots) in SQL

In SQL Server, the term used for crosstabs is pivots and the ability to create them is built in. No added tools are needed.

As with VFP, let's start with a simple example to demonstrate the need for pivots. Suppose you want to know how many employees AdventureWorks has in each country for each job title. The query in **Listing 15** answers the question, but the form of the result (partially shown in **Figure 12**) makes it hard to grasp. The query is included as JobTitleByCountry.SQL in the downloads for this session.

Listing 15. This query provides the number of employees with each job title in each country, but each record represents one job title/country combination.

```
SELECT JobTitle, CR.Name, Count(*) AS EmpCount
  FROM [HumanResources].[Employee]
    JOIN [Person].[BusinessEntityAddress] BEA
      ON Employee.BusinessEntityID =
         BEA.BusinessEntityID
    JOIN [Person].[Address]
      ON BEA.AddressID = Address.AddressID
    JOIN [Person].[StateProvince] SP
      ON Address.StateProvinceID =
         SP.StateProvinceID
    JOIN [Person].[CountryRegion] CR
      ON SP.CountryRegionCode =
         CR.CountryRegionCode
  WHERE Employee.CurrentFlag = 1
  GROUP BY JobTitle, CR.Name
  ORDER BY JobTitle, CR.Name
```

Recruiter	United States	2
Research and Development Engineer	United States	2
Research and Development Manager	United States	2
Sales Representative	Australia	1
Sales Representative	Canada	2
Sales Representative	France	1
Sales Representative	Germany	1
Sales Representative	United King…	1
Sales Representative	United States	8
Scheduling Assistant	United States	4
Senior Design Engineer	United States	1
Senior Tool Designer	United States	2

Figure 12. Each row here shows the number of employees with the specified job title in the specified country.

A better format would have one column for each country and one row for each job title, with the intersection of the two containing the number of employees in that country with that job title. As in VFP, one way to get this result, especially when the number of countries is small, is to use SUM() to do the counting, though in T-SQL, the keyword that makes it work is CASE rather than IIF. **Listing 16**, included in the downloads for this session as JobTitleByCountryCase.SQL, shows code to do it this way. **Figure 13** shows partial results, much easier to interpret than the previous version.

Listing 16. You can create a simple crosstab using CASE to break out the individual columns.

```
SELECT JobTitle,
       SUM(CASE CR.Name WHEN 'Australia'
           THEN 1 ELSE 0 END) AS nAustralia,
       SUM(CASE CR.Name WHEN 'Canada'
           THEN 1 ELSE 0 END) AS nCanada,
       SUM(CASE CR.Name WHEN 'France'
           THEN 1 ELSE 0 END) AS nFrance,
       SUM(CASE CR.Name WHEN 'Germany'
           THEN 1 ELSE 0 END) AS nGermany,
       SUM(CASE CR.Name WHEN 'United Kingdom'
           THEN 1 ELSE 0 END) AS nUK,
       SUM(CASE CR.Name WHEN 'United States'
           THEN 1 ELSE 0 END) AS nUSA
  FROM [HumanResources].[Employee]
    JOIN [Person].[BusinessEntityAddress] BEA
      ON Employee.BusinessEntityID =
         BEA.BusinessEntityID
    JOIN [Person].[Address]
      ON BEA.AddressID = Address.AddressID
    JOIN [Person].[StateProvince] SP
      ON Address.StateProvinceID =
         SP.StateProvinceID
    JOIN [Person].[CountryRegion] CR
      ON SP.CountryRegionCode =
         CR.CountryRegionCode
  WHERE Employee.CurrentFlag = 1
  GROUP BY JobTitle
  ORDER BY JobTitle
```

JobTitle	nAustralia	nCanada	nFrance	nGermany	nUK	nUSA
Purchasing Manager	0	0	0	0	0	1
Quality Assurance Manager	0	0	0	0	0	1
Quality Assurance Supervisor	0	0	0	0	0	1
Quality Assurance Technician	0	0	0	0	0	4
Recruiter	0	0	0	0	0	2
Research and Development Engineer	0	0	0	0	0	2
Research and Development Manager	0	0	0	0	0	2
Sales Representative	1	2	1	1	1	8
Scheduling Assistant	0	0	0	0	0	4
Senior Design Engineer	0	0	0	0	0	1
Senior Tool Designer	0	0	0	0	0	2
Shipping and Receiving Clerk	0	0	0	0	0	2
Shipping and Receiving Supervisor	0	0	0	0	0	1

Figure 13. Using CASE with SUM() gives one column per country and makes the results more readable.

But T-SQL offers an easier way to do this.

Introducing PIVOT

The PIVOT operator provides a way to crosstab without having to write out all the CASE expressions. PIVOT goes into the FROM clause of the query. **Listing 17** shows the syntax for using PIVOT.

In my experience, this is a case where it's easiest to use "*" rather than listing specific field names. The source table can be an actual table, a derived table, or a table created as part of a CTE (common table expression).

Listing 17. The PIVOT operator appears in the FROM clause of a query and specifies an aggregation function.

```
SELECT <non-pivoted column>,
       <list of pivoted columns with aliases>
FROM <source table>
PIVOT
(<aggregation function>(<column to aggregate>)
  FOR [<column name column>]
    IN (<list of values>)
) AS <alias for the pivot table>
```

The interesting part is what goes after the PIVOT keyword. First, you need an aggregation function, such as SUM(OrderTotal). After FOR, you list the name of the source column whose values are to become columns in the result. In the job title by country example, that's the Country column.

Finally, after IN, you have to include a list of all the values of interest. Having an explicit list is both a good thing and a bad thing. It's a good thing because it allows you to include only a subset of the values from the relevant column. It's a bad thing, of course, because it requires you to know the list of values from that column. (I'll show you how to avoid providing an explicit list in "Handling unknown data," later in this paper.)

Listing 18 shows a query using PIVOT that produces the same results as the query in **Listing 16**. A CTE collects the list of employees with their job titles and countries. The main query uses PIVOT to count the number of employees by country. The CTE has three columns: JobTitle, Country and EmpID. The main query specifies that all three are in the result (SELECT *), but the PIVOT clause indicates that Country determines the columns (that is, the column headers are the actual values from Country), and that EmpID is aggregated, in this case, by counting. **Figure 14** shows partial results. The query is included in the downloads for this session as JobTitleByCountryPivot.SQL.

Listing 18. This query pivots on country to produce one record per job title with a column for each country where any employees are located.

```
WITH csrJobCountry
  (JobTitle, Country, EmpID)
AS
(SELECT JobTitle, CR.Name,
```

```
        Employee.BusinessEntityID
  FROM [HumanResources].[Employee]
    JOIN [Person].[BusinessEntityAddress] BEA
      ON Employee.BusinessEntityID =
         BEA.BusinessEntityID
    JOIN [Person].[Address]
      ON BEA.AddressID = Address.AddressID
    JOIN [Person].[StateProvince] SP
      ON Address.StateProvinceID =
         SP.StateProvinceID
    JOIN [Person].[CountryRegion] CR
      ON SP.CountryRegionCode =
         CR.CountryRegionCode
  WHERE Employee.CurrentFlag = 1
  )

SELECT *
  FROM csrJobCountry
    PIVOT(COUNT(EmpID)
    FOR Country
    IN (Australia, Canada, France, Germany,
        [United Kingdom], [United States]))
    AS EmpTotal
```

JobTitle	Australia	Canada	France	Germany	United Kingdom	United States
Purchasing Manager	0	0	0	0	0	1
Quality Assurance Manager	0	0	0	0	0	1
Quality Assurance Supervisor	0	0	0	0	0	1
Quality Assurance Technician	0	0	0	0	0	4
Recruiter	0	0	0	0	0	2
Research and Development Engineer	0	0	0	0	0	2
Research and Development Manager	0	0	0	0	0	2
Sales Representative	1	2	1	1	1	8
Scheduling Assistant	0	0	0	0	0	4
Senior Design Engineer	0	0	0	0	0	1
Senior Tool Designer	0	0	0	0	0	2
Shipping and Receiving Clerk	0	0	0	0	0	2
Shipping and Receiving Supervisor	0	0	0	0	0	1

Figure 14. The results here are the same as in Figure 13, except for the column headers, which are the actual country names from the CountryRegion table.

I suspect that the most commonly used function in the pivot is SUM, letting you see some kind of total across a set of time periods or regions or other way of dividing up data. For example, **Listing 19** produces total sales for each salesperson for each year; **Figure 15** shows partial results. This query is included in the downloads for this session as SalesPersonAnnualSalesCTE.SQL.

Listing 19. This query computes total sales for each salesperson for each year.

```
WITH SalesByYear
  (SalesPersonID, SalesYear, SubTotal)
AS
(SELECT SalesPersonID, YEAR(OrderDate),
        SubTotal
```

```
     FROM Sales.SalesOrderHeader
     WHERE SalesPersonID IS NOT NULL)

SELECT *
  FROM SalesByYear
  PIVOT(SUM(SubTotal)
    FOR SalesYear
     IN ([2011], [2012], [2013], [2014]))
   AS TotalSales
  ORDER BY SalesPersonID
```

SalesPersonID	2011	2012	2013	2014
274	28926.2465	453524.5233	431088.7238	178584.3625
275	875823.8318	3375456.8947	3985374.8995	1057247.3786
276	1149715.3253	3834908.674	4111294.9056	1271088.5216
277	1311627.2918	4317306.5741	3396776.2674	1040093.4071
278	500091.8202	1283569.6294	1389836.8101	435948.9551
279	1521289.1881	2674436.3518	2188082.7813	787204.4289
280	648485.5862	1208264.3834	963420.5805	504932.044
281	967597.2899	2294210.5506	2387256.0616	777941.6519
282	1175007.4753	1835715.8705	1870884.182	1044810.8277
283	599987.9444	1288068.7236	1351422.362	490466.319
284	NULL	441639.5961	1269908.9235	600997.1704
285	NULL	NULL	151257.1152	21267.336
286	NULL	NULL	836055.1236	585755.8006
287	NULL	116029.652	560091.7843	56637.7478

Figure 15. Each row here represents one salesperson, while each column represents a year. The intersection shows the dollar total of sales for that salesperson for that year.

In this example, again, the CTE includes exactly three columns. One (SalesPersonID) determines the rows, one (SalesYear) determines the columns, and one (SubTotal) is aggregated to produce the data values.

Of course, this data would be more useful with the salespeople's names as well as their IDs. You can turn the query with the PIVOT into a CTE and add the names afterward, as in **Listing 20** (which is included as SalesPersonAnnualSalesWithNameCTE.SQL in the downloads for this session. Partial results are shown in **Figure 16**.

Listing 20. A query that uses PIVOT can be a CTE, so you can add more data.

```
WITH SalesByYear
  (SalesPersonID, SalesYear, SubTotal)
AS
(SELECT SalesPersonID, YEAR(OrderDate) ,
        SubTotal
   FROM Sales.SalesOrderHeader
   WHERE SalesPersonID IS NOT NULL),

SalesByYearPivot
AS
(SELECT *
   FROM SalesByYear
   PIVOT(SUM(SubTotal)
```

```
    FOR SalesYear
      IN ([2011], [2012], [2013], [2014]))
    AS TotalSales)

SELECT Person.FirstName, Person.LastName,
       SalesByYearPivot.*
  FROM SalesByYearPivot
    JOIN Person.Person
    ON SalesByYearPivot.SalesPersonID =
       Person.BusinessEntityID
  ORDER BY LastName, FirstName
```

FirstName	LastName	SalesPersonID	2011	2012	2013	2014
Syed	Abbas	285	NULL	NULL	151257.1152	21267.336
Amy	Alberts	287	NULL	116029.652	560091.7843	56637.7478
Pamela	Ansman-Wolfe	280	648485.5862	1208264.3834	963420.5805	504932.044
Michael	Blythe	275	875823.8318	3375456.8947	3985374.8995	1057247.3786
David	Campbell	283	599987.9444	1288068.7236	1351422.362	490466.319
Jillian	Carson	277	1311627.2918	4317306.5741	3396776.2674	1040093.4071
Shu	Ito	281	967597.2899	2294210.5506	2387256.0616	777941.6519
Stephen	Jiang	274	28926.2465	453524.5233	431088.7238	178584.3625
Tete	Mensa-Annan	284	NULL	441639.5961	1269908.9235	600997.1704
Linda	Mitchell	276	1149715.3253	3834908.674	4111294.9056	1271088.5216
Jae	Pak	289	NULL	3014278.0472	4106064.0146	1382996.5839
Tsvi	Reiter	279	1521289.1881	2674436.3518	2188082.7813	787204.4289

Figure 16. Salesperson names are added to this pivoted result by putting the pivot into a CTE.

Getting meaningful column names

By default, the list you include in the IN portion of PIVOT determines the names of pivoted columns. So, in the sales example, the columns are called 2011, 2012, etc., while in the jobs example, they're the names of the countries. (This also explains why numeric values or values containing spaces need to be surrounded by square brackets; that's the standard way of referring to a column that has a name that can't stand alone.)

However, you can specify alternative names for these columns in the field list of the query, just as you can for any field. The query in **Listing 21** pulls sales data for one year and then pivots on month. The field list changes the names for those columns from the numeric month to the standard abbreviation. **Figure 17** shows partial results, and the query is included as SalesPerson2013MonthlySalesWithMonthNames.SQL in the downloads for this session.

Listing 21. You can rename pivoted columns in the field list of the query.

```
WITH SalesByMonth
AS
(SELECT SalesPersonID,
        MONTH(OrderDate) As SalesMonth,
        SubTotal
   FROM Sales.SalesOrderHeader
   WHERE SalesPersonID IS NOT NULL
     AND YEAR(OrderDate) = 2013)
```

```
SELECT SalesPersonID,
       [1] AS Jan, [2] AS Feb, [3] AS Mar,
       [4] AS Apr, [5] AS May, [6] AS Jun,
       [7] AS Jul, [8] AS Aug, [9] AS Sep,
       [10] AS Oct, [11] AS Nov, [12] AS Dec
  FROM SalesByMonth
  PIVOT(SUM(SubTotal)
    FOR SalesMonth
     IN ([1], [2], [3], [4], [5], [6], [7],
         [8], [9], [10], [11], [12]))
    AS TotalSales
  ORDER BY SalesPersonID;
```

SalesPersonID	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug
274	NULL	43254.2036	5255.3088	1466.01	NULL	129426.5658	88118.9333	1946.022
275	260648.3902	314936.4504	376270.9093	327588.3495	248292.2912	449798.1591	529975.9896	222182.8369
276	164516.324	88379.2611	614957.4404	263161.852	308192.3055	659607.3976	333022.1095	208571.1413
277	186124.981	400651.4944	383608.9199	277065.913	262105.7294	345113.6639	415660.7059	242981.9756
278	68720.8323	8091.5083	214520.8152	80733.1441	16659.9281	270339.3915	176996.7345	13416.924
279	125988.2839	172527.4835	212608.3495	148977.4823	170875.4565	274576.6757	210465.0176	153249.8817
280	34427.3177	49152.4316	NULL	32195.7427	112336.0762	60425.9465	184786.5098	112009.8219
281	186406.7991	87934.131	194265.9328	238055.2337	354330.5892	129124.5308	262870.0206	305661.8575
282	115841.3487	47113.0052	69036.5755	79163.3631	152484.4903	192678.1351	462619.6221	116094.1021
283	4244.1186	155124.1524	106704.8744	2802.5973	172106.6824	183419.7065	174135.1983	117742.353
284	30335.6999	9479.9522	219048.6425	35479.6027	98549.9789	124466.0599	169425.3874	97920.6609
285	NULL	NULL	NULL	NULL	NULL	NULL	111924.9312	NULL
286	NULL	NULL	NULL	NULL	49824.7139	134280.9823	154853.7768	55341.462
287	1308.9375	968.4284	43180.9774	51365.2568	3711.966	106126.4301	35381.8571	323.994
288	NULL	NULL	NULL	NULL	178101.5456	193871.6511	127080.5935	193828.4706

Figure 17. One year's sales were pivoted by month. Then, the field names were replaced by something more meaningful.

This query also shows why it's generally easier to use SELECT * in a PIVOT. Otherwise, you need to list each pivoted column by name.

Determining rows by multiple columns

In the examples above, the set of rows was determined by a single field, JobTitle in the first case and SalesPersonID in the second. But it's possible to use multiple fields to specify the rows. All you have to do is have multiple columns in the query that aren't listed in the PIVOT clause.

For example, the query in **Listing 22** has one row for each salesperson for each month. The CTE result has four fields: salesperson ID, month, year and order amount (Subtotal). The main query totals the order amount and specifies that year determines the columns. That leaves both salesperson ID and month to specify the rows. Partial results are shown in **Figure 18**. The query is included as SalesPersonMonthlySales.SQL in the downloads for this session.

Listing 22. This query uses two fields (SalesPersonID and SalesMonth) to specify the rows in the pivoted result.

```
WITH csrSalesByYear
```

```
AS
(SELECT SalesPersonID,
        MONTH(OrderDate) As SalesMonth,
        YEAR(orderDate) AS SalesYear,
        SubTotal
   FROM Sales.SalesOrderHeader
   WHERE SalesPersonID IS NOT NULL)

SELECT *
   FROM csrSalesByYear
   PIVOT(SUM(SubTotal)
     FOR SalesYear
       IN ([2011], [2012], [2013], [2014]))
     AS TotalSales
   ORDER BY SalesPersonID, SalesMonth;
```

SalesPersonID	SalesMonth	2011	2012	2013	2014
274	1	NULL	79514.2242	NULL	1414.248
274	2	NULL	33406.7043	43254.2036	NULL
274	3	NULL	NULL	5255.3088	139517.1925
274	4	NULL	44670.6854	1466.01	NULL
274	5	NULL	3575.7202	NULL	37652.922
274	6	NULL	55616.5989	129426.5658	NULL
274	7	20544.7015	523.788	88118.9333	NULL
274	8	2039.994	56210.9496	1946.022	NULL
274	9	NULL	2709.6518	90806.321	NULL
274	10	6341.551	79994.1743	NULL	NULL
274	11	NULL	NULL	70815.3593	NULL
274	12	NULL	97302.0266	NULL	NULL
275	1	NULL	283832.0699	260648.3902	248125.5411
275	2	NULL	143767.8366	314936.4504	NULL
275	3	NULL	172429.5757	376270.9093	439114.8552

Figure 18. Here, the pivot result uses two columns to distinguish the rows.

