More on OVER

The OVER keyword lets you compute group totals, running totals and more.

Tamar E. Granor, Ph.D.

In the May, 2014 issue, I showed how the OVER keyword lets you find the top N in a group. This month, we'll look at some other ways to use OVER to simplify calculations.

My previous article showed how to use OVER with the various ranking functions (such as RANK and RECORD_NUMBER) to assign each record in a result set a rank. But OVER can also be used with the aggregate functions: COUNT, SUM, AVG, MIN, MAX and so forth. In addition, in SQL Server 2012 and later, there's also a set of analytic functions that work with OVER. In this article, I'll look at the aggregate functions, plus an additional way to specify which records are used by the function.

A quick review

The OVER keyword lets you order data and apply a function to the records in that order. You can also divide (PARTITION) your data into groups and apply a specified function to the records in each group. For example, the query in Listing 1 numbers employees in the order in which they joined their current department.

Listing 1. OVER lets you put records in order and divide them into groups, then apply a function to each record.

```
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
```

Aggregating with OVER

The aggregate functions are usually used in conjunction with GROUP BY to compute things like total sales for each salesperson each year, or the number of days each student has been absent each semester. At first glance, it would appear that using aggregate functions with OVER would do the same thing, but there are some important differences.

First, using OVER, you can aggregate on different groups within a single query. For example, the query in Listing 2 computes the yearly, monthly and daily number sold for each product; Figure 1 shows a portion of the results when the query is run against the AdventureWorks 2014 example database. The results show the other significant difference between aggregating by GROUP BY and aggregating by OVER. With GROUP BY, you end up with a single record for each group. With OVER, you get whatever records the JOIN and WHERE clauses give you, but they contain aggregated results.

Listing 2. OVER can be combined with the aggregate functions to let you aggregate by different groups in a single query.

```
SELECT Orderdate, ProductID,
       SUM(sod.orderqty) OVER
         (PARTITION BY sod.productID,
          YEAR(orderdate)) AS Yearly,
       SUM(sod.orderqty) OVER
         (PARTITION BY sod.productID,
          YEAR(orderdate), MONTH(orderdate))
          AS Monthly,
       SUM(sod.orderqty) OVER
        (PARTITION BY sod.productID,
         orderdate) AS Daily
  FROM Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
      ON soh.SalesOrderID = sod.SalesOrderID
  ORDER BY ProductID, OrderDate
```

Orderdate	ProductID	Yearly	Monthly	Daily
2014-05-19 00:00:00.000	998	580	127	2
2014-05-19 00:00:00.000	998	580	127	2
2014-05-20 00:00:00.000	998	580	127	1
2014-05-21 00:00:00.000	998	580	127	2
2014-05-21 00:00:00.000	998	580	127	2
2014-05-26 00:00:00.000	998	580	127	1
2014-05-27 00:00:00.000	998	580	127	2
2014-05-27 00:00:00.000	998	580	127	2
2014-05-28 00:00:00.000	998	580	127	1
2014-05-29 00:00:00.000	998	580	127	1
2014-05-30 00:00:00.000	998	580	127	1
2013-05-30 00:00:00.000	999	826	98	97
2013-05-30 00:00:00.000	999	826	98	97
2013-05-30 00:00:00.000	999	826	98	97
2013-05-30 00:00:00.000	999	826	98	97

Figure 1. When you use OVER for aggregation, you get all the records you'd get without it.

In this example, if you want to see just one record for each date, add DISTINCT to the query, as in Listing 3 (included in this month's downloads as SalesByYearMonthDay.sql). Figure 2 shows partial results.

