Using OVER with analytic functions, Part 1

With SQL Server 2012 and later, you can pull data from multiple records into a single result.

Tamar E. Granor, Ph.D.

In my last article, I looked at the OVER keyword with aggregate functions and showed how to do things like compute running totals and moving averages. In this article and the next, we'll look at the analytical functions that work with OVER; they provide ways to do side-by-side comparisons, compute percentiles and more.

First, a quick review. The OVER keyword lets you apply a function to a group of records. You can divide the records addressed by the query into groups (partitions) and indicate the order in which the function should be applied. In the May, 2014 issue, I showed how OVER lets you find the top N records in a group, and more broadly, how it lets you number records within a group. In the last issue, I looked at OVER with the aggregate functions like SUM and AVERAGE, allowing you aggregate different groups of records in a single query.

PARTITION BY sets up the groups to which the function is applied. ORDER BY determines the order in which the function is applied.

The final group of functions that can be used with OVER are analytical functions; they were added in SQL SERVER 2012. They can be broken into two broad subsets. The first gives you access to values from other records in the group: FIRST_VALUE and LAST_VALUE, as their names suggest, let you grab values from the first or last record in a partition; LEAD and LAG, provide access to records following or preceding the current record.

The second subset looks at percentiles and distributions: CUME_DIST, PERCENTILE_CONT, PERCENTILE_DISC, and PERCENT_RANK.

We'll look at the first group in this article, and the second group in the next.

Comparing across records

The functions that give you access to different records in the partition allow you to put data from multiple records into a single result record without doing a self-join. Let's start with LEAD and LAG, which are the easiest to understand.

Suppose you'd like to see the number of units sold for each product by year, and include the prior year's sales and the next year's sales in the same row. That is, you want each record to show three years' worth of sales for a single product.

In VFP, you need to use three copies of the table (or cursor) that contains the totals to do this, as in Listing 1 (included in this month's downloads as ThreeYearProductSales.PRG). The first query computes the yearly totals for each product, and puts them into a cursor called csrYearlySales. Then, the second query joins three instances of csrYearlySales, matching records on ProductID and then looking one year back and one year forward, respectively, in the second part of each join condition. As the partial results in Figure 1 show, you get the null value for the previous year in the first record for each product and for the following year in the last record for each product. (Since the Northwind database has data for only three years, you get exactly three rows per product here, but if there were data covering a longer span of years, there'd be more rows for each product.)

Listing 1. To include data from multiple records in the same table into a single record in the result in VFP, you have to use a self-join, including the source table once for each record you want to access.

```
SELECT YEAR(OrderDate) AS OrderYear,
       ProductID, SUM(Quantity) AS NumSold ;
  FROM Orders ;
    JOIN OrderDetails ;
      ON Orders.OrderID = ;
         OrderDetails.OrderID ;
  GROUP BY 1, 2 ;
  ORDER BY 2, 1 ;
  INTO CURSOR csrYearlySales

SELECT Curr.ProductID, Curr.OrderYear, ;
       Prev.NumSold AS PrevYear, ;
       Curr.NumSold AS CurrYear, ;
       Foll.NumSold AS FollYear ;
  FROM csrYearlySales Curr ;
    LEFT JOIN csrYearlySales Prev ;
      ON Curr.ProductID = Prev.ProductID ;
      AND Curr.OrderYear = Prev.OrderYear + 1;
    LEFT JOIN csrYearlySales Foll ;
      ON Curr.ProductID = Foll.ProductID ;
      AND Curr.OrderYear = Foll.OrderYear - 1;
  ORDER BY 1, 2 ;
  INTO CURSOR csrThreeYears
```

Productid	Orderyear	Prevyear	Curryear	Follyear
1	1996	.NULL.	125	304
1	1997	125	304	399
1	1998	304	399	.NULL.
2	1996	.NULL.	226	435
2	1997	226	435	396
2	1998	435	396	.NULL.
3	1996	.NULL.	30	190
3	1997	30	190	108
3	1998	190	108	.NULL.
4	1996	.NULL.	107	264
4	1997	107	264	82
4	1998	264	82	.NULL.
5	1996	.NULL.	129	19
5	1997	129	19	150
5	1998	19	150	.NULL.
6	1996	.NULL.	36	100
6	1997	36	100	165
6	1998	100	165	.NULL.

Figure 1. To get totals for three different years into the same row of the result, you join three instances of the table that contains the data.