Aggregating on more than one column

A more complicated problem is computing more than one aggregate result. For example, suppose you want to get both total sales and the number of sales by year for each salesperson. You might think that you could simply list multiple aggregate functions after PIVOT, but that doesn't work.

In fact, to include multiple pivoted aggregations, you have to perform the pivots separately and then join the results. You also have to make sure that whatever you're selecting from contains only the columns relevant to that particular aggregation.

The easiest way to do this is with a series of CTEs, as in **Listing 23**. The first two CTEs, SalesByYear and SalesTotal, are the same as previous examples, producing one row per salesperson with one column per year. The final CTE, SalesCount, produces one row per salesperson with one column per year containing the number of orders for that salesperson in that year. Finally, the main query joins SalesTotal and SalesCount on SalesPersonID, including all the pivoted columns from each of them. **Figure 19** shows

partial results. This query is included as SalesPersonAnnualSalesMulti.SQL in the downloads for this session.

Listing 23.To pivot and aggregate on multiple columns, you have to do each pivot separately, and then join the results.

```
WITH SalesByYear
   (SalesPersonID, SalesYear, SubTotal)
AS
(SELECT SalesPersonID,
        YEAR(OrderDate),
        SubTotal
   FROM Sales.SalesOrderHeader
   WHERE SalesPersonID IS NOT NULL),

SalesTotal
AS
(SELECT SalesPersonID,
        [2011] AS Total2011,
        [2012] AS Total2012,
        [2013] AS Total2013,
        [2014] AS Total2014
   FROM SalesByYear
   PIVOT(SUM(SubTotal)
     FOR SalesYear
       IN ([2011], [2012], [2013], [2014]))
     AS TotalSales),

SalesCount
AS
(SELECT SalesPersonID,
        [2011] AS Count2011,
        [2012] AS Count2012,
        [2013] AS Count2013,
        [2014] AS Count2014
   FROM SalesByYear
   PIVOT(COUNT(SubTotal)
     FOR SalesYear
       IN ([2011], [2012], [2013], [2014]))
     AS Sales)

SELECT ST.SalesPersonID,
       SC.Count2011, ST.Total2011,
       SC.Count2012, ST.Total2012,
       SC.Count2013, ST.Total2013,
       SC.Count2014, ST.Total2014
   FROM SalesTotal ST
     JOIN SalesCount SC
       ON ST.SalesPersonID = SC.SalesPersonID
   ORDER BY ST.SalesPersonID
```

SalesPersonID	Count2011	Total2011	Count2012	Total2012	Count2013	Total2013	Count2014	Total2014
274	4	28926.2465	22	453524.5233	14	431088.7238	8	178584.3625
275	65	875823.8318	148	3375456.8947	175	3985374.8995	62	1057247.3786
276	46	1149715.3253	151	3834908.674	162	4111294.9056	59	1271088.5216
277	59	1311627.2918	166	4317306.5741	185	3396776.2674	63	1040093.4071
278	30	500091.8202	80	1283569.6294	89	1389836.8101	35	435948.9551
279	63	1521289.1881	153	2674436.3518	159	2188082.7813	54	787204.4289
280	22	648485.5862	45	1208264.3834	19	963420.5805	9	504932.044
281	33	967597.2899	74	2294210.5506	98	2387256.0616	37	777941.6519
282	56	1175007.4753	86	1835715.8705	86	1870884.182	43	1044810.8277
283	28	599987.9444	63	1288068.7236	72	1351422.362	26	490466.319
284	0	NULL	24	441639.5961	82	1269908.9235	34	600997.1704
285	0	NULL	0	NULL	12	151257.1152	4	21267.336

Figure 19. By joining the results of two separate pivots, we can do two different aggregations.

In my initial attempts at doing this (because, for some reason, I mistakenly thought that doing COUNT(Subtotal) would count only distinct values), I tried using a single CTE containing both Subtotal and SalesOrderID as the source for both pivots. However, even though the unneeded field was omitted from the field list of the queries performing the pivots, the field was still used in determining the rows of the result. Every field in the source table for a pivot is used either in determining rows, determining columns, or aggregation. The query in **Listing 24** demonstrates the issue. The CTE includes SalesOrderID, though it's not mentioned in the main query. Nonetheless, the results (partially shown in **Figure 20**) have one row per sales order rather than one row per salesperson. This faulty query is included in the downloads for this session as SalesPersonAnnualExtraField.SQL

Listing 24. Every field in the table specified for a pivot is used somehow. If it's not otherwise specified, it helps determine the list of rows.

```
WITH SalesByYear
  (SalesPersonID, SalesYear, SubTotal, OrderID)
AS
(SELECT SalesPersonID, YEAR(OrderDate), SubTotal, SalesOrderID
   FROM Sales.SalesOrderHeader
  WHERE SalesPersonID IS NOT NULL)

SELECT SalesPersonID,
       [2011] AS Total2011,
       [2012] AS Total2012,
       [2013] AS Total2013,
       [2014] AS Total2014
  FROM SalesByYear
  PIVOT(SUM(SubTotal)
    FOR SalesYear
      IN ([2011], [2012], [2013], [2014])) AS TotalSales
```

SalesPersonID	Total2011	Total2012	Total2013	Total2014
279	20565.6206	NULL	NULL	NULL
279	1294.2529	NULL	NULL	NULL
282	32726.4786	NULL	NULL	NULL
282	28832.5289	NULL	NULL	NULL
276	419.4589	NULL	NULL	NULL
280	24432.6088	NULL	NULL	NULL
283	14352.7713	NULL	NULL	NULL
276	5056.4896	NULL	NULL	NULL
277	6107.082	NULL	NULL	NULL
282	35944.1562	NULL	NULL	NULL
283	714.7043	NULL	NULL	NULL
275	6122.082	NULL	NULL	NULL
283	8128.7876	NULL	NULL	NULL

Figure 20. Because the table used for this pivot includes SalesOrderID, the result has one row per sales order, rather than just one per salesperson.

Handling unknown data

The biggest issue to me with these examples is the need to list out the values in the field that determines the result columns. It means you must know what data to expect and more importantly, that query results may be wrong if there's data you didn't expect.

Fortunately, there's a solution, using dynamic SQL. You can store a query in a string and then execute it; it's similar to using the & macro operator or the ExecScript() function in VFP. Full details on dynamic SQL are beyond the scope of this paper, but you don't have to know much to use it for pivoting without knowing the values of the field you want to pivot on. I'll cover it in the context of the example in **Listing 19**, seeing total sales by salesperson for each year; the complete code is in **Listing 25** and included in the downloads for this session as SalesPersonAnnualSalesDynamic.SQL.

You need two variables, one to hold the list of values and one to hold the query you construct. In the example, they're called @years and @query. Step one is to run a query to store the list of distinct values in the first variable. It takes advantage of SQL's ability to populate a variable via a query. The QUOTENAME() function converts to character (actually varchar) and adds delimiters to make sure the result can serve as an identifier. In this example, @years is assigned the value '[2011],[2012],[2013],[2014]', that is, exactly the list we've been hard-coding in the previous examples.

Step two is to build a string that contains the query we want to execute. The SET command here simply fills @query with the same query we've been using, except that instead of hard-coding the list of years, we plug in the computed value in @years.

Finally, we call the built-in stored procedure sp_executesql to execute the query we've built. The first parameter to sp_executesql is always the statement to execute. It can accept additional parameters to be used in executing that statement, but in this case, we don't need any.

Listing 25. When you don't know the list of values for the column you want to pivot on, you can retrieve a list of values and use dynamic SQL to do the actual pivot.

```
DECLARE @query AS NVARCHAR(max);
DECLARE @years AS NVARCHAR(max);

-- Get the list of years
WITH DistinctYear (nYear)
AS
(SELECT DISTINCT YEAR(OrderDate)
  FROM Sales.SalesOrderHeader)

SELECT @years = ISNULL(@years + ', ', '') + QUOTENAME(nYear)
  FROM DistinctYear
  ORDER BY nYear;

-- Build the query including the list of years
SET @query =
  'WITH SalesByYear
   AS
   (SELECT SalesPersonID,
           YEAR(OrderDate) AS SalesYear,
           SubTotal
      FROM Sales.SalesOrderHeader
      WHERE SalesPersonID IS NOT NULL)

   SELECT *
    FROM SalesByYear
     PIVOT(SUM(SubTotal)
       FOR SalesYear IN (' + @years + '))
       AS TotalSales
    ORDER BY SalesPersonID';

-- Run the query
EXEC sp_executesql @query;
```

Not surprisingly, this example produces the same results as the one where the years are hard-coded; they're shown in **Figure 15**.

The query that produces the list of years deserves a little more attention, since it does something you can't do with a single query in VFP. First, as noted above, rather than storing its result in some kind of table, it puts the result into a variable (@years); that's what `SELECT @years` = does. The more interesting piece is that @years appears on the right-hand side of the equals sign, as well. So, the result is built up one record at a time. For the first record, @years is null, and `ISNULL(@years + ', ', '')` returns the empty string. After that, it returns the string so far with a trailing comma and space. For each record, we then add the bracketed version of the year.

The one big advantage of having to list each value in PIVOT's IN section is that you can limit the result to a specified subset of the data. To do that when generating the list of values dynamically, there has to be a rule you can apply in the query that assembles the list of values. For example, if you're only interested in sales in 2013 and later, you can add a

WHERE clause to the CTE of the query that populates @years, as in **Listing 26**. If you want to see sales for the last three years, you could specify YEAR(GETDATE())-2 rather than 2013 in the WHERE clause.

Listing 26. You can limit the column list by filtering the query that collects the list of values.

```
-- Get the list of years
WITH DistinctYear (nYear)
AS
(SELECT DISTINCT YEAR(OrderDate)
  FROM Sales.SalesOrderHeader
  WHERE YEAR(OrderDate) >= 2013)

SELECT @years = ISNULL(@years + ', ', '') + QUOTENAME(nYear)
  FROM DistinctYear
  ORDER BY nYear;
```

Undoing PIVOT

SQL Server lets you undo pivots through the UNPIVOT keyword, though you can't always get back to the original data (because of the aggregation performed as part of the pivot process). The syntax for UNPIVOT is quite similar to the syntax for PIVOT; it's shown in **Listing 27**.

Listing 27. The syntax for UNPIVOT is pretty much the same as for PIVOT, except there's no aggregation function involved.

```
SELECT <row identifier columns>,
       [ <column identifier column>, ]
       <column to extract>
FROM <source table>
UNPIVOT
(<column to extract>
  FOR [<generic name for group of columns>]
    IN (<list of columns to unpivot>)
) AS <alias>
```

For UNPIVOT, you generally list out the columns you want. Depending on the data you're unpivoting, you may or may not want to turn the column names into data. (That's the optional "<column identifier column>" in the syntax.) In addition, the columns you want in the result can be listed in any order.

The key items are to provide a name for the column to contain the data you're unpivoting (shown as "<column to extract>" in the syntax diagram), and to provide a list of the columns that contain data following the IN keyword. The latter is the same information you provide to PIVOT, but here it's the column names rather than the values in a particular column. The item that follows the FOR keyword is simply a name to let you refer to that list as a group; that lets you include them as data in the result. As you'll see later in this paper, it also allows you to operate on those column names as data.

Let's start with a simple example that (sort of) reverses the annual sales pivot. Assume that rather than simply returning the result of the query in **Listing 19**, we've stored it in a temporary table called #SalesByYearCT. **Listing 28** shows how to unpivot that result and get one row per salesperson per year; it's included (with the code to create #SalesByYearCT) in the downloads for this session as UnpivotSales.SQL. The query lists the three fields we want in the result: the salesperson's ID, the year, and the total sales for that salesperson in that year. Inside the UNPIVOT section, we first specify that the data in the columns we're unpivoting should go into a column called AnnualSales. Then, we indicate that those columns are the ones named [2011], [2012], [2013] and [2014]. The name SalesYear is assigned to that group of fields; it matches up with the name in the field list, so that the year appears in the result. The alias for the UNPIVOT (SalesPersonYear) is required but doesn't actually add anything.

Listing 28. The query sort of unpivots the result of the query in **Listing 19**.

```
SELECT SalesPersonID, SalesYear, AnnualSales
  FROM #SalesByYearCT
  UNPIVOT (AnnualSales
    FOR SalesYear
    IN ([2011], [2012], [2013], [2014]))
    AS SalesPersonYear;
```

Figure 21 shows partial results and explains why I said this query "sort of" reverses the original. The query in **Listing 19** starts with the raw sales data, one record per sales order. But the original query aggregates those records. UNPIVOT has no way to disaggregate that aggregated data.

SalesPersonID	SalesYear	AnnualSales
284	2012	441639.5961
284	2013	1269908.9235
284	2014	600997.1704
278	2011	500091.8202
278	2012	1283569.6294
278	2013	1389836.8101
278	2014	435948.9551
281	2011	967597.2899
281	2012	2294210.5506
281	2013	2387256.0616
281	2014	777941.6519
275	2011	875823.8318
275	2012	3375456.8947
275	2013	3985374.8995
275	2014	1057247.3786

Figure 21. The unpivoted annual sales have one record per salesperson per year.

Using UNPIVOT to normalize data

UNPIVOT is useful for normalizing data. It's quite common to find databases that are not normalized, particularly tables that use multiple columns for essentially the same data. UNPIVOT lets you gather that data into a single column.

The AdventureWorks database doesn't have any such examples. So, to demonstrate this use of UNPIVOT, we'll have to create our own example data. (These examples are inspired by and adapted from Aaron Bertrand's article on UNPIVOT at http://tinyurl.com/grlkfyf.)

One of the most common examples of multiple columns rather than normalized data is the use of separate fields for different phone numbers, that is, having fields named Phone1, Phone2, etc., or Home, Mobile and Work. **Listing 29** creates and populates a simplified table that uses the latter approach. (To make it easy to see that the normalization code works, the phone numbers here use a pattern. The exchange—the 4th through 6th digits—for all home numbers is 555; for mobile numbers, it's 666; and for work numbers, it's 777. In addition, each number for an individual has the same last four digits. None of this makes a difference in how the code works; it simply makes checking the results easier.) **Figure 22** shows the data in the #Person table.

Listing 29. This code creates and populates a temporary table that uses separate named fields for home, mobile and work phone numbers.

```
CREATE TABLE #Person
(First varchar(15), Last varchar(20),
 Home varchar(10), Mobile varchar(10),
 Work varchar(10))
INSERT INTO #Person VALUES
  ('Jane', 'Smith', '5555551234', '5556661234', '5557771234'),
  ('Andrew', 'Jones', '5555557890', '5556667890', '5557777890'),
  ('Deborah', 'Cohen', NULL, '5556667474', '5557777474');
```

First	Last	Home	Mobile	Work
Jane	Smith	5555551234	5556661234	5557771234
Andrew	Jones	5555557890	5556667890	5557777890
Deborah	Cohen	NULL	5556667474	5557777474

Figure 22. This approach to storing phones is not normalized and means the data structure must change to accommodate each new type of phone.

With the proliferation of phone numbers for an individual, it makes much more sense to use a separate table to contain all phone numbers with one record per phone per person. The query in **Listing 30** extracts the data from the #Person table in the desired format. The result, shown in **Figure 23**, includes a column to indicate the type of phone; it's populated based on the column names in the original. Note also that there's no home phone record for Deborah Cohen, since the Home field was NULL. The downloads for this session include NormalizePhones.SQL, which creates the temporary #Person table and has the query to normalize the data.

Listing 30. This query normalizes the phone data, using the column names to indicate the type of phone.

```
SELECT First, Last, PhoneType, Phone
  FROM #Person
  UNPIVOT (Phone
    FOR PhoneType
    IN (Home, Mobile, Work)) AS Phones;
```

First	Last	PhoneType	Phone
Jane	Smith	Home	5555551234
Jane	Smith	Mobile	5556661234
Jane	Smith	Work	5557771234
Andrew	Jones	Home	5555557890
Andrew	Jones	Mobile	5556667890
Andrew	Jones	Work	5557777890
Deborah	Cohen	Mobile	5556667474
Deborah	Cohen	Work	5557777474

Figure 23. The normalized phone data is much more flexible.

The PhoneType column in this result is optional. If, for some reason, you don't want to include it, you can just leave it out of the field list, as in **Listing 31**. In this example, it's hard to see why you'd want to do that, but if the original columns were simply Phone1, Phone2 and Phone3, it would make sense.

Listing 31. This version of the query normalizes the phone data without including the phone type information from the column name.

```
SELECT First, Last, Phone
  FROM #Person
  UNPIVOT (Phone
    FOR PhoneType
    IN (Home, Mobile, Work)) AS Phones;
```

First	Last	Phone
Jane	Smith	5555551234
Jane	Smith	5556661234
Jane	Smith	5557771234
Andrew	Jones	5555557890
Andrew	Jones	5556667890
Andrew	Jones	5557777890
Deborah	Cohen	5556667474
Deborah	Cohen	5557777474

Figure 24. You can normalize without including the name of the column where the data originated.