Listing 3. Adding DISTINCT to the query gives us one record per date, but still includes yearly, monthly and daily totals.

```
SELECT DISTINCT Orderdate, ProductID,
       SUM(sod.orderqty) OVER
         (PARTITION BY sod.productID,
          YEAR(orderdate)) AS Yearly,
       SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           YEAR(orderdate), MONTH(orderdate))
           AS Monthly,
       SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           orderdate) AS Daily
  FROM Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
      ON soh.SalesOrderID = sod.SalesOrderID
  ORDER BY ProductID, OrderDate
```

Orderdate	ProductID	Yearly	Monthly	Daily
2014-05-19 00:00:00.000	998	580	127	2
2014-05-20 00:00:00.000	998	580	127	1
2014-05-21 00:00:00.000	998	580	127	2
2014-05-26 00:00:00.000	998	580	127	1
2014-05-27 00:00:00.000	998	580	127	2
2014-05-28 00:00:00.000	998	580	127	1
2014-05-29 00:00:00.000	998	580	127	1
2014-05-30 00:00:00.000	998	580	127	1
2013-05-30 00:00:00.000	999	826	98	97
2013-05-31 00:00:00.000	999	826	98	1
2013-06-02 00:00:00.000	999	826	117	1
2013-06-06 00:00:00.000	999	826	117	1
2013-06-07 00:00:00.000	999	826	117	1
2013-06-09 00:00:00.000	999	826	117	1
2013-06-10 00:00:00.000	999	826	117	1

Figure 2. The query in Listing 3 results in one record per date

Computing percentages

You can use OVER to compute what percent of a total a particular record represents. Listing 4 builds on the previous example to indicate what percent of annual and monthly sales for the product a given day's sales represent. The number sold for the day is divided by the number sold in the month or year; that value is then multiplied by 100 and cast as a decimal to show the percentage. Figure 3 shows partial results. The query is included in this month's downloads as SalesByYearMonthDayWithPcts.sql.

Listing 4. In this query, OVER is used with SUM() to figure out what percent of a product's monthly and yearly sales came on a particular day.

```
SELECT DISTINCT Orderdate, ProductID,
       SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           YEAR(orderdate)) AS Yearly,
       SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           YEAR(orderdate), MONTH(orderdate))
           AS Monthly,
       SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           orderdate) AS Daily,
       CAST(1. * SUM(OrderQty) OVER
          (PARTITION BY sod.productID,
           OrderDate) / SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           YEAR(orderdate)) * 100
           AS decimal(5,2)) AS PctOfYear,
       CAST(1. * SUM(OrderQty) OVER
          (PARTITION BY sod.productID,
           orderdate)/SUM(sod.orderqty) OVER
          (PARTITION BY sod.productID,
           YEAR(orderdate), Month(orderdate))
           * 100 AS decimal(5,2)) AS PctOfMonth
  FROM Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
      ON soh.SalesOrderID = sod.SalesOrderID
  ORDER BY OrderDate, ProductID
```

Orderdate	Produc...	Yea...	Mont...	Daily	PctOfY...	PctOfMo...
2011-05-31 00:00:00.000	707	331	24	24	7.25	100.00
2011-07-01 00:00:00.000	707	331	58	58	17.52	100.00
2011-08-01 00:00:00.000	707	331	96	55	16.62	57.29
2011-08-31 00:00:00.000	707	331	96	41	12.39	42.71
2011-10-01 00:00:00.000	707	331	141	77	23.26	54.61
2011-10-31 00:00:00.000	707	331	141	64	19.34	45.39
2011-12-01 00:00:00.000	707	331	12	12	3.63	100.00
2012-01-01 00:00:00.000	707	1278	61	31	2.43	50.82
2012-01-29 00:00:00.000	707	1278	61	30	2.35	49.18
2012-02-29 00:00:00.000	707	1278	27	27	2.11	100.00
2012-03-30 00:00:00.000	707	1278	93	93	7.28	100.00
2012-04-30 00:00:00.000	707	1278	52	52	4.07	100.00
2012-05-30 00:00:00.000	707	1278	162	162	12.68	100.00
2012-06-30 00:00:00.000	707	1278	214	214	16.74	100.00
2012-07-31 00:00:00.000	707	1278	197	197	15.41	100.00

Figure 3. You can use OVER to compute what percent of a group total a particular value or subset represents. Here, the day's sales are computed as a percentage of the annual and monthly sales for the product.

Running totals and moving averages

Although I'm working with SQL Server 2014, you can use OVER with aggregate functions all the way back to SQL Server 2005. However, until SQL Server 2012, you couldn't include an ORDER clause with OVER and an aggregate function; that use of OVER was restricted to PARTITION.

