Getting the Top N for each Group

Several features of SQL Server combine to make a hard problem in VFP easy.

Tamar E. Granor, Ph.D.

In my previous articles in this series, I've explored several features in SQL Server that don't have analogues in VFP. These features make it much easier to solve certain problems. This article introduces another such feature that simplifies another common problem.

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

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

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

Figure 1. The query in Listing 1 produces the total sales for each employee by year.

The VFP solution

Collecting the basic data you need to solve this problem is straightforward. **Listing 1** (EmployeeSalesByYear.PRG in this month's downloads) shows a query that provides each employees sales by year; **Figure 1** shows part of the results. (The VFP examples in this article use the Northwind database.)

Listing 1. Getting total sales by employee by year is easy in VFP.

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

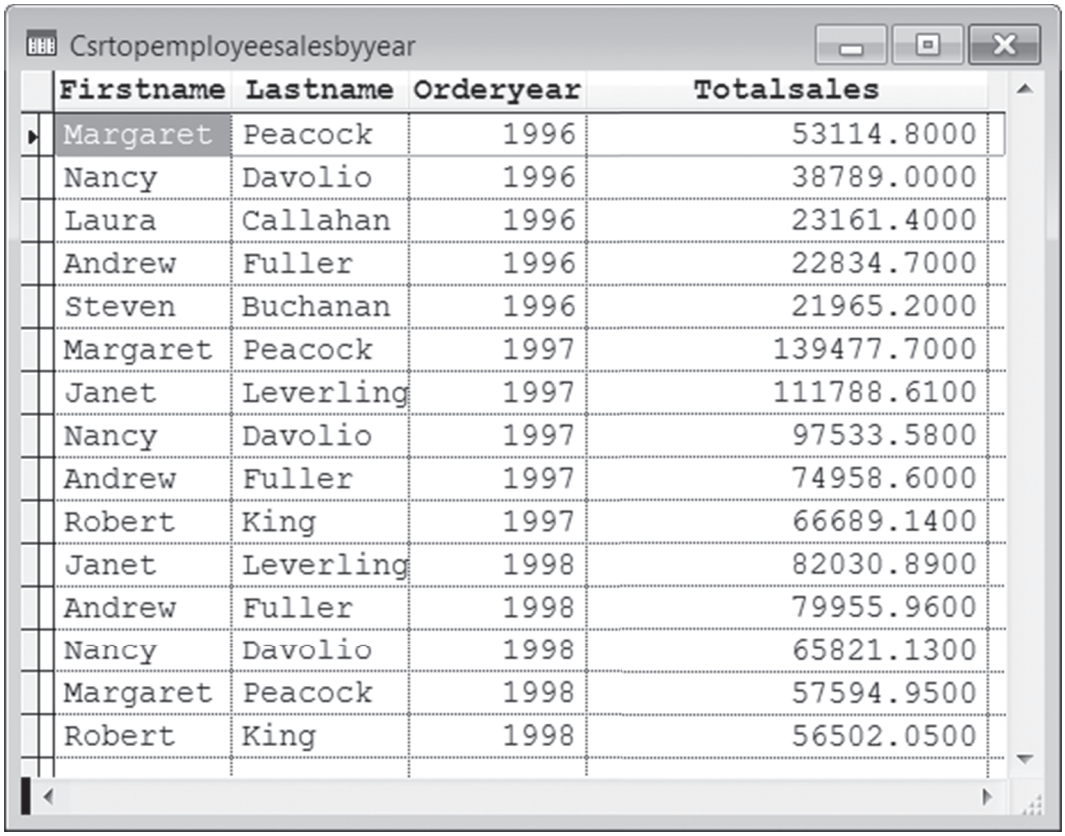
However, when you want to keep only the top five for each year, you need to either combine SQL code with some Xbase code or use a bit of a trick that can result in a significant slowdown with large datasets.

SQL plus Xbase

The mixed solution is easier to follow, so let's start with that one. The idea is to first select the raw data needed, in this case, the total sales by employee by year. Then we loop through on the grouping field, and select the top n (five, in this case) in each group and put them into a cursor. Listing 2 (TopnEmployeeSalesByYear-Loop.PRG in this month's downloads) shows the code; Figure 2 shows the result.

Listing 2. One way to find the top n in each group is to collect the data, then loop through it by group.

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



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

Figure 2. The query in **Listing 2** produces these results.