You can solve the problem the same way in T-SQL (though you'd probably use a CTE rather than a separate query to compute the yearly totals). But the LAG and LEAD functions provide a better, more flexible solution.

In its simplest form, LEAD lets you include data from the next record in the partition into the results for the current record. Similarly, the simplest form of LAG pulls data from the preceding record into the result for the current record. For example, the query in Listing 2 (SalesByYearWithPrevAndFoll.sql in this month's downloads) shows the total number sold for each product by year, and includes the number sold for the preceding year and the following year. The CTE computes the total for each product for each year, and then the main query pulls the total for the preceding record (LAG), the current record, and the following record (LEAD). LAG and LEAD are both partitioned by ProductID, so we look only at records for the same product. Figure 2 shows partial results; note that, just as in the VFP version, the PrevYear column is null for the first record for each product, and the FollYear column is null for the last record for each product.

Listing 2. LEAD and LAG let you pull data from other records in the partition into the results for a record.

```
WITH csrYearlySales
     (OrderYear, ProductID, NumSold)
AS
(SELECT YEAR(OrderDate) AS OrderYear,
        ProductID,
        SUM(OrderQty) AS NumSold
   FROM Sales.SalesOrderHeader SOH
     JOIN Sales.SalesOrderDetail SOD
       ON SOH.SalesOrderID = SOD.SalesOrderID
   GROUP BY YEAR(OrderDate), ProductID)

SELECT OrderYear,
       ProductID,
       LAG(NumSold) OVER
         (PARTITION BY ProductID
          ORDER BY OrderYear) AS PrevYear,
       NumSold AS CurrYear,
       LEAD(NumSold) OVER
         (PARTITION BY ProductID
          ORDER BY OrderYear) AS FollYear
   FROM csrYearlySales
   ORDER BY ProductID, OrderYear
```

OrderYear	ProductID	PrevYear	CurrYear	FollYear
2011	707	NULL	331	1278
2012	707	331	1278	2940
2013	707	1278	2940	1717
2014	707	2940	1717	NULL
2011	708	NULL	341	1387
2012	708	341	1387	3088
2013	708	1387	3088	1716
2014	708	3088	1716	NULL
2011	709	NULL	608	499
2012	709	608	499	NULL
2011	710	NULL	66	24
2012	710	66	24	NULL
2011	711	NULL	360	1519
2012	711	360	1519	3088

Figure 2. With LAG and LEAD, you can include data from other records in the same partition.

You can actually pass an expression to LAG and LEAD, not just a single field name. In addition, the two functions have two optional parameters. The second parameter, called offset in the documentation, lets you specify which record to use. It's an offset from the current position, and defaults to 1. So when you omit the parameter, you get the record immediately preceding or immediately following the current record. But you can jump two back or six forward, or whatever. The query in Listing 3 (included in this month's downloads as FiveYearProductSales.sql) shows five years' worth of totals for each product in each record, putting the year the record represents in the middle. As the partial result in Figure 3 shows, we don't actually have five years' sales data, so every record contains some nulls.

Listing 3. You can specify records more than one record away from the current record using the optional second parameter to LAG and LEAD.

```
WITH csrYearlySales
     (OrderYear, ProductID, NumSold)
AS
(SELECT YEAR(OrderDate) AS OrderYear,
        ProductID,
        SUM(OrderQty) AS NumSold
   FROM Sales.SalesOrderHeader SOH
     JOIN Sales.SalesOrderDetail SOD
       ON soh.SalesOrderID = sod.SalesOrderID
   GROUP BY YEAR(OrderDate), ProductID)

SELECT OrderYear,
       ProductID,
       LAG(NumSold, 2) OVER
         (PARTITION BY ProductID
          ORDER BY OrderYear) AS Year1,
       LAG(NumSold) OVER
         (PARTITION BY ProductID
          ORDER BY OrderYear) AS Year2,
       NumSold AS Year3,
       LEAD(NumSold) OVER
```

```
        (PARTITION BY ProductID
         ORDER BY OrderYear) AS Year4,
      LEAD(NumSold,2) OVER
        (PARTITION BY ProductID
         ORDER BY OrderYear) AS Year5
FROM csrYearlySales
ORDER BY ProductID, OrderYear
```

OrderYear	ProductID	Year1	Year2	Year3	Year4	Year5
2011	707	NULL	NULL	331	1278	2940
2012	707	NULL	331	1278	2940	1717
2013	707	331	1278	2940	1717	NULL
2014	707	1278	2940	1717	NULL	NULL
2011	708	NULL	NULL	341	1387	3088
2012	708	NULL	341	1387	3088	1716
2013	708	341	1387	3088	1716	NULL
2014	708	1387	3088	1716	NULL	NULL
2011	709	NULL	NULL	608	499	NULL
2012	709	NULL	608	499	NULL	NULL
2011	710	NULL	NULL	66	24	NULL
2012	710	NULL	66	24	NULL	NULL
2011	711	NULL	NULL	360	1519	3088
2012	711	NULL	360	1519	3088	1776

Figure 3. Using the Offset parameter of LEAD and LAG, you can reach forward and back an arbitrary number of records.