The ability to include ORDER BY with OVER and aggregate functions lets you compute running totals and what are called *moving averages*. When ORDER BY is included, the specified aggregate is computed for all records in the group up to and including the current record. Listing 5 (included in this month's downloads as RunningSalesByCustomer.sql) demonstrates; it computes daily, monthly and yearly sales by customer and includes running totals for the monthly and yearly sales. Partial results are shown in Figure 4 look at the rows for customer 11007 to see the monthly running total change.

Listing 5. You can add ORDER BY to an OVER clause using an aggregate function to get a running total or moving average.

```
SELECT DISTINCT CustomerID, Orderdate,
       SUM(sod.orderqty*sod.UnitPrice) OVER
          (PARTITION BY CustomerID,
           YEAR(orderdate)) AS Yearly,
       SUM(sod.orderqty*sod.UnitPrice) OVER
          (PARTITION BY CustomerID,
           YEAR(orderdate), MONTH(orderdate))
           AS Monthly,
       SUM(sod.orderqty*sod.UnitPrice) OVER
          (PARTITION BY CustomerID, orderdate)
           AS Daily,
       SUM(sod.orderqty*sod.UnitPrice) OVER
          (PARTITION BY CustomerID,
           YEAR(orderdate) ORDER BY orderdate)
           AS YearlyRunning,
       SUM(sod.orderqty*sod.UnitPrice) OVER
          (PARTITION BY CustomerID,
```

```
        YEAR(orderdate), MONTH(orderdate)
        ORDER BY orderdate)
        AS MonthlyRunning
  FROM Sales.SalesOrderHeader SOH
   JOIN Sales.SalesOrderDetail SOD
     ON soh.SalesOrderID = sod.SalesOrderID
ORDER BY CustomerID, OrderDate
```

```
      SUM(Subtotal) OVER
        (PARTITION BY CustomerID,
          YEAR(orderdate), MONTH(orderdate))
        AS Monthly,
      AVG(Subtotal) OVER
        (PARTITION BY CustomerID,
          YEAR(OrderDate) ORDER BY OrderDate)
        AS RunningAvg
  FROM Sales.SalesOrderHeader
  ORDER BY CustomerID, OrderDate
```

CustomerID	Orderdate	Yearly	Monthly	Daily	YearlyRunning	MonthlyRunning
11005	2011-06-01 00:00:00.000	3374.99	3374.99	3374.99	3374.99	3374.99
11005	2013-06-25 00:00:00.000	4746.34	2362.27	2362.27	2362.27	2362.27
11005	2013-10-02 00:00:00.000	4746.34	2384.07	2384.07	4746.34	2384.07
11006	2011-06-26 00:00:00.000	3399.99	3399.99	3399.99	3399.99	3399.99
11006	2013-05-31 00:00:00.000	4719.04	2334.97	2334.97	2334.97	2334.97
11006	2013-10-14 00:00:00.000	4719.04	2384.07	2384.07	4719.04	2384.07
11007	2011-06-11 00:00:00.000	3399.99	3399.99	3399.99	3399.99	3399.99
11007	2013-06-23 00:00:00.000	4811.01	2391.95	2391.95	2391.95	2391.95
11007	2013-08-19 00:00:00.000	4811.01	2419.06	2419.06	4811.01	2419.06
11008	2011-06-27 00:00:00.000	3374.99	3374.99	3374.99	3374.99	3374.99
11008	2013-06-05 00:00:00.000	4731.32	2312.26	2312.26	2312.26	2312.26
11008	2013-08-02 00:00:00.000	4731.32	2419.06	2419.06	4731.32	2419.06
11009	2011-06-29 00:00:00.000	3374.99	3374.99	3374.99	3374.99	3374.99
11009	2013-06-22 00:00:00.000	4716.34	2297.28	2297.28	2297.28	2297.28
11009	2013-10-09 00:00:00.000	4716.34	2419.06	2419.06	4716.34	2419.06

Figure 4. Include ORDER BY when using OVER with SUM() to get a running total.