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

SELECT distinct OrderYear ;
  FROM csrEmpSalesByYear ;
  ORDER BY OrderYear ;
  INTO CURSOR csrYears

LOCAL nYear

SCAN
  nYear = csrYears.OrderYear

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

ENDSCAN

USE IN csrYears
USE IN csrEmpSalesByYear
SELECT csrTopEmployeeSalesByYear
```

The first query is just a simpler version of Listing 1, omitting the Employees table and the ORDER BY clause; both of those will be used later. Next, we create a cursor to hold the final results. Then, we get a list of the years for which we have data. Finally, we loop through the cursor of years and, for each, grab the top five salespeople for that year, and put them into the result cursor, adding the employee's name and sorting as we insert.

You can actually consolidate this version a little by turning the first query into a derived table in the query inside the INSERT command. Listing 3 (TopnEmployeeSalesByYear-Loop2.PRG in this month's downloads) shows the revised version. Note that you have to get the list of years directly from the Orders table in this version. This version, of course, gives the same results.

Listing 3. The code in **Listing 2** can be reworked to use a derived table to compute the totals for each year.

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

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

LOCAL nYear

SCAN
  nYear = csrYears.OrderYear

  INSERT INTO csrTopEmployeeSalesByYear ;
    SELECT TOP 5 ;
      FirstName, LastName, ;
      OrderYear, TotalSales ;
    FROM Employees ;
     JOIN ( ;
      SELECT EmployeeID, ;
             YEAR(OrderDate) as OrderYear, ;
```

```
            SUM(UnitPrice * Quantity) ;
              AS TotalSales ;
        FROM Orders ;
         JOIN OrderDetails ;
          ON Orders.OrderID = ;
             OrderDetails.OrderID ;
         WHERE YEAR(OrderDate) = m.nYear ;
         GROUP BY 1, 2) csrEmpSalesByYear ;
     ON Employees.EmployeeID = ;
        csrEmpSalesByYear.EmployeeID ;
     ORDER BY OrderYear, TotalSales DESC

ENDSCAN

USE IN csrYears
SELECT csrTopEmployeeSalesByYear
```

SQL-only

The alternative VFP solution uses only SQL commands, but relies on a trick of sorts. Like the mixed solution, it starts with a query to collect the basic data needed. It then joins that data to itself in a way that results in multiple records for each employee/year combination and uses HAVING to keep only those that represent the top n records. Finally, it adds the employee name. Listing 4 (TopNEmployeeSalesByYear-Trick.prg in this month's downloads) shows the code.

Listing 4. This solution uses only SQL, but requires a tricky join condition.

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

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

The first query here is just a variant of Listing 1. The key portion of this approach is the derived table in the second query, in particular, the join condition between the two instances of csrEmpSalesByYear, shown in Listing 5. Records are matched up first by having the same year and then by having sales in the first instance be the same or more than sales in the second instance. This results in a single record for the employee from that year with the highest sales total, two records for the employee with the second highest sales total and so on.

Listing 5. The key to this solution is the unorthodox join condition between two instances of the same table.

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

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

To make more sense of this solution, first consider the query in Listing 6 (included in this month's downloads as TopNEmployeeSalesByYearBeforeGrouping.prg). It assumes we've already run the query to create the EmpSalesByYear cursor. It shows the results from the derived table in **Listing 4** before the GROUP BY is applies. In the partial results shown in Figure 3 you can see one record for employee 9 in 1996, two for employee 6, three for employee 7 and so forth. (If this still doesn't make sense, try adding the field ESBY2.TotalSales to the query, so you can see that each row represents an employee with the same or lower total sales as the one you're looking at.)

Listing 6. This query demonstrates the intermediate results for the derived table in **Listing 4**.

```
SELECT ESBY1.EmployeeID, ;
       ESBY1.OrderYear, ;
       ESBY1.TotalSales ;
  FROM EmpSalesByYear ESBY1 ;
    JOIN EmpSalesByYear ESBY2 ;
      ON ESBY1.OrderYear = ESBY2.OrderYear ;
     AND ESBY1.TotalSales >= ;
         ESBY2.TotalSales ;
  ORDER BY ESBY1.OrderYear, ;
           ESBY1.TotalSales ;
  INTO CURSOR csrIntermediate
```

Employeeid	Orderyear	Totalsales
9	1996	11365.7000
6	1996	17731.1000
6	1996	17731.1000
7	1996	18104.8000
7	1996	18104.8000
7	1996	18104.8000
3	1996	19231.8000
3	1996	19231.8000
3	1996	19231.8000
3	1996	19231.8000
5	1996	21965.2000

Figure 3. The query in **Listing 6** unfolds the data that's grouped in the derived table.