Normalizing multiple columns

What if rather than a single set of columns that need to be normalized, you have multiple related sets? Again, telephone numbers provide a simple example. Suppose that instead of having one column per phone number, named with the phone type, you have a pair of columns for each phone number, one indicating the type and the second containing the

phone number. If the column names follow a pattern, UNPIVOT lets you normalize without losing any of the data.

Listing 32 creates and populates a different version of the #Person table. In this version, there are three pairs of columns, named Phone*n* and Phone*n*Type. Each holds one phone number, along with its type, and a person can have up to three. The data here is the same as in the previous version of the table, except that I've changed the order of the phone numbers in one record. As in the previous example, Deborah Cohen has no home number. **Figure 25** shows the table's contents.

Listing 32. This version of the #Person table uses two columns for each phone number, to specify the number and the type.

```
CREATE TABLE #Person
  (First varchar(15), Last varchar(20),
   Phone1 varchar(10), Phone1Type varchar(6),
   Phone2 varchar(10), Phone2Type varchar(6),
   Phone3 varchar(10), Phone3Type varchar(6));
INSERT INTO #Person VALUES
  ('Jane', 'Smith',
   '5555551234', 'Home',
   '5556661234', 'Mobile',
   '5557771234', 'Work'),
  ('Andrew', 'Jones',
   '5556667890', 'Mobile',
   '5557777890', 'Work',
   '5555557890', 'Home');
INSERT INTO #Person
  (First, Last, Phone1, Phone1Type,
   Phone2, Phone2Type) Values
  ('Deborah', 'Cohen',
   '5556667474', 'Mobile',
   '5557777474', 'Work');
```

First	Last	Phone1	Phone1Type	Phone2	Phone2Type	Phone3	Phone3Type
Jane	Smith	5555551234	Home	5556661234	Mobile	5557771234	Work
Andrew	Jones	5556667890	Mobile	5557777890	Work	5555557890	Home
Deborah	Cohen	5556667474	Mobile	5557777474	Work	NULL	NULL

Figure 25. In this version of the #Person table, each phone number has separate columns for number and type.

Surprisingly, given that doing multiple pivots requires separate queries, you can do multiple unpivots in a single query. That's what's required to normalize this data, as shown in **Listing 33** (included as NormalizePhoneAndType.SQL in the downloads for this session); the result is shown in **Figure 26**. This example also shows how you can manipulate the data in the new collective columns.

Listing 33. You can unpivot multiple related fields in a single query.

```
WITH AllPhones AS
```

```
(SELECT First, Last, Phone, PhoneType,
        RIGHT(Phones, 1) AS nPhone,
        SUBSTRING(PhoneTypes, 6, 1) AS nPhoneType
  FROM #Person
    UNPIVOT
      (Phone FOR Phones IN (Phone1, Phone2, Phone3)) AS PhoneList
    UNPIVOT
      (PhoneType FOR PhoneTypes
        IN (Phone1Type, Phone2Type, Phone3Type)) AS PhoneTypeList)

SELECT First, Last, Phone, PhoneType
  FROM AllPhones
  WHERE nPhone = nPhoneType;
```

First	Last	Phone	PhoneType
Jane	Smith	5555551234	Home
Jane	Smith	5556661234	Mobile
Jane	Smith	5557771234	Work
Andrew	Jones	5556667890	Mobile
Andrew	Jones	5557777890	Work
Andrew	Jones	5555557890	Home
Deborah	Cohen	5556667474	Mobile
Deborah	Cohen	5557777474	Work

Figure 26. Using a pair of UNPIVOTs and a little more work, we can normalize a table involving multiple related fields.

The actual unpivots are done in a CTE. First, we unpivot the phone number fields, specifying Phone as the field to hold the data, Phones as the collective name for the existing fields and listing them as Phone1, Phone2, and Phone3. The second unpivot does the same thing for the phone types, indicating that the data goes into PhoneType, the collective name for the existing fields is PhoneTypes and specifying the list of those fields as Phone1Type, Phone2Type and Phone3Type.

However, the pair of UNPIVOT clauses operate like a cross-join (also known as a Cartesian join); each item unpivoted from the first is joined to each item unpivoted from the second. We need a way to match up the corresponding numbers and types. The two extra fields in the CTE give us what we need to do that. They're also the reason we can only do this if there's a pattern to the field names. The first pulls out the digit from the phone number field name, while the second does the same for the phone type field. Each of those expressions uses the collective name for the list of fields as the way to find the name of the original field for that data item.

Figure 27 shows partial results from running just the query in the CTE. For Jane Smith, with three phone numbers, there are 9 records, one for each match of each phone number with each phone type. But the only valid records are those where nPhone and nPhoneType match.

First	Last	Phone	PhoneType	nPhone	nPhoneType
Jane	Smith	5555551234	Home	1	1
Jane	Smith	5555551234	Mobile	1	2
Jane	Smith	5555551234	Work	1	3
Jane	Smith	5556661234	Home	2	1
Jane	Smith	5556661234	Mobile	2	2
Jane	Smith	5556661234	Work	2	3
Jane	Smith	5557771234	Home	3	1
Jane	Smith	5557771234	Mobile	3	2
Jane	Smith	5557771234	Work	3	3
Andrew	Jones	5556667890	Mobile	1	1
Andrew	Jones	5556667890	Work	1	2
Andrew	Jones	5556667890	Home	1	3
Andrew	Jones	5557777890	Mobile	2	1
Andrew	Jones	5557777890	Work	2	2
Andrew	Jones	5557777890	Home	2	3

Figure 27. The two UNPIVOTs in the CTE in **Listing 33** do a cross-join of the unpivoted data. The additional fields nPhone and nPhoneType help us see where each item originated, so that the main query can filter out the mismatches.

With that data available, the main query in **Listing 33** keeps only those records for corresponding phone numbers and phone types.

Unpivoting unknown columns

It seems less likely to me that you might need to unpivot without knowing the column names, but I guess there can situations where data comes in from multiple sources with different numbers of unnormalized columns. Fortunately, you can build the list of columns for the unpivot and use dynamic SQL here, too. As in the matched columns example, doing so depends on the relevant column names following a pattern.

The key to doing this is to use the system columns table (that is, a system table named Columns that contains information about the columns in the database) to extract the relevant list. The columns of interest in that table are Name, which contains the field name, and Object_ID, which identifies the table to which the field belongs. As with the dynamic pivot, we need QUOTENAME() to wrap the field names to ensure they're valid. **Listing 34** shows the code to create and run the query from **Listing 33**; it's included in the downloads for this session as DynamicUnpivot.SQL. (This example is inspired by and adapted from Aaron Bertrand's article at http://tinyurl.com/z464ll9.)

Listing 34. You can build the list of fields and use dynamic SQL to unpivot when you're not sure how many fields you have.

```
-- Use variables here. In production,
-- these might be parameters
DECLARE @table AS NVARCHAR(max) = N'tempdb..#Person';
DECLARE @phonenames AS NVARCHAR(max) = N'Phone[0-9]';
DECLARE @typenames AS NVARCHAR(max) =  N'Phone[0-9]Type';

DECLARE @phonecols AS NVARCHAR(max);
```

```
DECLARE @typecols AS NVARCHAR(max);
DECLARE @query AS NVARCHAR(max);

SELECT @phonecols = ISNULL(@phonecols + ', ', '') + QUOTENAME(Name)
  FROM tempdb.sys.Columns
  WHERE Object_ID = OBJECT_ID(@table)
    AND Name LIKE @phonenames;

SELECT @typecols = ISNULL(@typecols + ', ', '') + QUOTENAME(Name)
  FROM tempdb.sys.columns
  WHERE object_id = OBJECT_ID(@table)
    AND name LIKE @typenames;

SET @query =
'WITH AllPhones AS
  (SELECT First, Last, Phone, PhoneType,
          RIGHT(Phones, 1) AS nPhone,
          SUBSTRING(PhoneTypes, 6, 1) AS nPhoneType
  FROM #Person
    UNPIVOT
      (Phone FOR Phones
       IN (' + @phonecols +')) AS PhoneList
    UNPIVOT
      (PhoneType FOR PhoneTypes
       IN (' + @typecols +')) AS PhoneTypeList)

SELECT First, Last, Phone, PhoneType
  FROM AllPhones
    WHERE nPhone = nPhoneType';

-- Run the query
EXEC sp_executesql @query;
```

First, we declare several variables that hold information about the query we need to build. As the comment says, in real code, these might be parameters to a stored procedure. The first, @table, identifies the table containing the data we want to normalize. Because we're working with a temporary table here, it needs the "tempdb.." prefix. The next two, @phonenames and @typenames, specify the pattern for the two sets of columns we're interested in.

Next, we declare variables to hold the two lists of columns and the query to be executed.

Then, two consecutive, quite similar queries populate the @phonecols and @typecols variables by extracting the list from the system Columns table. (Here again, we specify tempdb because we're interested in the columns of a temporary table. For a permanent table, you can omit that.) The strategy for building each list is the same as in the dynamic pivot example; start with a null string and build it up. The OBJECT_ID() function looks up the unique identifier for the table of interest, so that we consider only fields from that table.

Next, we populate the @query variable with the query of interest, plugging in the field names from the two variables @phonecols and @typecols.

Finally, we call sp_executesql to run the query we've built. Not surprisingly, in this case, we get the same results as for the static query, shown in **Figure 26**.

Reporting on crosstab and pivot data

Reporting on crosstab/pivot data offers some challenges, because we may not know how many columns are in the data or the names of those columns. But as noted earlier in this paper, the main reason for creating crosstabs and pivots is reporting.

The remainder of this paper is devoted to various ways to report on crosstabs and pivots. I'll cover traditional VFP reports, as well as exporting the data to Excel, creating Excel pivot tables, and charting in both Excel and VFP. For VFP reports, I'll show a technique I first learned nearly 30 years ago in a FoxTalk article by Nancy Jacobsen. Improvements in the Report Designer since then make the technique easier to use today.

The Problem with Reports

To create a report in VFP's Report Designer, you drop controls onto the report and position them as desired. (There's lots more you can do, but that's the core activity.) To include fields, you use the Field control, which lets you specify an expression using the dialog shown in **Figure 28**. For most reports, the expression for most Field controls is simply a field of a table or cursor. Sometimes, you add formatting with TRANSFORM() or combine a table field with some fixed text or combine multiple fields into a single report field.
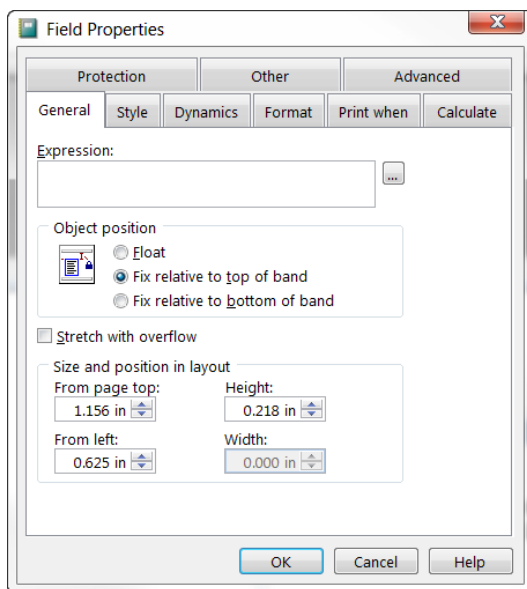
Figure 28. The Field Properties dialog lets you specify what a Field control in a report shows. Most often, it's just a field of a table or cursor.

The key point is that the expression includes the name of the field. When reporting on a crosstab, you may not know the name of each field in the cursor at the time you design the report.

Even trickier, because the list of fields in the crosstab result depends on the values in the original data, you may not know how many fields are in the crosstab result. In addition, it's possible to have more fields than fit on a single page.

For example, the query in **Listing 9** (earlier in this paper; **Figure 7** shows partial results) computes the total sales, average sale and number of sales for each employee for each year. The number of columns in the result depends on the number of years for which there is data. Columns have names like N_1997 (for total sales in 1997).

FIELD() and EVAL() to the rescue

The solution to not knowing the field names is to not put them into the report. Instead, use the FIELD() function. As the syntax shown in **Listing 35** indicates, FIELD() accepts two parameters. The first is the field number; it's required. The second parameter is optional and specifies the table you're interested in.

Listing 35. The FIELD() functions lets you refer to fields without knowing their names.

```
cFieldName = FIELD( nFieldNumber [, cAlias | nWorkarea ])
```

So rather than referring to N_1997 in the report, we can refer to FIELD(6). Using FIELD() solves the problem of not knowing the actual field names.

To determine the contents of a field by its position, wrap the EVAL() function around the FIELD() function. EVAL() is short for EVALUATE() and that's what it does. The function evaluates the expression you pass to it and returns the result. (I wrote at length about EVAL() and related language in this article.)

So to get the value of N_1997 for the current record, use EVAL(FIELD(6)).

How many fields?

The other problems are that we don't know how many columns we need in a report and we don't know whether they'll fit on a single page. We solve these two together by creating a report with as many fields as fit and then running it as many times as necessary to show all fields. Doing so requires a combination of a report and code to control it.

We'll return to the example in **Listing 9** later, but first let's consider a simpler example. The code in **Listing 36** computes sales for each employee for a given year (1997) by month. (It's the same as the code in **Listing 13**, except for the year in the filter.) There's one column in the result for each month of the year. In this case, we know how many columns there are and their names, but they still don't fit onto a single page. **Figure 29** shows partial results; in keeping with the theme of this section, they're cut off on the right.

Listing 36. This crosstab computes sales by employee by month for a single year.

```
SELECT EmployeeID, OrderDate, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
  FROM Orders ;
    JOIN OrderDetails ;
```

```
      ON Orders.OrderID = OrderDetails.OrderID ;
   GROUP BY 1, 2 ;
   INTO CURSOR csrMonthlyTotals

LOCAL oXTab AS FastXTab OF "fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab.prg")
WITH oXTab AS FastXTab OF fastxtab.prg"
   .cRowField = 'EmployeeID'
   .cColField = 'MONTH(OrderDate)'
   .cDataField = 'OrderTotal'
   .cOutFile = "csrXtab"
   .cCondition = 'year(OrderDate) = 1997'
   .lCursorOnly = .T.
   .lCloseTable = .T.
   .RunXtab()
ENDWITH

* Add employee name
SELECT PADR(ALLTRIM(FirstName) + (' ' + LastName), 30) AS cName, ;
       csrXtab.* ;
   FROM csrXtab ;
     JOIN Employees ;
       ON csrXtab.EmployeeID = Employees.EmployeeID ;
   ORDER BY LastName, FirstName ;
   INTO CURSOR csrReport
```

Cname	Employeeid	N_1	N_2	N_3	N_4	N_5	N_6
Steven Buchanan	5	0.0000	0.0000	2634.4000	0.0000	5127.5000	3124.9000
Laura Callahan	8	6701.1000	7764.4000	4806.1000	800.5000	4792.6000	2616.0500
Nancy Davolio	1	7331.6000	2504.6000	5493.9000	240.0000	9168.2500	6112.6500
Anne Dodsworth	9	1208.5000	0.0000	1770.8000	611.0000	139.8000	3761.5000
Andrew Fuller	2	3150.2000	1584.0000	2905.1000	14019.3000	4589.6000	7058.6000
Robert King	7	13703.4000	3891.0000	3867.2000	5707.3500	6041.2500	2082.0000
Janet Leverling	3	7477.9000	10581.3000	11599.4000	10297.3500	18639.8000	5871.6000
Margaret Peacock	4	25620.1000	13530.3000	5644.8000	14333.1500	7573.2000	4232.4000
Michael Suyama	6	1500.0000	1351.6000	1258.2000	9690.7400	751.7000	4228.3000

Figure 29. This cursor contains sales for each employee for each month of a single year.

We can refer to the data fields in this cursor as FIELD(3), FIELD(4), and so on, but how can we get them all into a report?

First, we need to figure out how many data columns we can fit on a page. I decided to keep the paper in portrait mode (you can fit more columns in landscape, of course) and a little experimentation showed me that along with the employee's name, four data columns would fit. **Figure 30** shows the report; we'll dig into its contents in the next section.
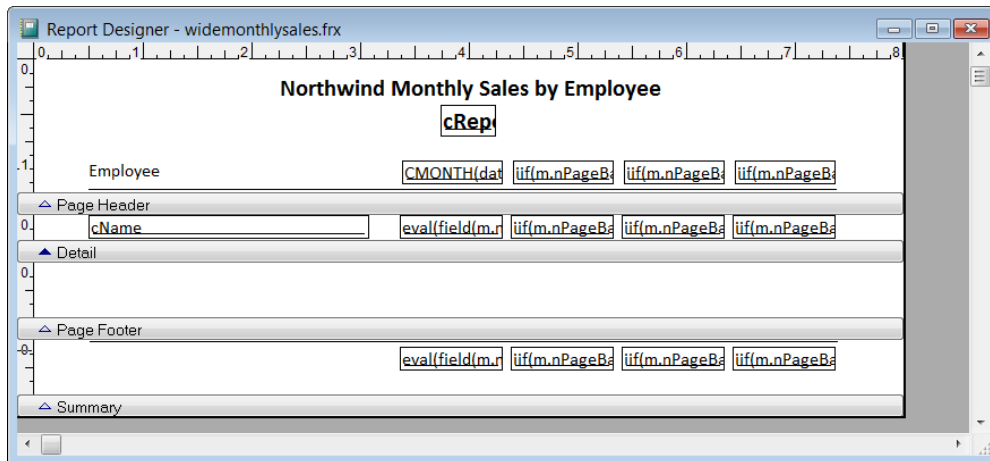
Figure 30. We can fit the employee's name and four months' data on one page in portrait mode.