While you can get analogous results in VFP, you'd have to use two more self-joins to csrYearlySales with the appropriate join conditions.

The third parameter to LAG and LEAD lets you specify a default value to use when the computed value is null. For example, if you'd prefer to see zeroes rather than nulls where there's no data, you can specify a third parameter of 0 for each LAG and LEAD in Listing 3.

Looking at first and last records

The second pair of functions that give you access to other records in the same partition is FIRST_VALUE and LAST_VALUE. Though they sound like they'd be exact analogues of each other, they're not. FIRST_VALUE is simpler, so we'll look at it first. (Like LAG and LEAD, these functions let you look at multiple records simultaneously without a self-join, but writing such code without these functions is a lot more complex.)

FIRST_VALUE accepts an expression and returns the value of that expression for the first record in the partition, according to the specified order. For example, the query in Listing 4 (PayHistoryWithOrig.sql in this month's downloads) shows each employee's pay history in chronological order. Each record shows one pay rate and the date it took effect, as well as the original pay rate for this employee. We partition the data on BusinessEntityID, which is the primary key for Person. In each partition, records are ordered by the date of the pay change, so the original pay rate appears first. Look at the last three rows in Figure 4 to see an employee with multiple records.

Listing 4. FIRST_VALUE lets you include data from the first record in the partition with each record in the result.

```
SELECT FirstName, LastName,
       Rate, RateChangeDate,
       FIRST_VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID
           ORDER BY RateChangeDate)
          AS OrigRate
  FROM Person.Person
    JOIN [HumanResources].[EmployeePayHistory]
       EPH
      ON Person.BusinessEntityID =
         EPH.BusinessEntityID
   ORDER BY LastName, FirstName,
            RateChangeDate
```

FirstName	LastName	Rate	RateChangeDate	OrigRate
Karen	Berge	10.25	2009-02-09 00:00:00.000	10.25
Andreas	Berglund	10.5769	2009-02-02 00:00:00.000	10.5769
Matthias	Berndt	9.50	2009-01-20 00:00:00.000	9.50
Jo	Berry	9.25	2010-03-07 00:00:00.000	9.25
Jimmy	Bischoff	9.00	2009-02-26 00:00:00.000	9.00
Michael	Blythe	23.0769	2011-05-31 00:00:00.000	23.0769
David	Bradley	24.00	2007-12-20 00:00:00.000	24.00
David	Bradley	28.75	2009-07-15 00:00:00.000	24.00
David	Bradley	37.50	2012-04-30 00:00:00.000	24.00

Figure 4. Here, each employee pay rate record is shown along with the original pay rate for the employee.

While the example in Listing 4 doesn't seem terribly useful, a small extension of the idea does. The query in Listing 5 (PayHistoryWithPctInc.sql in this month's downloads) computes the percentage increase from the original pay rate and includes only those records that represent changes in pay in the result. The CTE here is required in order to be able to use the computed increase in the WHERE clause.

Listing 5. You can use the analytical functions as part of a larger expression. Here, the original rate found by FIRST_VALUE divides the new rate to find the percent increase.

```
WITH csrPayHikes
   (FirstName, LastName, Rate,
    RateChangeDate, OrigRate, Inc)
AS
(SELECT FirstName, LastName,
       Rate, RateChangeDate,
       FIRST_VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID
           ORDER BY RateChangeDate)
          AS OrigRate,
      CAST((1.00 * Rate/FIRST_VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID
           ORDER BY RateChangeDate)-1)
          AS DECIMAL(5,2)) AS Inc
   FROM Person.Person
    JOIN [HumanResources].[EmployeePayHistory]
       EPH
      ON Person.BusinessEntityID =
         EPH.BusinessEntityID)

SELECT *
   FROM csrPayHikes
   WHERE Inc <> 0
   ORDER BY LastName, FirstName,
            RateChangeDate
```

You'd expect LAST_VALUE to behave the same way, except returning the last value in the partition for the specified expression. However,

by default, the function returns the "running last value," that is, the one you're up to. For example, suppose we replace FIRST_VALUE with LAST_VALUE in the query in Listing 4, so we have the query shown in Listing 6 (included in this month's downloads as PayHistoryWithLast.sql). We get results like those shown in Figure 5. The computed value for CurrRate is the same as the Rate column, because LAST_VALUE looks at the partition only up to the current record.