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

The SQL Server solution

Solving the top n by group problem in SQL Server uses a couple of CTEs (computed table expressions, explained in my last article), but also uses another construct that's not available in VFP's version of SQL.

The OVER clause lets you apply a function to all or part of a result set; it's used in the field list. There are several variations, but the basic structure is shown in Listing 7.

Listing 7. The OVER clause lets you apply a function to all or some of the records in a query.

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

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

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

For example, Listing 8 (included in this month's downloads as EmployeeOrderNumber.sql) shows a query that lists AdventureWorks (2008) employees in the order they were hired, giving each an "employee order number." Here, the data is ordered by HireDate and then ROW_NUMBER() applied to provide the rank of each record. Figure 4 shows partial results.

Listing 8. Using ROW_NUMBER() with OVER lets you give records a rank.

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

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

The ROW_NUMBER() funtion doesn't know anything about ties. However, the RANK() function is aware of ties and assigns them the same value, then skips the appropriate number of values. Listing 9 (EmployeeRank.SQL in this month's downloads) shows the same query using RANK() instead of ROW_NUMBER(); Figure 5 shows the first few records. This time, you can see that Ellerbock and Erickson have the same rank, 8, while Barry Johnson, who immediately follows them, still has a rank of 10.

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

Figure 4. The query in **Listing 8** applies a rank to each employee by hire date.

Listing 9. The RANK() function is aware of ties, assigning them the same value.

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

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

Figure 5. Using RANK() assigns the same EmployeeOrderNumber to records with the same hire date.

You can't say that either ROW_NUMBER() or RANK() is right; which one you want depends on the situation. In fact, there's a third related function, DENSE_RANK() that behaves like RANK(), giving ties the same value, but then continues numbering

in order. That is, if we used DENSE_RANK() in this example, Barry Johnson would have a rank of 9, rather than 10.

Partitioning with OVER

In addition to specifying ordering, OVER also allows us to divide the data into groups before applying the function, using the PARTITION BY clause. The query in Listing 10 (included in this month's downloads as EmployeeRankByDept.sql) assigns employee ranks within each department rather than for the company as a whole by using both PARTITION BY and ORDER BY. Figure 6 shows partial results; note that the numbering begins again for each department and the handling of ties.

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

Figure 6. Here, employees are numbered within their current department, based on when they started in that department.

Listing 10. Combining PARTITION BY and ORDER BY in the OVER clause lets you apply ranks within a group.

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

This example should provide a hint as to how we'll solve the top n by group problem, since we now have a way to number things by group. All we need to do is filter so we only keep those whose rank within the group is in the range of interest. However, it's not possible to filter on the computed field EmployeeOrderNumber in the same query. Instead, we turn that query into a CTE and filter in the main query, as in Listing 11 (LongestStandingEmployeesByDept.sql in this month's downloads).

Listing 11. Once we have the rank for an item within its group, we just need to filter to get the top n items by group.

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

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

Figure 7 shows part of the result. Note that there are many more than three records shown for the Sales department because a whole group of people started on the same day. If you really want only three per department and don't care which records you omit from a last-place tie, use RECORD_NUMBER() instead of RANK().

Applying the same principle to finding the top five salespeople by year at AdventureWorks (to match our VFP example) is a little more complicated because we have to compute sales totals first. To make that work, we first use a CTE to compute those totals and then a second CTE based on that result to add the ranks. Listing 12 (TopSalesPeopleByYear.sql in this month's downloads) shows the complete query.

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

Figure 7. The query in Listing 11 provides the three longest-standing employees in each department. When there are ties, it may produce more than three results.

Listing 12. Finding the top five salepeople by year requires cascading CTEs, plus the OVER clause.

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

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

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

The first CTE, TotalSalesBySalesPerson, contains the ID for the salesperson, the year and that person's total sales for the year. The second CTE, RankSalesPerson, adds rank within the group to the data from TotalSalesByPerson. Finally, the main query keeps only the top five in each and adds the actual name of the person. Figure 8 shows partial results.

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

Final thoughts

The OVER clause has other uses, such as helping to de-dupe a list. In SQL 2012, it's even more useful, with the ability to apply the function to a group of records based not only on an expression, but based on position within a group. Future articles in this series may present some of these techniques.

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

Figure 8. These partial results show the top five salespeople by year.

Author Profile

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