Once we know how many columns fit, we can figure out how many pages we'll need. (I'm using the word "page" here to indicate pages across. It's entirely possible that there could be enough records in the cursor to print multiple pages of length, as well.) In the code to drive the report, we can set a variable, nFieldsPerPage, to that value (4, in this case). We also need to know where in the cursor the data fields start. In this case, they start in column 3, because column 1 is the employee name and column 2 is the employee number. Store that number in a variable as well; I call it nFirstDataCol. With that information, we can divide the total number of data fields by fields per page, to determine the number of pages. **Listing 37** shows this part of the code.

Listing 37. To make the report work, we need to ask some questions about the data: how many fields there are total, where the data fields start, and how many fields fit on a page. Using those, we can determine the number of pages we need.

```
nFieldCount = FCOUNT("csrReport")
nFirstDataCol = 3
nFieldsPerPage = 4
nPages = CEILING((m.nFieldCount - m.nFirstDataCol + 1)/m.nFieldsPerPage)
```

Obviously, though, we only want to create a single report, not a separate report for each page. So, we need to specify the fields and headings in the report in a way that works for each page. Using FIELD() is clearly part of the answer, but we also need to calculate the parameter we pass to FIELD(), so that for each page of the report, we get the right subset of fields. **Listing 38** shows the code that runs the report. The key is the calculation of nPageBase, a variable that determines which column from the cursor is the first data column on each page. For our example, on the first page, nPageBase = 3, the first data column. On the second page, nPageBase = 7, and so on. We can use nPageBase in the report to figure out which fields appear on this page.

Listing 38. We run the report in a loop, once for each page.

```
FOR m.nPage = 1 TO m.nPages
  nPageBase = (m.nPage-1) * m.nFieldsPerPage + m.nFirstDataCol
```

```
  REPORT FORM WideMonthlySales PREVIEW
ENDFOR
```

Making the report generic

The final piece is to design the report to use the pieces we've collected. The column headings, the fields themselves and any totals need to be generic.

We can assume that we only print a page if we have at least one data column to show there, but any other data column may or may not appear on the last page. In the sales by month example we're working on, there are 12 data columns (one per month) and 4 data columns per page, so in fact, this isn't an issue. But we'll look at how to handle it anyway.

In the first data column, to show the value, we use EVAL(FIELD(m.nPageBase)), as in **Figure 31**.
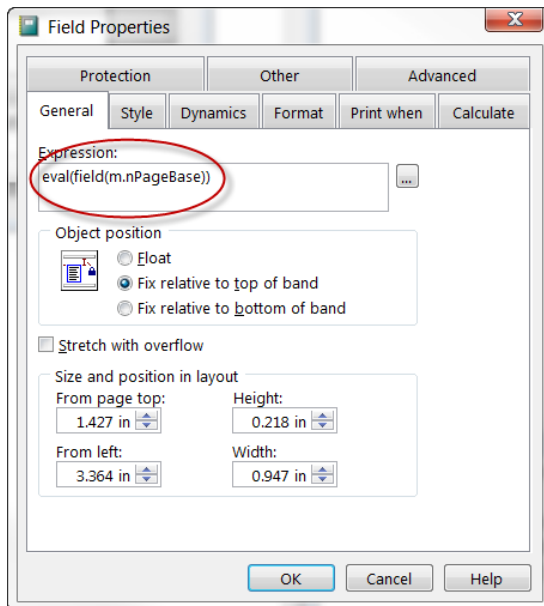
Figure 31. The expression for the value of the first data field on the page is fairly simple because we know that it will always be present.

For the other data columns, we need to do two things. First, we need to count upward from nPageBase, so the second data column will show data from FIELD(m.nPageBase + 1), the third from FIELD(m.nPageBase + 2) and so on.

Second, we need to consider the possibility that we've reached the end of the data columns. To handle that case, we wrap the expression in IIF(), as in **Listing 39**, which shows the expression for the fourth data column. **Figure 31** shows the expression in the Field Properties dialog.

Listing 39. After the first data column on the page, we must take into account the possibility that there is no such column.

```
iif(m.nPageBase + 3 <= m.nFieldCount, eval(field(m.nPageBase + 3)), 0)
```
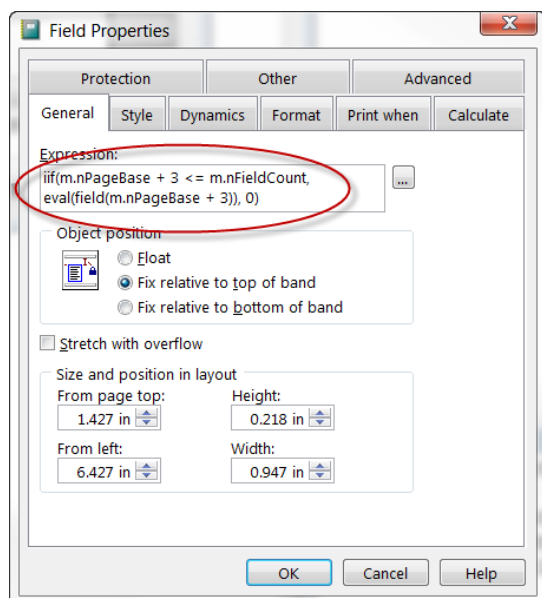
Figure 32. For data columns after the first, we use IIF() so that we don't try to evaluate a field that doesn't exist.

As the expression indicates, though, we're still returning 0 when there's no field. We want to make sure nothing appears in a column for which there's no corresponding field. To do that, we use the Report Designer's Print When capability, specifying the same condition we used in IIF(). **Figure 33** shows the dialog.
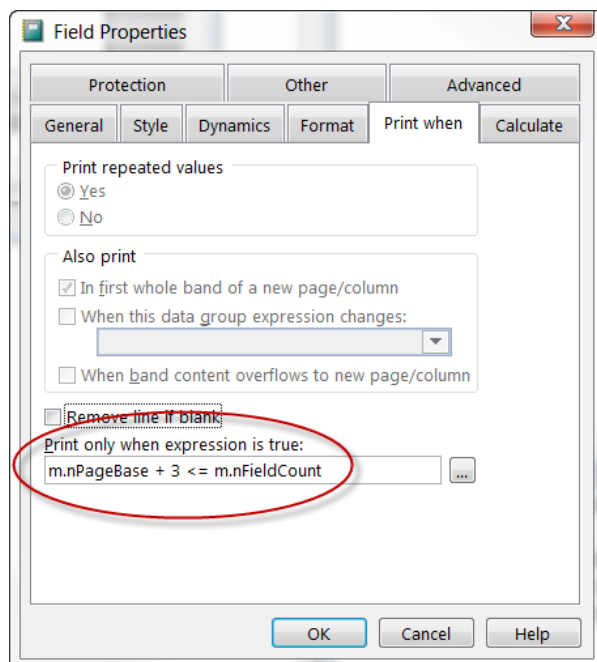
Figure 33. To prevent anything from showing up for a non-existent column, we use the Print When tab of the Field Properties dialog.

For column totals, you apply the same techniques, using EVAL(FIELD()), adding IIF() for data columns after the first, and setting the total field to print only when the column exists.

Column headings use similar techniques, but need a little creativity because the field names in a crosstab may not be terribly informative. In this example, they look like N_1, N_2, etc. We know that the number at the end is the month number we're interested in, so we can use that to build an informative heading.
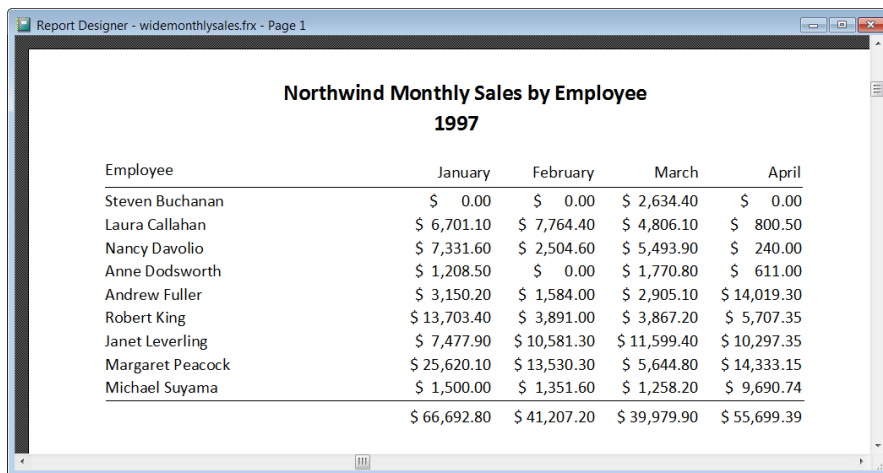
As before, we know we'll have data in the first column on the page. For the first data column, we use the expression in **Listing 40**. Explaining from the inside out, we grab the field name and extract everything after the underscore. We convert that from character to numeric, and then use it as the month parameter to DATE(). Finally, we call CMONTH() to get the name of the month.

Listing 40. To get a meaningful column heading, we extract the numeric part of the field name, build a date using that number for the month and then get the month name.

```
CMONTH(date(m.nReportYear, eval(strextract(field(m.nPageBase),"_")), 1))
```

For subsequent columns, we need to wrap the heading in the same IIF() condition we used for the data values and totals, and specify Print When to make the column appear only when it actually exists.
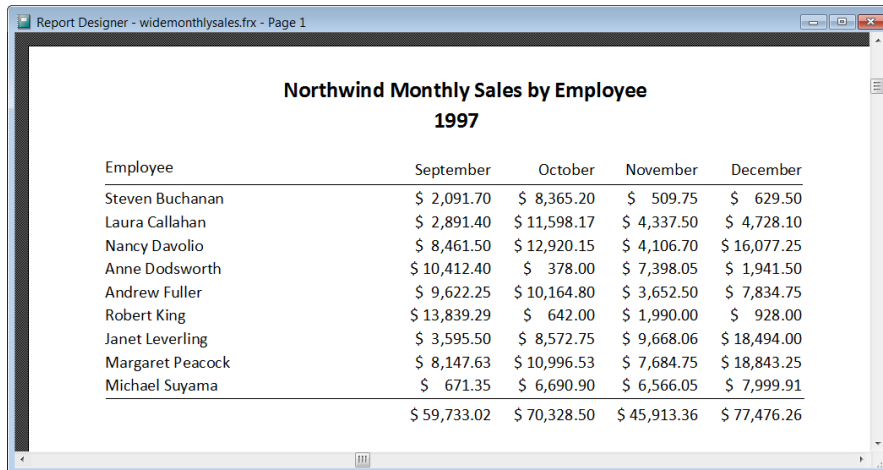
For this example, that's everything. **Figure 34** shows the first page of the report; **Figure 35** shows the last (third) page. The downloads for this session include ReportSalesPersonMonthly.prg, which puts the whole process together, including running the report, and WideSalesAvgCount.frx, the report itself.

Report Designer - widemonthlysales.frx - Page 1

Northwind Monthly Sales by Employee
1997

Employee	January	February	March	April
Steven Buchanan	$ 0.00	$ 0.00	$ 2,634.40	$ 0.00
Laura Callahan	$ 6,701.10	$ 7,764.40	$ 4,806.10	$ 800.50
Nancy Davolio	$ 7,331.60	$ 2,504.60	$ 5,493.90	$ 240.00
Anne Dodsworth	$ 1,208.50	$ 0.00	$ 1,770.80	$ 611.00
Andrew Fuller	$ 3,150.20	$ 1,584.00	$ 2,905.10	$ 14,019.30
Robert King	$ 13,703.40	$ 3,891.00	$ 3,867.20	$ 5,707.35
Janet Leverling	$ 7,477.90	$ 10,581.30	$ 11,599.40	$ 10,297.35
Margaret Peacock	$ 25,620.10	$ 13,530.30	$ 5,644.80	$ 14,333.15
Michael Suyama	$ 1,500.00	$ 1,351.60	$ 1,258.20	$ 9,690.74
	$ 66,692.80	$ 41,207.20	$ 39,979.90	$ 55,699.39

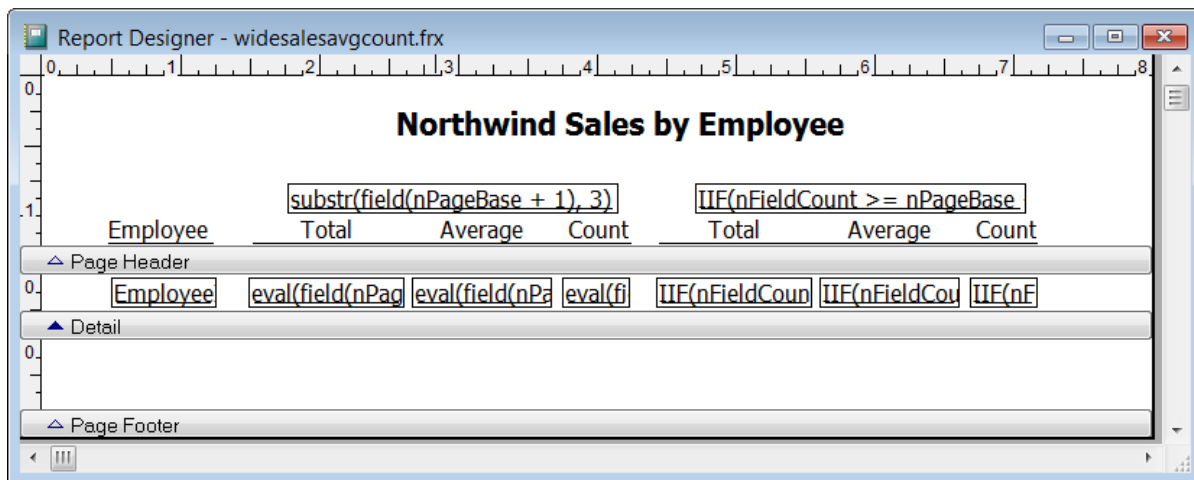**Figure 34**. When we run the loop, we see each (width) page of the report one at a time.

Figure 35. In this example, the final page has the same number of columns as the others, but that's not required.

Handling more complicated data

Sometimes, as in **Listing 9**, the data created by the crosstab is more complex. That code creates a cursor with three data columns for each year. In the report, we want all three of those columns on the same page. So, we need to make sure that the number of data columns per page is a multiple of 3. I chose to put 6 data columns, two years' worth on a page. **Figure 36** shows the report layout; it's included in the downloads for this session as WideSalesAvgCount.frx.

Figure 36. There's room for two years' data on one page in this report.

Ideally, we should group the column headings as well, and show the year above the three columns for that year. In this case, the field names are in the form N_1996, etc., so it's easy to parse the year portion out and build an expression, as in **Listing 41**.

Listing 41. We want one column heading across all three columns for a year. This expression parses the year from the name of the first of the three fields for that year.

```
substr(field(nPageBase + 1), 3) + ' sales'
```

As before, after the first data column on a page, we need to make sure there's actually data. In this case, because we know the data columns come in groups of 3, we don't have to check until we get to column 4, the beginning of the second year's data on the page. **Figure 37** shows the Field Properties dialog for the heading over the 4th, 5th and 6th data columns on the page.
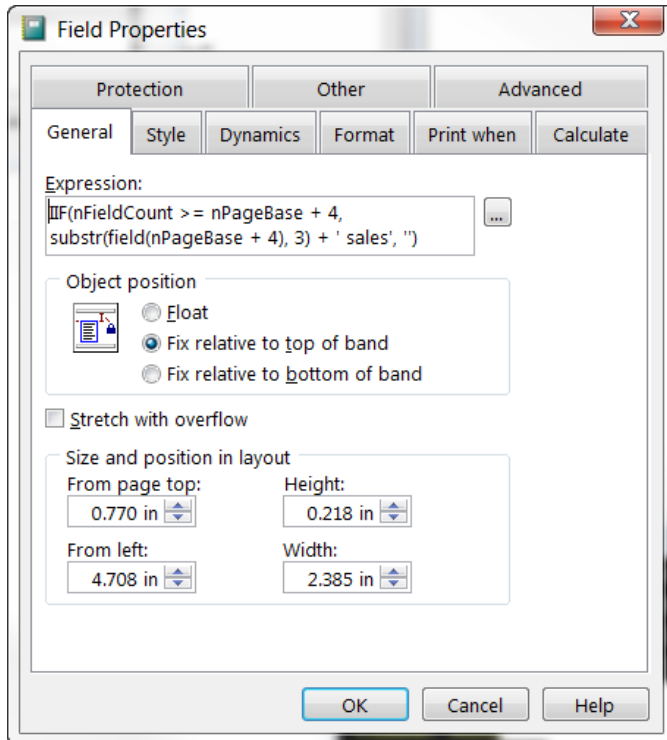

Figure 37. This heading for the second year's data on the page uses IIF() to make sure there is actual data to report.