Listing 6. By default, LAST_VALUE returns the last value of the expression up to the row we're on, not the last value in the partition.

```
SELECT FirstName, LastName,
       Rate, RateChangeDate,
       LAST_VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID
           ORDER BY RateChangeDate) AS OrigRate
   FROM Person.Person
    JOIN [HumanResources].[EmployeePayHistory]
         EPH
      ON Person.BusinessEntityID =
         EPH.BusinessEntityID
   ORDER BY LastName, FirstName,
            RateChangeDate
```

FirstName	LastName	Rate	RateChangeDate	CurrRate
Paula	Barreto de ...	27.1394	2008-12-06 00:00:00.000	27.1394
Wanida	Benshoof	13.4615	2011-01-07 00:00:00.000	13.4615
Karen	Berg	27.4038	2009-02-16 00:00:00.000	27.4038
Karen	Berge	10.25	2009-02-09 00:00:00.000	10.25
Andreas	Berglund	10.5769	2009-02-02 00:00:00.000	10.5769
Matthias	Berndt	9.50	2009-01-20 00:00:00.000	9.50
Jo	Berry	9.25	2010-03-07 00:00:00.000	9.25
Jimmy	Bischoff	9.00	2009-02-26 00:00:00.000	9.00
Michael	Blythe	23.0769	2011-05-31 00:00:00.000	23.0769
David	Bradley	24.00	2007-12-20 00:00:00.000	24.00
David	Bradley	28.75	2009-07-15 00:00:00.000	28.75
David	Bradley	37.50	2012-04-30 00:00:00.000	37.50

Figure 5. Because of the default behavior of LAST_VALUE, the CurrRate column here is always the same as the Rate column.

The secret to getting the actual last value in the partition is to use the window notation discussed in my last article. Here, we need only the simplest case: RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. This tells LAST_VALUE to look at the entire partition. The query in Listing 7 (included in this month's downloads as PayHistoryWithOrigAndCurr.sql) shows the pay rate represented by the particular record, the original pay rate and the current pay rate. Figure 6 shows partial results; the last three records in the figure demonstrate the correct results for an employee with multiple pay rates.

Listing 7. Use the RANGE clause with LAST_VALUE to find the last value across the entire partition.

```
SELECT FirstName, LastName,
       Rate, RateChangeDate,
       FIRST_VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID
           ORDER BY RateChangeDate)
          AS OrigRate,
       LAST_VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID
           ORDER BY RateChangeDate
           RANGE BETWEEN UNBOUNDED PRECEDING
           AND UNBOUNDED FOLLOWING)
          AS CurrRate
   FROM Person.Person
    JOIN [HumanResources].[EmployeePayHistory]
         EPH
      ON Person.BusinessEntityID =
         EPH.BusinessEntityID
   ORDER BY LastName, FirstName,
            RateChangeDate
```

FirstName	LastName	Rate	RateChangeDate	OrigR...	CurrRate
Paula	Barreto de ...	27.1394	2008-12-06 00:00:00.000	27.1394	27.1394
Wanida	Benshoof	13.4615	2011-01-07 00:00:00.000	13.4615	13.4615
Karen	Berg	27.4038	2009-02-16 00:00:00.000	27.4038	27.4038
Karen	Berge	10.25	2009-02-09 00:00:00.000	10.25	10.25
Andreas	Berglund	10.5769	2009-02-02 00:00:00.000	10.5769	10.5769
Matthias	Berndt	9.50	2009-01-20 00:00:00.000	9.50	9.50
Jo	Berry	9.25	2010-03-07 00:00:00.000	9.25	9.25
Jimmy	Bischoff	9.00	2009-02-26 00:00:00.000	9.00	9.00
Michael	Blythe	23.0769	2011-05-31 00:00:00.000	23.0769	23.0769
David	Bradley	24.00	2007-12-20 00:00:00.000	24.00	37.50
David	Bradley	28.75	2009-07-15 00:00:00.000	24.00	37.50
David	Bradley	37.50	2012-04-30 00:00:00.000	24.00	37.50

Figure 6. When LAST_VALUE is applied together with RANGE UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING, you get the value from the last record in the partition.