This report also addresses an issue ignored in the previous example, handling the lines under headings (and, if you have them, between rows or before totals). You usually only want lines where there's data. Here, each year's data has a separate line under its headings. The line for the second group on the page has a Print When condition to ensure that it appears only when there's actually data. **Figure 38** shows the Print When tab of the Line Properties dialog with the expression.
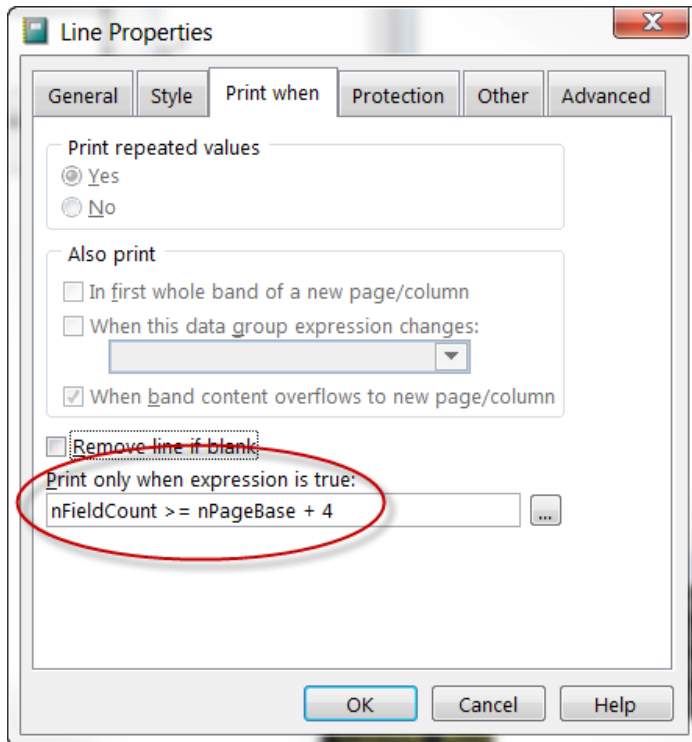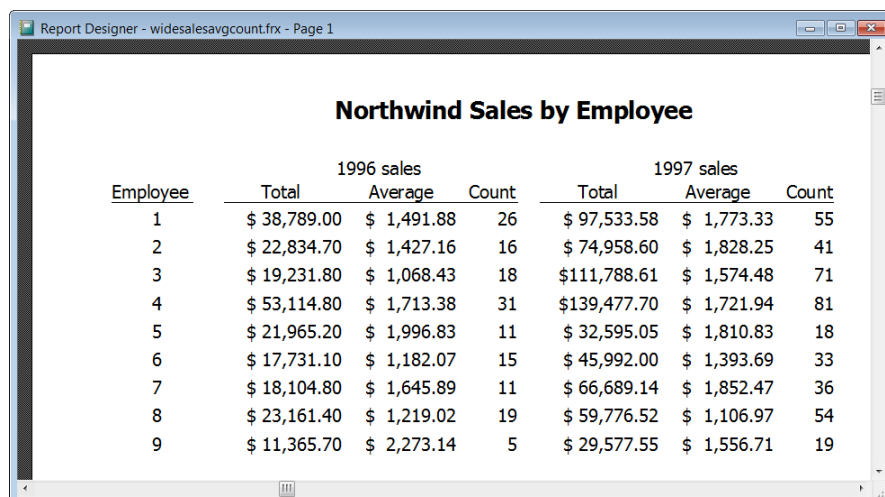
Figure 38. This Print When condition ensures that the underline for the second year's headings shows up only if there is a second year on the page.

The data fields themselves use the same EVAL(FIELD(n)) approach as in the previous example. The fields for the second year on the page are wrapped with IIF() and have a matching Print When condition.

The code to drive the report is pretty much the same as in the previous example, figuring out how many fields there are and how many pages, then looping to run the report repeatedly. The complete program to drive this report is included in the downloads for this session as ReportSalespersonAnnualSumAvgCnt.prg.
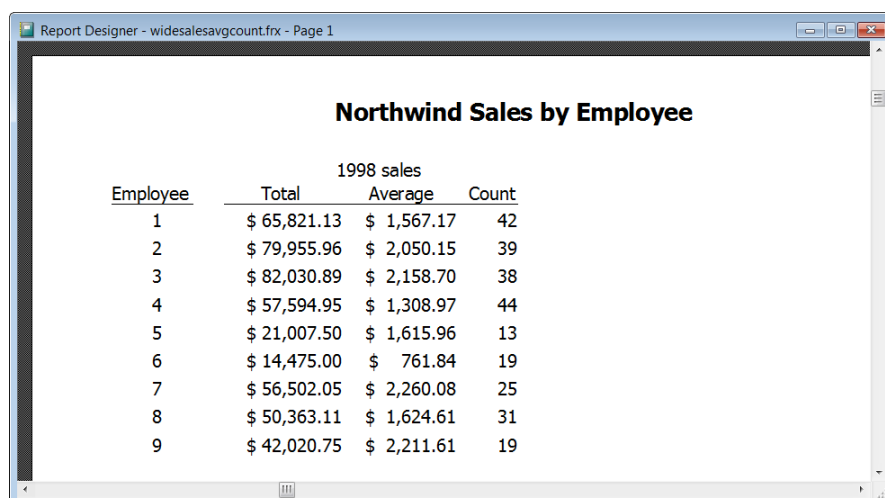
Figure 39 shows the first page of the report, while **Figure 40** shows the final (second) page. Note that the right-hand side in **Figure 40** is entirely blank, with no headings, no values, and no lines.

Figure 39. The first page of the report includes data for two years.

Figure 40. The last page of the report has data for only one year.

Alternative approaches to reports

Doug Hennig published an alternative approach to this problem in FoxTalk. It uses functions rather than a loop to control output and thus produces a single report output rather than one for each horizontal page. You'll find his article at https://doughennig.com/Papers/Pub/199704dhen.pdf.

One problem with both the Jacobsen and Hennig approaches is that, if you save the report as a PDF, the data isn't organized as you'd want it to be. One option, discussed in the next section, is to send the data to Excel and output it from there. But there is another option, provided by Cathy Pountney. In her paper "OutFox the VFP Report Writer," originally presented at Southwest Fox 2007, Cathy provides a report listener that allows you to change the output paper size at runtime. With this approach, you can create that wide, wide report and send it to PDF. With her permission, the materials for Cathy's session are included with the downloads for this session.

Sending crosstabs to Excel

Exporting crosstab data to Excel can be a great solution because Excel can handle lots of columns, and the various export techniques don't generally require you to know how many columns you have ahead of time. In addition, sending data to Excel gives users the chance to do additional crunching on it as needed.

We can export the crosstab data as is or we can send raw data and use Excel's pivot table capabilities to crosstab it there. We'll look at both options.

We'll work with the example in **Listing 42**, which collects sales by employee by month.

Listing 42. This code produces a cursor with one row for each employee for each month in which the employee had any sales.

```
SELECT EmployeeID, ;
       YEAR(OrderDate) AS Year, ;
       MONTH(OrderDate) as Month, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
    FROM Orders ;
      JOIN OrderDetails ;
        ON Orders.OrderID = ;
           OrderDetails.OrderID ;
    GROUP BY 1, 2, 3 ;
    INTO CURSOR csrMonthlyTotals

* Add employee name
SELECT PADR(ALLTRIM(FirstName) + (' ' + LastName), 30) AS cName, ;
       csrMonthlyTotals.* ;
    FROM csrMonthlyTotals ;
      JOIN Employees ;
        ON csrMonthlyTotals.EmployeeID = Employees.EmployeeID ;
    ORDER BY LastName, FirstName ;
    INTO CURSOR csrReport
```

Sending cross-tabbed data to Excel

The simpler solution is to take the crosstab or pivot results and just send them to Excel. Depending how much control you want, there are a variety of ways to do this.

The code in **Listing 43** gives us the crosstabbed results, with one row per employee and one column per month. The combined code from these two listings is included in the downloads for this session as CrossTabSalesPersonMonthly.prg.

Listing 43. This code creates a crosstab from the results of the query in Listing 42.

```
LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"

oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
   .cOutFile = "csrXtab"
   .cRowFIELD = "cName"
   .cColField = [PADL(OrdYear,4) + "-" + PADL(OrdMonth,2)]
```

```
    .cDATAFIELD = "OrderTotal"
    .lCursorOnly = .T.
    .lCLOSETABLE = .T.
    .RunXtab()
ENDWITH
```

The most basic way to send this data to Excel is the COPY TO command. You have two output options, Excel data or CSV (comma-separated value). The only difference in the COPY TO command is the keyword you place after TYPE. To create an Excel file, use XL5; for a CSV, use CSV.

COPY TO supports two different Excel types, XLS and XL5. Both are somewhat outdated. XLS is the file format from Excel 2.0. Do not use it; it has two important limitations. First, dates aren't handled property. Second, it's limited to 16,384 rows. (Obviously, this is not an issue for our example, but might be in other cases.)

TYPE XL5 handles dates properly, but is limited to 32,767 rows. That's one reason you may prefer TYPE CSV, which doesn't have that limit and is treated by Excel as if it were a native type. (As this discussion implies, COPY TO has no way to create the more recent XLSX file; see the next section for options for doing that.)

Listing 44 shows the code to export to TYPE XL5; the code is included in the downloads for this session as ExportToXL5SalesPersonMonthly.prg. **Figure 41** shows partial results before doing any formatting in Excel.

Listing 44.This code exports the crosstab results to Excel directly.

```
LOCAL cXLSFile
cXLSFile =  FORCEPATH(FORCEEXT( "SalesPersonMonthly", "XLS"), SYS(2023))

COPY TO (m.cXLSFile) TYPE XL5
```

	A	B	C	D	E	F	G	H	I	J	
1	cname	c_1996_7	c_1996_8	c_1996_9	c_1996_1(	c_1996_1	c_1996_1:	c_1997_1	c_1997_2	c_1997_3	c_
2	Andrew Fu	1176	1814	2950.8	5725.7	4759	6409.2	3150.2	1584	2905.1	
3	Anne Dod:	4955.3	0	0	6244.4	0	166	1208.5	0	1770.8	
4	Janet Leve	2998.2	3557.2	1762	4317.6	3838	2758.8	7477.9	10581.3	11599.4	1
5	Laura Call:	1726	8485.8	5248	385.2	704.8	6611.6	6701.1	7764.4	4806.1	
6	Margaret F	12988.9	3670.5	3575.1	14422.1	12017.4	6440.8	25620.1	13530.3	5644.8	1
7	Michael Su	2587.9	2595.4	4465.6	0	2299.8	5782.4	1500	1351.6	1258.2	
8	Nancy Dav	2018.6	6007.1	6883.7	4061.4	10261.2	9557	7331.6	2504.6	5493.9	
9	Robert Kin	0	479.4	1330.8	4577.2	11717.4	0	13703.4	3891	3867.2	
10	Steven Bu	1741.2	0	1420	1470	4106.4	13227.6	0	0	2634.4	

Figure 41. When you export with COPY TO, the data is simply dumped into an Excel file with no formatting.

The code to export to CSV is nearly identical; it's shown in **Listing 45** and included as ExportToCSVSalespersonMonthly.prg in the downloads for this session. When you open the file in Excel, before doing any formatting, it looks almost identical to **Figure 41**. (The only

difference I see is in the font used. On my Windows 10 computer, the XLS file uses 10-point Arial, while the CSV uses 11-point Calibri, my default for Excel.)

Listing 45. Exporting to a CSV file is another way to get data into Excel.

```
LOCAL cCSVFile
cCSVFile =  FORCEPATH(FORCEEXT( "SalesPersonMonthly", "CSV"), SYS(2023))

COPY TO (m.cCSVFile) TYPE CSV
```

If you need to format the spreadsheet before passing it along to users, you can use Automation to open it in Excel and do the formatting.

Creating XLSX files

Given that the XLSX format first appeared in Office 2007, your users may want that rather than the older XLS or CSV format. One easy way to create an XLSX is to generate an XLS or CSV file with COPY TO, and then use Automation to save it in a newer format. The code in **Listing 46** (included as ExportToXLSXSalespersonMonthly.prg in the downloads for this session) does that, with a few safeguards in case Excel or the workbook can't be opened.

Listing 46. This code exports data to Excel using COPY TO and then opens the XLS file and saves it in the XLSX format.

```
LOCAL cXLSFile, cXLSXFile
LOCAL oXL, oWorkbook, lSuccess

cXLSFile =  FORCEPATH(FORCEEXT("SalesPersonMonthly", "XLS"), SYS(2023))
cXLSXFile = FORCEEXT(m.cXLSFile, "XLSX")

COPY TO (m.cXLSFile) TYPE XL5

TRY
   oXL = CREATEOBJECT("Excel.Application")
   lSuccess = .T.
CATCH
   MESSAGEBOX("Unable to open Excel.")
   lSuccess = .F.
ENDTRY

IF m.lSuccess
   TRY
      oWorkbook = oXL.Workbooks.Open(m.cXLSFile)
      lSuccess = .T.
   CATCH
      MESSAGEBOX("Unable to open workbook in Excel for conversion to XLSX.")
      lSuccess = .F.
      oXL.Quit()
   ENDTRY
ENDIF

IF m.lSuccess
   TRY
```

```
    oWorkbook.SaveAs(m.cXLSXFile, 51) && 51 = xlOpenXMLWorkbook
  CATCH
    MESSAGEBOX("Unable to convert workbook to XLSX.")
  ENDTRY
  oXL.Quit()
ENDIF
```

As before, you still need to format the workbook afterward; the result looks the same as **Figure 41**.

In addition, this approach requires Excel 2007 or later to be installed on the machine where the code is running. Since that may be a problem, there are several open source projects that allow you to create XLSX files without Excel. I'll briefly cover two of them here.

XLSX Workbook

XLSX Workbook comes from Greg Green, who has created a number of utilities for VFP developers. This one is a class that lets you not only export data to Excel, but format it as well. Even better, it has extensive documentation. Like FastXTab, this tool has been added to VFPX since I first offered this session. You'll find the latest version at https://github.com/ggreen86/XLSX-Workbook-Class.

Exporting a table or cursor takes just a couple of lines of code, as in **Listing 47**. First, you instantiate the VFPxWorkbookXLSX class; be sure to either put the class library in your path, or to add the path to the NewObject() call. Then, call the SaveTableToWorkbookEx method, passing the table name or alias to export and the name of the file you want to create. The code is included as ExportXLSXWorkbookSalespersonMonthly.prg in the downloads for this session. Partial results are shown in **Figure 42**.

Listing 47. Exporting a table or cursor to Excel with XLSX Workbook is simple.

```
LOCAL cXLSFile, oToExcel, lReturn
cXLSXFile =  FORCEPATH(FORCEEXT("SalesPersonMonthly", "XLSX"), SYS(2023))

oToExcel = NEWOBJECT("VFPxWorkbookXLSX", "VFPxWorkbookXLSX.vcx")
lReturn = oToExcel.SaveTableToWorkbookEx("csrXTab", m.cXLSXFile)

IF NOT m.lReturn
   MESSAGEBOX("Unable to create workbook")
ENDIF
```

	A	B	C	D	E	F	G	H	I	J
1	CNAME	C_1996_7	C_1996_8	C_1996_9	C_1996_1(	C_1996_1:	C_1996_1:	C_1997_1	C_1997_2	C_1997_3 (
2	Andrew Fu	1176	1814	2950.8	5725.7	4759	6409.2	3150.2	1584	2905.1
3	Anne Dods	4955.3	0	0	6244.4	0	166	1208.5	0	1770.8
4	Janet Leve	2998.2	3557.2	1762	4317.6	3838	2758.8	7477.9	10581.3	11599.4
5	Laura Calla	1726	8485.8	5248	385.2	704.8	6611.6	6701.1	7764.4	4806.1
6	Margaret F	12988.9	3670.5	3575.1	14422.1	12017.4	6440.8	25620.1	13530.3	5644.8
7	Michael Su	2587.9	2595.4	4465.6	0	2299.8	5782.4	1500	1351.6	1258.2
8	Nancy Dav	2018.6	6007.1	6883.7	4061.4	10261.2	9557	7331.6	2504.6	5493.9
9	Robert Kin	0	479.4	1330.8	4577.2	11717.4	0	13703.4	3891	3867.2
10	Steven Buc	1741.2	0	1420	1470	4106.4	13227.6	0	0	2634.4

Figure 42. XSLX Workbook can export directly from a table or cursor to XSLX. By default, it does no formatting.

The class has many additional methods, including the ability to save a grid to a workbook, maintaining its formatting.

DBF2XLSX

This tool comes from Vilhelm-Ion Praisach (whose extensions to FastXtab are discussed earlier in this paper). The code is included in the downloads for this session, and is also linked in Praisach's blog at http://praisachion.blogspot.com/2017/04/export-dbf-to-excel-2007.html. It includes the basic export, as well as a class to provide UI for exporting. Praisach also provides VFP 6-compatible versions. (In addition, elsewhere in his blog, Praisach offers a tool for importing XLSX files and tools for exporting to other Office formats.)

Using DBF2XLSX is even easier than using XLSX Workbook. **Listing 48** shows the necessary code. A single line of code does the export; **Figure 43** shows partial results. The code is included as ExportDBF2XLSXSalespersonMonthly.prg in the downloads for this session.

Listing 48. With DBF2XLSX, one line of code exports a table or cursor to a modern Excel workbook.

```
LOCAL cXLSFile, oToExcel, lReturn
cXLSXFile =  FORCEPATH(FORCEEXT("SalesPersonMonthly", "XLSX"), SYS(2023))

DO CopyToXLSX WITH 'csrXTab', m.cXLSXFile, .T.
```

	A	B	C	D	E	F	G	H	I	J
1	CNAME	C_1996_7	C_1996_8	C_1996_9	C_1996_1(	C_1996_1:	C_1996_1:	C_1997_1	C_1997_2	C_1997_3
2	Andrew Fu	$1,176.00	$1,814.00	$2,950.80	$5,725.70	$4,759.00	$6,409.20	$3,150.20	$1,584.00	$2,905.10
3	Anne Dods	$4,955.30	$0.00	$0.00	$6,244.40	$0.00	$166.00	$1,208.50	$0.00	$1,770.80
4	Janet Leve	$2,998.20	$3,557.20	$1,762.00	$4,317.60	$3,838.00	$2,758.80	$7,477.90	########	########
5	Laura Calla	$1,726.00	$8,485.80	$5,248.00	$385.20	$704.80	$6,611.60	$6,701.10	$7,764.40	$4,806.10
6	Margaret F	########	$3,670.50	$3,575.10	########	########	$6,440.80	########	########	$5,644.80
7	Michael Su	$2,587.90	$2,595.40	$4,465.60	$0.00	$2,299.80	$5,782.40	$1,500.00	$1,351.60	$1,258.20
8	Nancy Dav	$2,018.60	$6,007.10	$6,883.70	$4,061.40	########	$9,557.00	$7,331.60	$2,504.60	$5,493.90
9	Robert Kin	$0.00	$479.40	$1,330.80	$4,577.20	########	$0.00	########	$3,891.00	$3,867.20
10	Steven Buc	$1,741.20	$0.00	$1,420.00	$1,470.00	$4,106.40	########	$0.00	$0.00	$2,634.40

Figure 43. The output from CopyToXLSX needs some formatting.

Only the first two parameters are required, providing the table or cursor to be copied and the name of the result file. The third parameter indicates whether the first row of the result should contain the column names. Additional parameters let you indicate which columns to include, as well as what to do with memo fields.

The downloads don't include documentation other than some test code, but several posts on Praisach's blog discuss how to use this routine and the others. Multiple posts indicate that the class ExportXLSX has the ability to export a grid, not just a table or cursor.

Other options

There are several other projects in the VFP world that have the capability of creating XLSX files.

FoxyXLS was created by Cesar Chalom, the creator of FoxCharts and FoxyPreviewer. Unlike the tools above, it doesn't provide a one-line way to convert, but it doesn't take too much code. Rick Schummer wrote about it in detail in the March, 2014 issue of FoxRockX.

Çetin Basoz's VFP2Excel is a simple procedure that copies VFP data to Excel using ADO. Unlike the other tools described here, it expects Excel to be present and running. One of its parameters is a reference to the range in Excel where the data should be placed.

Creating pivot tables

The previous solutions simply take the generated crosstab and dump it into a spreadsheet. But Excel has some very nice capabilities relating to crosstabs and pivots. You can give users the ability to slice and dice the data in different ways, including filtering based on the values provided and collapsing whole sections.

Figure 44 shows (part of) the result we'll be working toward in this section. The whole spreadsheet is included as PivotedSalesPersonMonthly.xlsx in the downloads for this session.

Salesperson	7	8	9	10	11	12	1996 Total	1	2	3	4	5	6	7
Andrew Fuller	$1,176	$1,814	$2,951	$5,726	$4,759	$6,409	$22,835	$3,150	$1,584	$2,905	$14,019	$4,590	$7,059	$10,320
Anne Dodsworth	$4,955			$6,244		$166	$11,366	$1,209		$1,771	$611	$140	$3,762	$28
Janet Leverling	$2,998	$3,557	$1,762	$4,318	$3,838	$2,759	$19,232	$7,478	$10,581	$11,599	$10,297	$18,640	$5,872	$1,109
Laura Callahan	$1,726	$8,486	$5,248	$385	$705	$6,612	$23,161	$6,701	$7,764	$4,806	$801	$4,793	$2,616	$3,984
Margaret Peacock	$12,989	$3,671	$3,575	$14,422	$12,017	$6,441	$53,115	$25,620	$13,530	$5,645	$14,333	$7,573	$4,232	$6,105 $
Michael Suyama	$2,588	$2,595	$4,466		$2,300	$5,782	$17,731	$1,500	$1,352	$1,258	$9,691	$752	$4,228	$1,301
Nancy Davolio	$2,019	$6,007	$6,884	$4,061	$10,261	$9,557	$38,789	$7,332	$2,505	$5,494	$240	$9,168	$6,113	$19,998
Robert King		$479	$1,331	$4,577	$11,717		$18,105	$13,703	$3,891	$3,867	$5,707	$6,041	$2,082	$6,145
Steven Buchanan	$1,741		$1,420	$1,470	$4,106	$13,228	$21,965			$2,634		$5,128	$3,125	$6,475
Grand Total	$30,192	$26,609	$27,636	$41,204	$49,704	$50,953	$226,299	$66,693	$41,207	$39,980	$55,699	$56,824	$39,088	$55,465 $

Figure 44. Excel's pivot tables present crosstab data in a way that lets users explore it.

About pivot tables

Excel has included pivot tables for a long time; the functionality was first introduced in Excel 5.0, back in 1994. The Pivot Table wizard, introduced in Excel 97, made it easy to create pivot tables.

As **Figure 44** shows, pivot tables allow you to use multiple criteria for pivoting the data. Here, the rows are based on salesperson, while the columns are based on both month and year. When you use multiple criteria, pivot tables let you collapse and expand; **Figure 45** shows the same worksheet with the 1996 data collapsed to a single column.

Salesperson	⊞ 1996	1	2	3	4	5	6	7	8
Andrew Fuller	$ 22,834.70	$ 3,150.20	$ 1,584.00	$ 2,905.10	$ 14,019.30	$ 4,589.60	$ 7,058.60	$ 10,320.00	$ 57.50
Anne Dodsworth	$ 11,365.70	$ 1,208.50		$ 1,770.80	$ 611.00	$ 139.80	$ 3,761.50	$ 28.00	$ 1,928.00
Janet Leverling	$ 19,231.80	$ 7,477.90	$ 10,581.30	$ 11,599.40	$ 10,297.35	$ 18,639.80	$ 5,871.60	$ 1,109.15	$ 5,881.80
Laura Callahan	$ 23,161.40	$ 6,701.10	$ 7,764.40	$ 4,806.10	$ 800.50	$ 4,792.60	$ 2,616.05	$ 3,984.10	$ 4,756.50
Margaret Peacock	$ 53,114.80	$ 25,620.10	$ 13,530.30	$ 5,644.80	$ 14,333.15	$ 7,573.20	$ 4,232.40	$ 6,104.80	$ 16,766.79
Michael Suyama	$ 17,731.10	$ 1,500.00	$ 1,351.60	$ 1,258.20	$ 9,690.74	$ 751.70	$ 4,228.30	$ 1,301.00	$ 3,982.25
Nancy Davolio	$ 38,789.00	$ 7,331.60	$ 2,504.60	$ 5,493.90	$ 240.00	$ 9,168.25	$ 6,112.65	$ 19,997.88	$ 5,119.10
Robert King	$ 18,104.80	$ 13,703.40	$ 3,891.00	$ 3,867.20	$ 5,707.35	$ 6,041.25	$ 2,082.00	$ 6,144.60	$ 7,853.05
Steven Buchanan	$ 21,965.20			$ 2,634.40		$ 5,127.50	$ 3,124.90	$ 6,475.40	$ 3,636.70
Grand Total	$ 226,298.50	$66,692.80	$41,207.20	$39,979.90	$55,699.39	$56,823.70	$39,088.00	$55,464.93	$49,981.69

Figure 45. When you pivot on multiple criteria, Excel's pivot tables let you collapse and expand on the higher-level values. Here, the 1996 data has been collapsed to a single column.

Excel's pivot tables also let you filter and sort, based on either row or column values. **Figure 46** shows the dropdown that appears when you click the arrow on the Salesperson header.
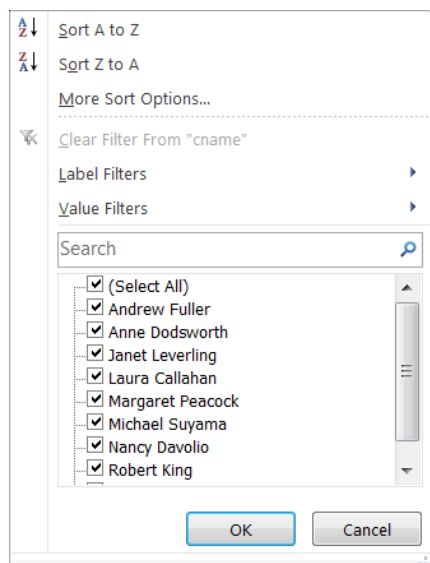
Figure 46. Excel's pivot tables let you filter and sort, based on row or column values. This window opens when you click the dropdown arrow on the Salesperson header.

Note that you can filter on either the label or the value. That is, you can select specific salespeople, or select only those salespeople in a certain part of the alphabet (which seems silly), or select only those with total sales (in the Grand Total column at the very right edge of the pivot table) in a certain range. **Figure 47** shows the pivot table with all three years collapsed and with a filter of total sales greater than or equal to $100,000. Note the icon on the Salesperson header that lets you know you're seeing filtered data.

	A	B	C	D	E
1					
2					
3	Orders ($)	Months/Years ▾			
4		⊞ 1996	⊞ 1997	⊞ 1998	Grand Total
5	Salesperson ⊺▾				
6	Andrew Fuller	$22,835	$74,959	$79,956	$177,749
7	Janet Leverling	$19,232	$111,789	$82,031	$213,051
8	Laura Callahan	$23,161	$59,777	$50,363	$133,301
9	Margaret Peacock	$53,115	$139,478	$57,595	$250,187
10	Nancy Davolio	$38,789	$97,534	$65,821	$202,144
11	Robert King	$18,105	$66,689	$56,502	$141,296
12	Grand Total	$175,237	$550,224	$392,268	$1,117,729

Figure 47. You can filter a pivot table based on the computed data.

When there are multiple criteria, as with the columns in this example, you can choose which criterion you're sorting or filtering on. **Figure 48** shows the dropdown that lets you pick the field to which the rest of the dialog applies.
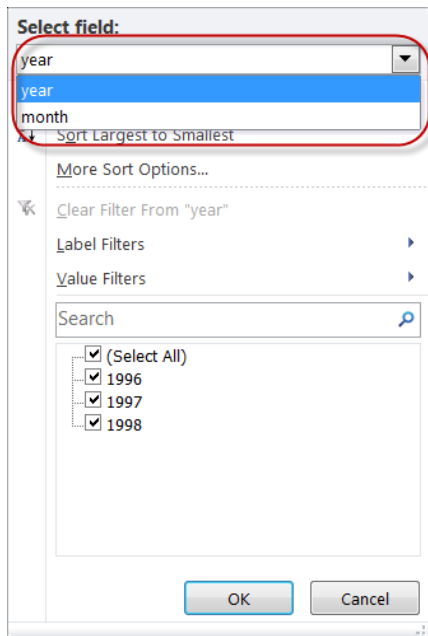
Figure 48. When you use multiple criteria to specify rows or columns, you can choose which to sort or filter on.

Creating pivot tables programmatically

If you can do it interactively in Excel, you can almost always do it programmatically. While I've written (for example, http://tinyurl.com/ybhfsdfc) that recording a macro can result in bad code, when you're first trying to automate a task and have no idea what objects are involved, recording a macro can get you started; that's what I did to figure out how to create a pivot table. In fact, I'd hardly looked at Excel's pivot tables before working on this session, so I started out using Excel's Pivot Table Wizard to figure out how to create them at all. (It wasn't obvious to me how to get started; on the Insert tab, choose PivotTable and then choose PivotTable from the submenu.) Once I felt comfortable, I recorded a macro using the Wizard, and then adapted that code in VFP.

The first step in creating a pivot table is to send the raw data to Excel. Since creating a pivot table implies that Excel is available, I chose to simply create a CSV file with COPY TO, as in **Listing 49**.

Listing 49. The first step in creating a pivot table is sending the raw data to Excel.

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

SELECT EmployeeID, ;
       YEAR(OrderDate) AS Year, ;
       MONTH(OrderDate) as Month, ;
       SUM(Quantity*UnitPrice) AS OrderTotal ;
   FROM Orders ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
   GROUP BY 1, 2, 3 ;
   INTO CURSOR csrMonthlyTotals
```

```
* Add employee name
SELECT PADR(ALLTRIM(FirstName) + (' ' + LastName), 30) AS cName, ;
       csrMonthlyTotals.* ;
  FROM csrMonthlyTotals ;
    JOIN Employees ;
      ON csrMonthlyTotals.EmployeeID = Employees.EmployeeID ;
  ORDER BY LastName, FirstName ;
  INTO CURSOR csrReport

LOCAL cCSVFile
cCSVFile = FORCEPATH(FORCEEXT("SalesPersonMonthly", "CSV"), SYS(2023))

COPY TO (m.cCSVFile) TYPE csv
```

Once the data is in Excel, the simplest way to create a pivot table is to first create a PivotCache object. A PivotCache is an object containing the data you want to put in the pivot table. (Besides being easy, creating a PivotCache first lets you use the same data for multiple pivot tables.) The PivotCache object has a CreatePivotTable method that does most of the heavy lifting. In **Listing 50**, the CSV file is opened in Excel, and the PivotCache and PivotTable objects are created. Using Excel's UsedRange object to specify the data avoids having to know exactly which rows and columns are in the exported data.

Listing 50. Creating the pivot table object is straightforward, using a PivotCache.

```
LOCAL oExcel AS Excel.Application, ;
      oWorkbook as Excel.Workbook, ;
      oSheet AS Excel.Worksheet, ;
      oPC AS Excel.PivotCache, ;
      oPT AS Excel.PivotTable, ;
      oRange AS Excel.Range

oExcel = CREATEOBJECT("Excel.Application")
oWorkbook = oExcel.Workbooks.Open(m.cCSVFile)
oExcel.Visible = .T.

* Adapted from PivotTable macro
oRange = oExcel.ActiveSheet.UsedRange()
oSheet = oExcel.Sheets.Add()
oSheet.Name = "SalesPivot"
oPC = oExcel.ActiveWorkbook.PivotCaches.Create(1, m.oRange, 4)
        && First param: 1=xlDatabase
        && Third param: 5=xlPivotTable15; 4=xlPivotTable14

oPT = oPC.CreatePivotTable("SalesPivot!R3C1", "PivotTable2", .T., 4)
```

The PivotTable object created is empty. If you look at the workbook at this point, the worksheet exists, but there's no data there. Instead, you see a prompt from Excel, as in **Figure 49**. If you click in that area, the Pivot Table Wizard pane appears so you can specify the layout of the pivot table.
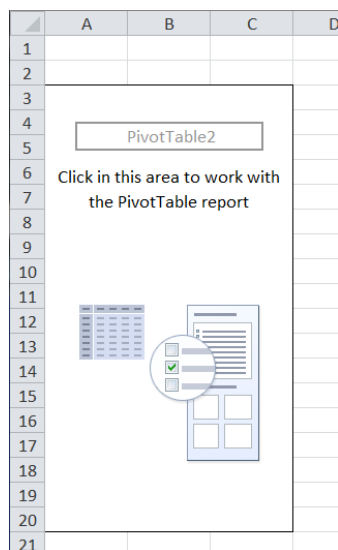
Figure 49. After creating a pivot table, Excel prompts you to populate it.

Of course, we don't want to work with the wizard; we want to do this programmatically. To do that, add one or more data fields using PivotTable's AddDataField method. The method accepts three parameters: the field to add, a caption for it, and a function to apply to the field. Only the first parameter is required.

The secret to providing the pivot field is PivotTable's PivotFields collection, which contains all the fields put into the PivotCache; they're (initially) named by the column headers in the original data. In our example, there are five: cname, employeeid, year, month, and ordertotal. The only field there that we want to treat as data is ordertotal, so that's the one we specify in the AddDataField method.

The caption parameter specifies the string that will appear in the upper-left corner of the pivot table. In our example, we want "Orders ($)." Omitting this parameter uses the field name as the caption. (Be aware that once you change the caption for a field, that becomes the name you use to refer to that field in the PivotFields collection.)

The function parameter is based on the xlConsolidationFunction enumeration; the most common values are shown in **Table 4**. Other available functions include variance and standard deviation, as well as some variations on count. If you omit this parameter, you get a sum.

Table 4. You can specify what aggregation function to use on a data field.

Function	Excel constant	Value
Average	xlAverage	-4106
Count	xlCount	-4112
Maximum	xlMax	-4136
Minimum	xlMin	-4139
Sum	xlSum	-4157

Listing 51 shows the code to add ordertotal as a data field to the pivot table.

Listing 51. It takes only one line of code to set up a data field in the pivot table.

```
oPT.AddDataField( oPT.PivotFields("ordertotal"), "Orders ($)", -4157) && xlSum
```

The other thing we need to do is specify which fields are columns and which are rows. The easiest way to do this is set the relevant properties for each field individually. The two key properties are Orientation and Position. Orientation determines whether the field is used for rows (1 = xlRowField) or for columns (2 = xlColumnField). Position determines the priority of the row or column when multiple rows or columns are specified. **Listing 52** shows the settings for our example, making cname a row field, and year and month column fields, with year coming first.

Listing 52. Set the orientation and position properties of the items in the PivotFields collection to specify the rows and columns you want to show in the pivot table.

```
With oPT.PivotFields("cname")
   .Orientation = 1   && xlRowField
   .Position = 1
EndWith
With oPT.PivotFields("year")
   .Orientation = 2 && xlColumnField
   .Position = 1
ENDWITH
With oPT.PivotFields("month")
   .Orientation = 2 && xlColumnField
   .Position = 2
EndWith
```

At this point, you have a pivot table showing the relevant data. The final steps are cosmetic: setting the row and column descriptions and formatting the data properly.

The CompactLayoutRowHeader and CompactLayoutColumnHeader properties of the pivot table specify the headers for the rows and columns, respectively. These are also the items that contain the dropdowns for sorting and filtering.

Formatting all the data cells in a pivot table is surprisingly simple; just set the NumberFormat property of the relevant member of the PivotFields collection. The code in **Listing 53** sets the headers and formats the data cells.

Listing 53. Formatting a pivot table takes just a few lines of code.