As with FIRST_VALUE, you can use LAST_VALUE as part of a larger expression, so you could compute, say, the percentage increase from the pay rate in the current record to the current pay rate returned by LAST_VALUE.

These two functions also let you work around a limitation of the MIN()and MAX() aggregate functions. For example, you might want to compute the number of units sold for each product in each year and include information about the best and worst years for that product. If all you want to know is the number sold in the best and worst years for each product, you can do that with a simple GROUP BY, as in Listing 8 (MinMaxProductsSold.sql in this month's downloads).

Listing 8. If all you want is to find a minimum or maximum value, you don't need FIRST_VALUE or LAST_VALUE.

```
WITH csrYearlySales
     (OrderYear, ProductID, NumSold)
AS

(SELECT YEAR(OrderDate) AS OrderYear,
        ProductID,
        SUM(OrderQty) AS NumSold
   FROM Sales.SalesOrderHeader SOH
     JOIN Sales.SalesOrderDetail SOD
       ON soh.SalesOrderID = sod.SalesOrderID
   GROUP BY YEAR(OrderDate), ProductID)

SELECT ProductID, MIN(NumSold) AS MinSold,
       MAX(NumSold) AS MaxSold
  FROM csrYearlySales
  GROUP BY ProductID
  ORDER BY ProductID
```

But suppose you want to know which year was best and which was worst. You can't just add OrderYear to the field list; that will give you an error. Specifying MIN(OrderYear) doesn't give you the year for the minimum sold; it gives you the first

year in the partition. But with FIRST_VALUE and LAST_VALUE, you can get exactly what you want, as in Listing 9 (included in this month's downloads as SalesByYearWithWorstAndBest.sql). Figure 7. shows partial results.

Listing 9. FIRST_VALUE and LAST_VALUE solve the problem that MIN and MAX can't give you the values of other fields in the record that produced the minimum or maximum.

```
WITH csrYearlySales
     (OrderYear, ProductID, NumSold)
AS

(SELECT YEAR(OrderDate) AS OrderYear,
        ProductID,
        SUM(OrderQty) AS NumSold
   FROM Sales.SalesOrderHeader SOH
     JOIN Sales.SalesOrderDetail SOD
       ON soh.SalesOrderID = sod.SalesOrderID
   GROUP BY YEAR(OrderDate), ProductID)

SELECT ProductID, OrderYear, NumSold,
       FIRST_VALUE(NumSold) OVER
         (PARTITION BY ProductID
          ORDER BY NumSold) AS MinSold,
       FIRST_VALUE(OrderYear) OVER
         (PARTITION BY ProductID
          ORDER BY NumSold) AS MinYear,
       LAST_VALUE(NumSold) OVER
         (PARTITION BY ProductID
          ORDER BY NumSold
          RANGE BETWEEN UNBOUNDED PRECEDING
          AND UNBOUNDED FOLLOWING) AS MaxSold,
       LAST_VALUE(OrderYear) OVER
         (PARTITION BY ProductID
          ORDER BY NumSold
          RANGE BETWEEN UNBOUNDED PRECEDING
          AND UNBOUNDED FOLLOWING) AS MaxYear
  FROM csrYearlySales
  ORDER BY ProductID, OrderYear
```

More to come

In my next article, I'll look at the NTILE function and the remaining analytic functions, which all address percentiles and distributions.

ProductID	OrderYear	NumSold	MinSold	MinYear	MaxSold	MaxYear
707	2011	331	331	2011	2940	2013
707	2012	1278	331	2011	2940	2013
707	2013	2940	331	2011	2940	2013
707	2014	1717	331	2011	2940	2013
708	2011	341	341	2011	3088	2013
708	2012	1387	341	2011	3088	2013
708	2013	3088	341	2011	3088	2013
708	2014	1716	341	2011	3088	2013
709	2011	608	499	2012	608	2011
709	2012	499	499	2012	608	2011
710	2011	66	24	2012	66	2011
710	2012	24	24	2012	66	2011
711	2011	360	360	2011	3088	2013
711	2012	1519	360	2011	3088	2013
711	2013	3088	360	2011	3088	2013

Figure 7. These results show sales by product by year, along with the worst and best year for that product.

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.*