```
oPT.CompactLayoutRowHeader = "Salesperson"
oPT.CompactLayoutColumnHeader = "Months/Years"
oPT.PivotFields("Orders ($)").NumberFormat = "$#,#0"
```

The complete code to create the pivot table shown in **Figure 44** is included in the downloads for this session as PivotExcelSalespersonMonthly.prg.

You can do lots more with pivot tables, including specifying multiple pivot fields, and including a third dimension (pages). You'll find documentation for the PivotTable object at http://tinyurl.com/y8dzgcft.

Charting crosstabs

While reports and spreadsheets are useful, especially for those who like number crunching, for many users, graphs and charts are far more informative. There are two straightforward ways to chart this data. Excel has strong graphing capabilities and includes a tool for building what it calls "Pivot Charts." If you need graphs inside your VFP application, VFPX's FoxCharts gives you what you need.

We'll look at sales by category, including tracking by country. **Listing 54** shows the query that collects the raw data, plus a couple of additional columns. **Figure 50** shows partial results.

Listing 54. This query collects information about sales by product, category, year and country.

```
SELECT ProductName, ;
       CategoryName, ;
       YEAR(OrderDate) AS Year, ;
       ShipCountry,  ;
       SUM(Quantity) AS NumSold ;
   FROM Orders ;
     JOIN OrderDetails OD ;
       ON Orders.OrderID = OD.OrderID ;
     JOIN Products ;
       ON OD.ProductID = Products.ProductID ;
     JOIN Categories ;
       ON Products.CategoryID = ;
          Categories.CategoryID ;
   GROUP BY 1, 2, 3, 4 ;
   INTO CURSOR csrProductsSold
```

Csrproductssold				
Productname	Categoryname	Year	Shipcountry	Numsold
Alice Mutton	Meat/Poultry	1996	Belgium	40
Alice Mutton	Meat/Poultry	1996	Canada	70
Alice Mutton	Meat/Poultry	1996	France	30
Alice Mutton	Meat/Poultry	1996	Germany	15
Alice Mutton	Meat/Poultry	1996	Mexico	8
Alice Mutton	Meat/Poultry	1996	USA	71
Alice Mutton	Meat/Poultry	1997	Austria	191
Alice Mutton	Meat/Poultry	1997	Canada	50
Alice Mutton	Meat/Poultry	1997	Italy	20
Alice Mutton	Meat/Poultry	1997	Mexico	18
Alice Mutton	Meat/Poultry	1997	Spain	48
Alice Mutton	Meat/Poultry	1997	Sweden	10
Alice Mutton	Meat/Poultry	1997	UK	25
Alice Mutton	Meat/Poultry	1997	USA	165
Alice Mutton	Meat/Poultry	1998	Brazil	27
Alice Mutton	Meat/Poultry	1998	Canada	6
Alice Mutton	Meat/Poultry	1998	France	37
Alice Mutton	Meat/Poultry	1998	Mexico	10
Alice Mutton	Meat/Poultry	1998	Spain	12
Alice Mutton	Meat/Poultry	1998	USA	125
Aniseed Syrup	Condiments	1996	UK	30
Aniseed Syrup	Condiments	1997	Austria	20
Aniseed Syrup	Condiments	1997	Canada	20
Aniseed Syrup	Condiments	1997	Denmark	14
Aniseed Syrup	Condiments	1997	Germany	66
Aniseed Syrup	Condiments	1997	Venezuela	70

Figure 50. The query in **Listing 54** collects data for sales of each product by year and country and includes the product's category.

Creating Excel's Pivot Charts

Excel's Insert tab includes both Pivot Table and Pivot Chart. Choosing Pivot Chart lets you specify a data range and then opens the Pivot Table Wizard with an added element for the chart itself. As you work through the wizard, you create both a pivot table and a graphical representation of the data. For example, **Figure 51** shows a graph of sales by category and country; you can see part of the pivot table behind the graph.
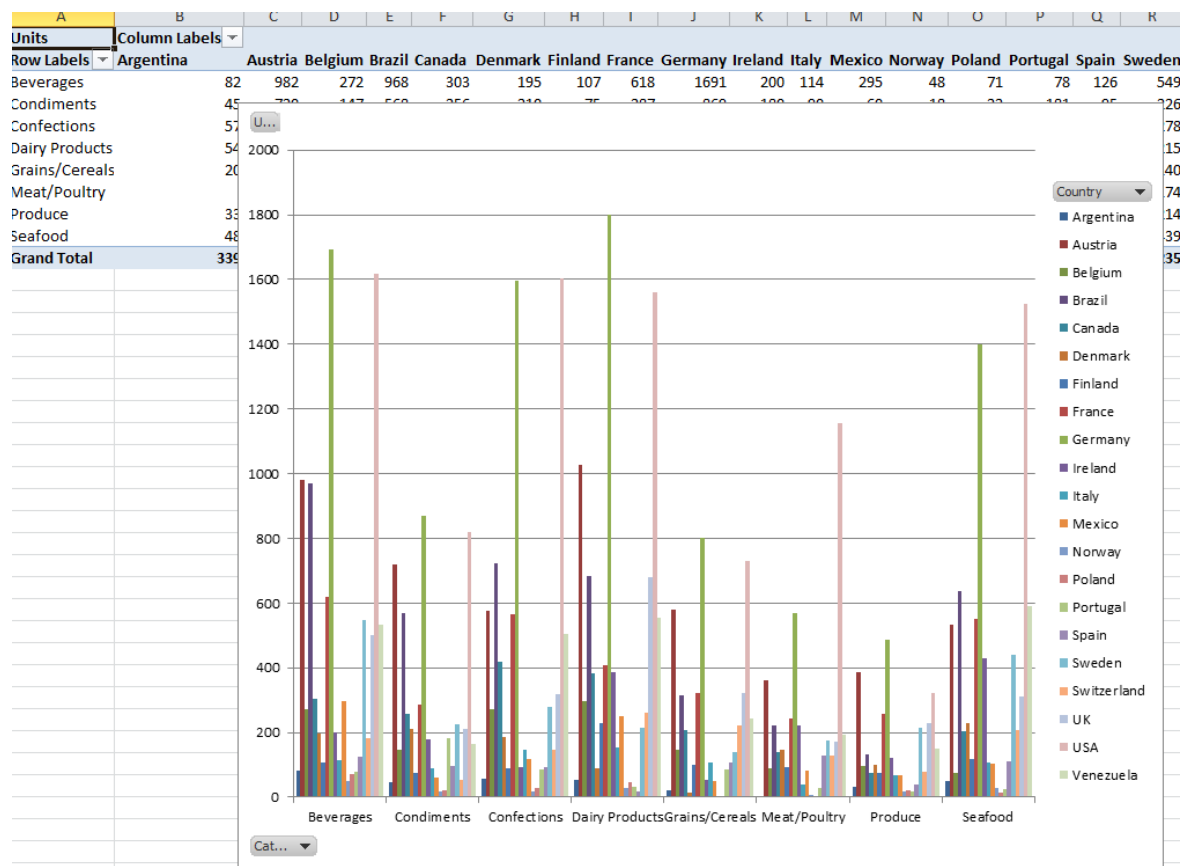
Figure 51. As you create an Excel pivot graph using the wizard, a pivot table is created as well.

You create a pivot graph programmatically by creating a pivot table and then adding a chart. The AddChart method of the Shapes collection adds a chart to the worksheet. It accepts five parameters. The first is the chart type (from the xlChartType enumeration). The other four are for positioning the chart: left, top, width, height. Excel is smart enough to make the chart based on the pivot table you just created without you having to do anything special.

The code in **Listing 55** creates the pivot table and chart. The complete code (including the query) is included in the downloads for this session as PivotProductSales.prg. As with the previous examples, the data is exported to a CSV file and then opened in Excel. Then, a PivotCache is created containing all the data, and used to create a PivotTable. Finally, AddChart is called. Chart type 51, which is the default when you work interactively, creates a vertical bar chart (which Excel calls a "column chart") with a separate bar for each data point.

Listing 55. This code creates the pivot table and pivot graph shown in **Figure 51**.

```
LOCAL cCSVFile
cCSVFile = FORCEPATH(FORCEEXT("ProductSales", "CSV"), SYS(2023))

COPY TO (m.cCSVFile) TYPE csv
```

```
LOCAL oExcel AS Excel.Application, ;
      oWorkbook as Excel.Workbook, ;
      oSheet AS Excel.Worksheet
LOCAL oPC AS Excel.PivotCache, ;
      oPT AS Excel.PivotTable
LOCAL oChart AS Excel.Chart, ;
      oRange AS Excel.Range

oExcel = CREATEOBJECT("Excel.Application")
oWorkbook = oExcel.Workbooks.Open(m.cCSVFile)
oExcel.Visible = .T.

oSheet = oExcel.Sheets.Add()
oSheet.Name = "SalesByCategoryAndCountry"

oRange = oExcel.ActiveWorkbook.Worksheets("ProductSales").UsedRange()
oPC = oExcel.ActiveWorkbook.PivotCaches.Create(1,  m.oRange, 4) && 1 = xlDatabase
oPT = oPC.CreatePivotTable("SalesByCategoryAndCountry!R1C1", "PivotTable1", .T., 4)
oPT.AddDataField(oPT.PivotFields("numsold"), "Units", -4157) && xlSum
WITH oPT.PivotFields("categoryname") AS Excel.PivotField
   .Orientation = 1 && xlRowField
   .Position = 1
   .Caption = "Category"
ENDWITH
WITH oPT.PivotFields("shipcountry") ;
  AS Excel.PivotField
   .Orientation = 2 && xlColumnField
   .Position = 1
   .Caption = "Country"
ENDWITH

oChart = oSheet.Shapes.AddChart(51, 150, 50, 600, 300) && 51 = xlColumnClustered

RETURN
```

As **Figure 51** shows, pivot charts offer the same sorting and filtering options as pivot tables. The Category and Country dropdowns let you sort the bars and let you filter some out based on either their labels or their values. As described in the previous section of this paper, value filters work only on the grand total, not the values in the individual columns.

To add a filter, call the Add method of the PivotFilters collection for the row or column you want to filter on. For example, adding the line in **Listing 56** to the end of the previous example produces the result in **Figure 52**. The first parameter indicates the type of filter, based on the xlPivotFilterType collection; 1 is a top count. (The collection is documented at http://tinyurl.com/ycwr2mtj.) The second parameter specifies the field to apply the filter to; here, it's the number of units sold. The third parameter is the value for the filter; here, we're asking for the top five. The result is the top five countries by total units sold. (The method has additional parameters not relevant in this case. The one you're most likely to use is a second value parameter, for cases where you're filtering for information between two values.) A version of the code setting this filter is included in the downloads for this session as PivotProductSalesFiltered.prg.

Listing 56. You can filter the data in the pivot table and pivot chart using the PivotFilters collection.

```
oPT.PivotFields("Country").PivotFilters.Add(1, oPT.PivotFields("units"), 5)
```
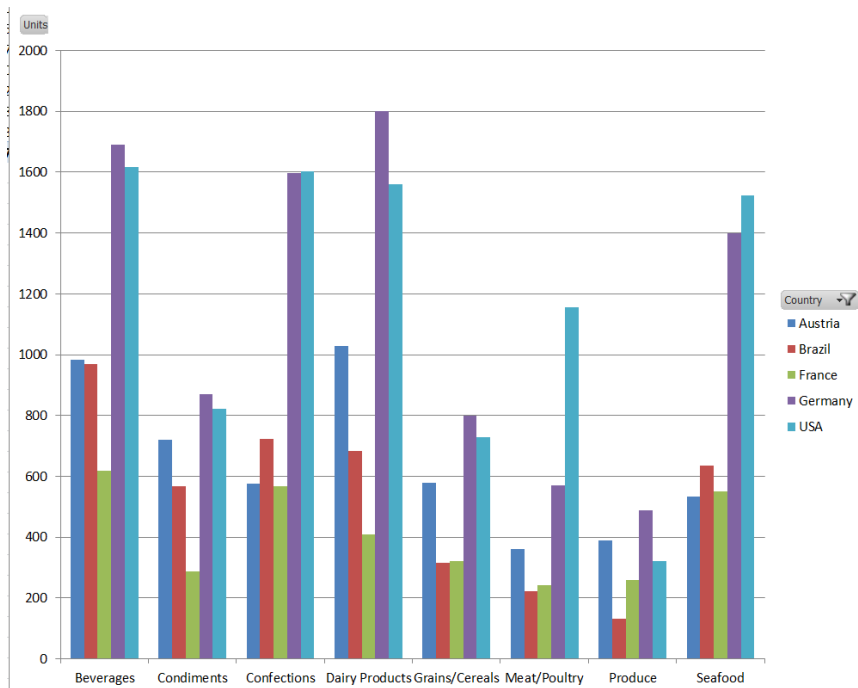

Figure 52. We can reduce the amount of data in the chart by filtering.

If you instead add a filter to choose the top three by category, as in **Listing 57**, you get a very different result, shown in **Figure 53**. (Complete code for this example is included in the downloads for this session as PivotProductSalesFiltered2.prg.)

Listing 57. This line filters for the top three categories by total sales.

```
oPT.PivotFields("Category").PivotFilters.Add(1, oPT.PivotFields("units"), 3)
```
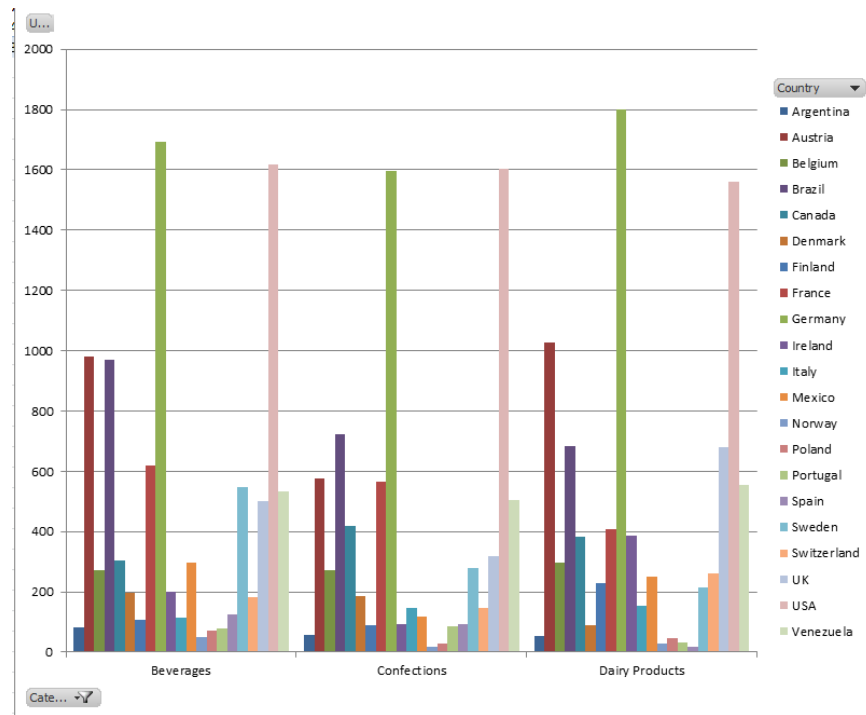
Figure 53. You can filter to show only the top-selling categories.

You can add multiple filters, resulting in reducing the data shown even more. It's also worth noting that this programmatic approach to filtering applies to pivot tables whether you chart them or not.

Creating other types of charts

Of course, you can create other types of charts and graphs. Interactively, you right-click on the chart and choose Change Chart Type. Programmatically, you simply pass a different value for the first parameter of the AddChart method.

Excel offers many types of charts. In **Figure 54**, some of the xlChartType enumeration is shown in the VFP Object Browser. The enumeration is documented at http://tinyurl.com/ydxz3vb3.
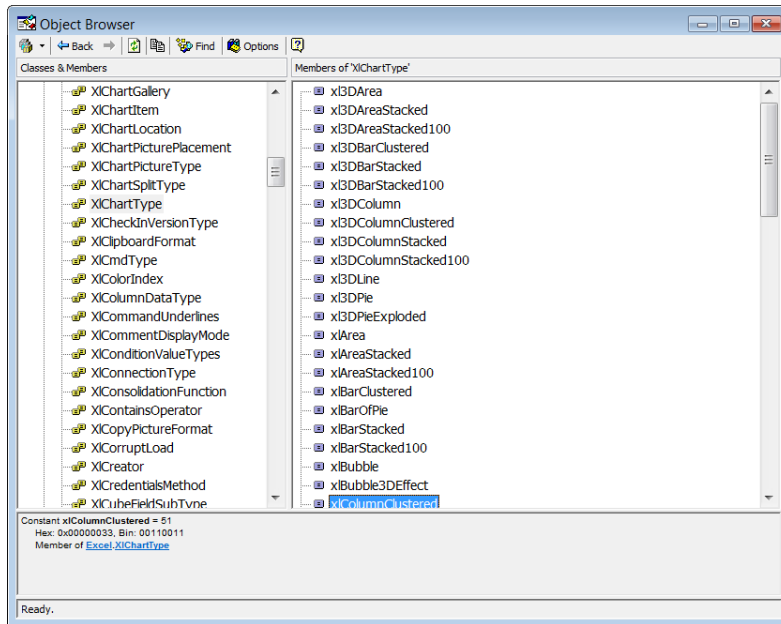
Figure 54. VFP's Object Browser lets you explore Excel's enumerations. Here, part of the list for xlChartType is shown.

The code in **Listing 58** builds a pie chart showing each category's share of sales. (It assumes the data has already been collected, exported from VFP and imported into Excel. The complete program is included as ProductSalesByCategoryPie.prg in the downloads for this session.) **Figure 55** shows the result.

Listing 58. The first parameter of the AddChart method determines the type of chart.

```
oSheet = oExcel.Sheets.Add()
oSheet.Name = "SalesByCategoryAndCountry"

oRange = oExcel.ActiveWorkbook.Worksheets("ProductSales").UsedRange()
oPC = oExcel.ActiveWorkbook.PivotCaches.Create(1, m.oRange, 4)
oPT = oPC.CreatePivotTable("SalesByCategoryAndCountry!R1C1", "PivotTable1", .T., 4)
oPT.AddDataField(oPT.PivotFields("numsold"), "Units", -4157) && xlSum
WITH oPT.PivotFields("categoryname") AS Excel.PivotField
   .Orientation = 1 && xlRowField
   .Position = 1
   .Caption = "Category"
ENDWITH

oChart = oSheet.Shapes.AddChart(70, 150, 50, 600, 300) && 70 = xl3DPieExploded
```
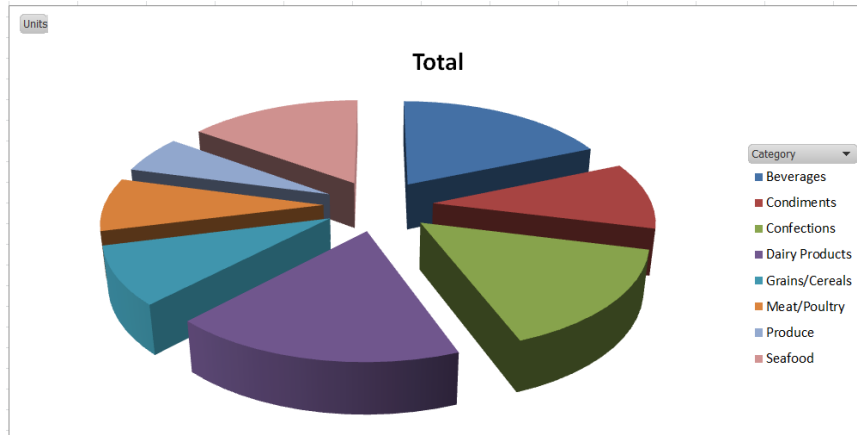
Figure 55. This chart shows the share of sales for each category.

With dozens of chart types, you should be able to find one that helps your users understand their data.

Producing graphs and charts in VFP with FoxCharts

FoxCharts is one of the premiere tools available from VFPX, the community open-source project for VFP. It uses GDIPlusX (another VFPX project) to provide a fairly easy way to put graphs and charts into VFP forms.

You can install FoxCharts through Thor or directly from VFPX's on GitHub. The direct link is https://github.com/VFPX/FoxCharts.

The download site includes a 43-page paper from Jim Nelson. In addition, Doug Hennig has a great paper on his website that shows you how to get started and gives you an idea as to its capabilities: (http://doughennig.com/papers/Pub/FoxCharts.pdf). With all that documentation available, I won't go into detail on getting started. Instead, I'll focus on the kinds of charts you're likely to want to create from crosstab data.

In general, to use FoxCharts, you drop the FoxCharts class on a form and set properties and add code. (FoxCharts also comes with a tool that lets you create charts interactively. You may find that easier.) To make it easy for you to replicate my examples, I'm doing most of the work in code rather than setting properties of the chart and the form in the Property Sheet.

However, I created a form class that includes the FoxCharts object and names it cntChart. The form also has a custom method called MakeChart and a call to that method in the form's Init method, as in **Listing 59**. The library containing the form class is included in the downloads for this session as GranT060.VCX; if you use it, you'll need to point to the location where you've installed FoxCharts.

Listing 59. To encapsulate all the code that populates the chart, the form's Init method calls a custom MakeChart method.

```
This.MakeChart()
```

Creating a pie chart

We'll start by replicating the pie chart from **Figure 55** (though it doesn't use a crosstab). The first step is collecting the necessary data. FoxCharts is a little finicky about its data and it's best if the cursor you supply contains only data you want to graph, so this form's MakeChart method starts with the query in **Listing 60**.

Listing 60. This query collects the data needed to create a pie chart of units sold by category.

```
SELECT CategoryName, ;
       SUM(Quantity) AS NumSold ;
   FROM Orders ;
     JOIN OrderDetails OD ;
       ON Orders.OrderID = OD.OrderID ;
     JOIN Products ;
       ON OD.ProductID = Products.ProductID ;
     JOIN Categories ;
       ON Products.CategoryID = Categories.CategoryID ;
   GROUP BY 1  ;
   INTO CURSOR csrProductsSold
```

FoxCharts lets you specify which slices should be exploded individually by providing a column in the underlying cursor. (There are other options for exploding pie charts, as well. You can indicate that a slice explodes when you click on it, or when you click on its legend, rather than specifying a fixed list.) So, the next step is to add that column to the cursor, as in **Listing 61**.

Listing 61. The added lDetach column lets us specify which slices of the pie should be "exploded," that is, pulled outward from the chart. Here, all columns should be exploded.

```
SELECT *, .T. AS lDetach ;
   FROM csrProductsSold ;
   INTO CURSOR csrProductsSold
```

Once we have data, we set properties of the cntChart object (as well as setting the form's Caption). **Listing 62** shows the code.

Listing 62. This code tells the FoxCharts object what type of chart to draw, what data to use, and more.

```
ThisForm.Caption = "Units sold by category"

WITH This.cntChart
   .Anchor = 15
   .ChartType = 1 && Pie
   .ColorType = 0

   .SourceAlias = "csrProductsSold"

   .ChartsCount = 1
   .Fields(1).FieldValue = "NumSold"
   .FieldLegend = "CategoryName"
```

```
    .Title.Caption = 'Units Sold'
    .Subtitle.Caption = ''
    .XAxis.Caption = 'Category'
    .YAxis.Caption = 'Units sold'

    .FieldDetachSlice = "lDetach"

    .DrawChart()

ENDWITH
```

After setting the form caption, we anchor the FoxCharts object to the form, so that resizing the form resizes the chart.

Next, we indicate that we want a pie chart (ChartType = 1) and that we should use the basic color set (ColorType = 0).

The SourceAlias property tells the chart where its data comes from, though it doesn't specify which data from the cursor to graph.

The ChartsCount property sounds like it specifies how many charts you're creating, but it actually indicates how many data series are to be specified. The value you provide is used to set the size of the Fields collection. So, after indicating we have only one data series (which is normal for a pie chart), we indicate the data for that series (the NumSold field of the cursor) by setting the FieldValue property of Fields(1), the first element in the Fields collection. The FieldLegend property specifies the field in the data source that provides the values to appear in the legend for the chart; here, it's the CategoryName column.

Next, we specify a title for the chart. As you can see, you can have both a title and a subtitle; a subtitle seemed like overkill for this simple example.

FieldDetachSlice applies to pie charts and indicates the field in the data source that specifies whether a given slice should be exploded, so we specify the lDetach column.

Finally, we call the DrawChart method to create the chart. The complete code for this example is included in the downloads for this session as CategorySalesPie.SCX. The resulting form is shown in **Figure 56**.
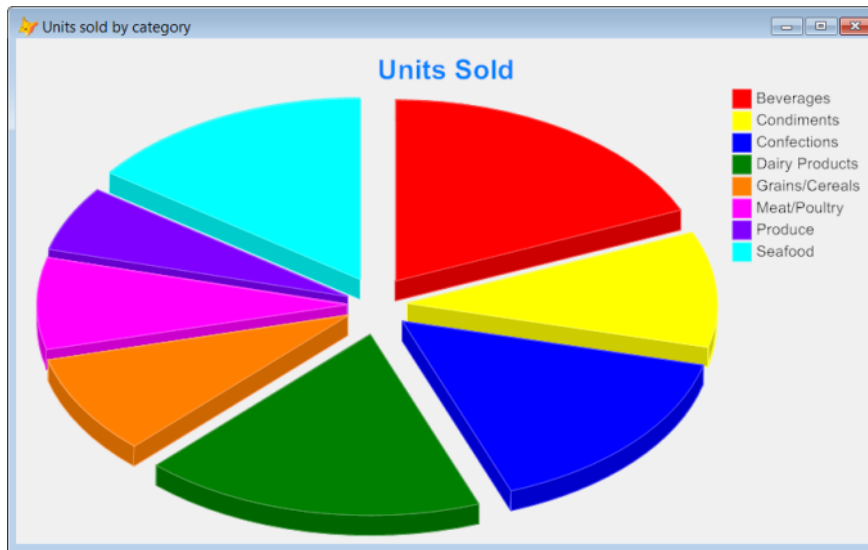
Figure 56. It takes only a little code to build this pie chart with FoxCharts.

Working with multiple data series

To graph crosstab results, we need to work with multiple data series. Each column created by a crosstab becomes a data series. The goal is to create a chart similar to **Figure 51**; the result is shown in **Figure 57**.
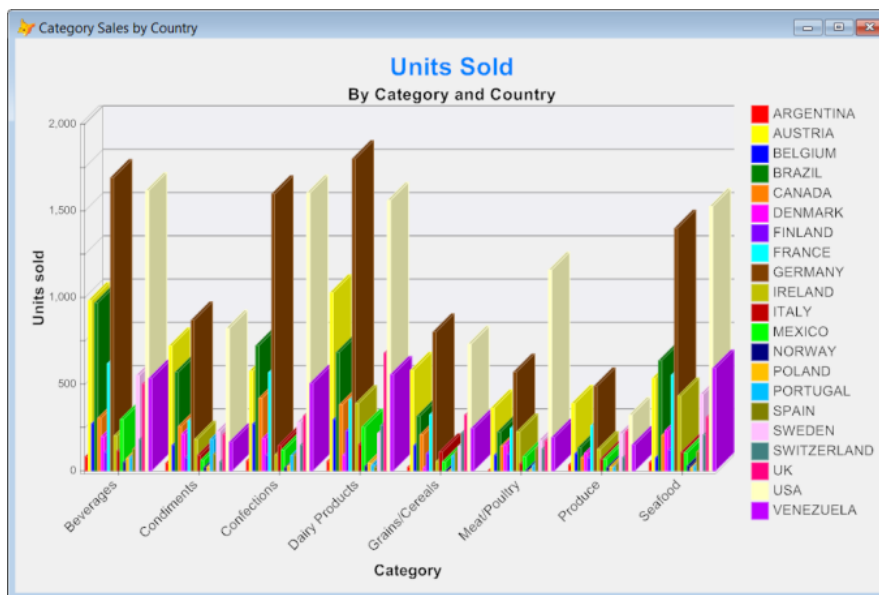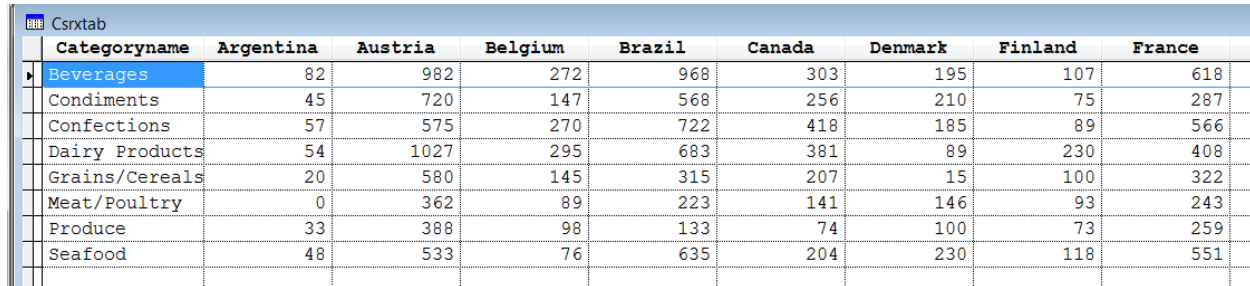

Figure 57. This is the FoxCharts version of the bar chart showing sales by country for each category.

As in the previous example, first we need to collect the data. For this chart, we start with the query in **Listing 54**, and then crosstab it. **Listing 63** shows the code that creates the crosstab cursor from the original query results. **Figure 58** shows partial results.

Listing 63. To create a chart showing units sold by category and country, we need to crosstab the data.

```
LOCAL oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
```

```
oXTab = NEWOBJECT("fastxtab", "fastxtab16\fastxtab.prg")
WITH oXTab AS FastXTab OF "fastxtab16\fastxtab.prg"
    .cOutFile = "csrXtab"
    .cRowFIELD = "CategoryName"
    .cColField = "ShipCountry"
    .cDATAFIELD = "NumSold"
    .lCursorOnly = .T.
    .lCLOSETABLE = .T.
    .RunXtab()
ENDWITH
```

Categoryname	Argentina	Austria	Belgium	Brazil	Canada	Denmark	Finland	France
Beverages	82	982	272	968	303	195	107	618
Condiments	45	720	147	568	256	210	75	287
Confections	57	575	270	722	418	185	89	566
Dairy Products	54	1027	295	683	381	89	230	408
Grains/Cereals	20	580	145	315	207	15	100	322
Meat/Poultry	0	362	89	223	141	146	93	243
Produce	33	388	98	133	74	100	73	259
Seafood	48	533	76	635	204	230	118	551

Figure 58. This is part of the data to be graphed. There's one column for each country.

Once we have the data, as before, we set properties of the FoxCharts object. In this case, we want to set ChartCount to the number of data columns (that is, the number of countries) in the cursor. Once we do that, we need to provide information about each column. **Listing 64** shows the code to specify the chart; the complete program for this chart is included in the downloads for this session as CategorySalesByCountry.SCX.

Listing 64. To create the chart in Figure 57, we need to set up a data series for each country.

```
LOCAL nCountries
nCountries = FCOUNT("csrXTab") - 1

This.Caption = "Category Sales by Country"

WITH This.cntChart
    .Anchor = 15
    .ChartType = 8 && MultiBar
    .ColorType = 0

    .SourceAlias = "csrXTab"
    .FieldAxis2 = "CategoryName"
    .AxisLegend2.Rotation = -45
    .AxisLegend2.Alignment = 1 && Right

    .ChartsCount = m.nCountries

    LOCAL nSeries, nFirstDataCol
    nFirstDataCol = 2

    FOR nSeries = 1 TO m.nCountries
        WITH .Fields(m.nSeries)
```

```
         .FieldValue = FIELD(m.nSeries + m.nFirstDataCol -1 )
         .Legend = .FieldValue
      ENDWITH
   ENDFOR

   .Title.Caption = 'Units Sold'
   .Subtitle.Caption = 'By Category and Country'
   .XAxis.Caption = 'Category'
   .YAxis.Caption = 'Units sold'

   .DrawChart()

ENDWITH
```

To create a bar chart with multiple data series, we set ChartType to 8. Bar charts have legends on their axes, as well as an optional block legend (like the one for the pie chart). For this chart, the Y-axis should have numeric values representing the data; we don't have to specify anything for that to happen. But the X-axis should be labelled with the category names; that's what we get by setting the FieldAxis2 property. The two lines after that angle the labels and right-justify them, so they don't overlap.

The key part of the code is the loop, which sets properties of the members of the Fields collection, based on the data columns in the crosstab. Setting the Legend property of the fields specifies that there should be a block legend (which, in this case, shows the colors for the countries).

Lots more options
Charts created with FoxCharts don't offer the ability for live filtering that Excel does, but by default, they include tooltips that show the value of a given item when the mouse hovers over it, as in **Figure 59**. (Set the FoxCharts object's ShowTips property to .F. to turn these off.) Some chart types have other dynamic options, as well.
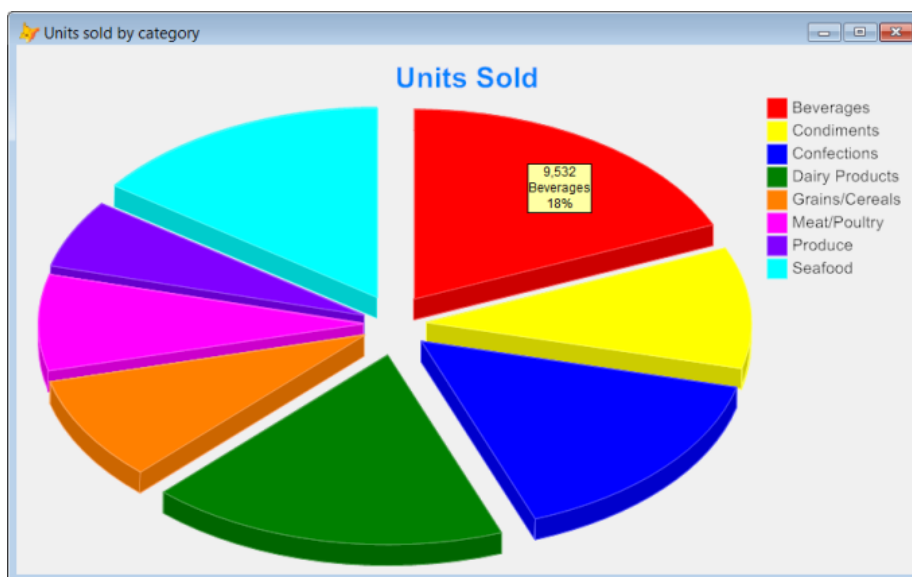

Figure 59. By default, FoxCharts shows tooltips with data values.

FoxCharts offers many more chart types, as well as control over colors and much more.

In addition, though we've looked at putting charts on forms, they can also be printed. See the documentation on GitHub for details.

Wrapping up

Crosstabs and pivots are a valuable way to present data. Both VFP and SQL Server make it reasonably easy to convert normalized data into crosstabs and pivots. Once you've converted the data, there are lots of options for presenting it to users through reports, spreadsheets, and charts.

All the tools described in this paper, whether for creating crosstabs or reporting on them, have many more options than are covered here. If the examples here don't give you exactly what you need, it's worth spending some time to figure out whether the tools can do so.