Similarly, when you use ORDER BY with MIN() and MAX(), you get the minimum or maximum value in the group to this point. The query in **Listing 7** shows the minimum and maximum quantity in a single order to date for each product. **Figure 6** shows partial results. The query is included in this month's downloads as SalesByYearMonthDayWithMinMax.sql.

Running totals are probably the easiest of this type of calculation to understand, but you can do the same thing with most of the aggregate functions. When you apply ORDER BY to AVG(), you get a moving average, that is, the average of the all the records in the group up to this point. The last record in the group will show the average for the whole group. Listing 6 (included in this month's downloads as SalesWithMovingAverage.sql) demonstrates by computing the moving average of sales for a customer within a year. Figure 5 shows partial results.

CustomerID	OrderDate	Yearly	Monthly	RunningA...
11000	2011-06-21 00:00:00.000	3399.99	3399.99	3399.99
11000	2013-06-20 00:00:00.000	4849.00	2341.97	2341.97
11000	2013-10-03 00:00:00.000	4849.00	2507.03	2424.50
11001	2011-06-17 00:00:00.000	3374.99	3374.99	3374.99
11001	2013-06-18 00:00:00.000	2419.93	2419.93	2419.93
11001	2014-05-12 00:00:00.000	588.96	588.96	588.96
11002	2011-06-09 00:00:00.000	3399.99	3399.99	3399.99
11002	2013-06-02 00:00:00.000	4714.05	2294.99	2294.99
11002	2013-07-26 00:00:00.000	4714.05	2419.06	2357.025
11003	2011-05-31 00:00:00.000	3399.99	3399.99	3399.99
11003	2013-06-07 00:00:00.000	4739.30	2318.96	2318.96
11003	2013-10-10 00:00:00.000	4739.30	2420.34	2369.65

Figure 5. The last column here shows the moving average of sales for a customer within a year. Look at the last record for each customer for the year to see the overall average for the year.

Listing 6. Using AVERAGE with OVER and an ORDER BY clause gives moving averages, the average of the records in the group up to and including the current record.

```
SELECT CustomerID, OrderDate,
       SUM(Subtotal) OVER
         (PARTITION BY CustomerID,
           YEAR(orderdate)) AS Yearly,
```

Listing 7. Applying OVER with ORDER BY to MIN() and MAX() lets you compute the minimum and maximum so far.

```
SELECT DISTINCT Orderdate, ProductID,
       SUM(sod.orderqty) OVER
         (PARTITION BY sod.productID,
          YEAR(orderdate)) AS Yearly,
       SUM(sod.orderqty) OVER
         (PARTITION BY sod.productID,
          YEAR(orderdate), MONTH(orderdate))
          AS Monthly,
       SUM(sod.orderqty) OVER
         (PARTITION BY sod.productID,
          orderdate) AS Daily,
       MIN(OrderQty) OVER
         (PARTITION BY ProductID
          ORDER BY OrderDate) as MinOrder,
       MAX(OrderQty) OVER
         (PARTITION BY ProductID
          ORDER BY OrderDate) as MaxOrder
  FROM Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
      ON soh.SalesOrderID = sod.SalesOrderID
  ORDER BY ProductID, OrderDate
```

Orderdate	Produc...	Yea...	Mont...	Daily	MinOr...	MaxOrd...
2011-05-31 00:00:00.000	709	608	38	38	2	6
2011-07-01 00:00:00.000	709	608	134	134	2	26
2011-08-01 00:00:00.000	709	608	167	79	2	26
2011-08-31 00:00:00.000	709	608	167	88	2	26
2011-10-01 00:00:00.000	709	608	224	123	2	26
2011-10-31 00:00:00.000	709	608	224	101	2	26
2011-12-01 00:00:00.000	709	608	45	45	2	26
2012-01-01 00:00:00.000	709	499	181	102	1	26
2012-01-29 00:00:00.000	709	499	181	79	1	26
2012-02-29 00:00:00.000	709	499	78	78	1	26
2012-03-30 00:00:00.000	709	499	135	135	1	26
2012-04-30 00:00:00.000	709	499	105	105	1	32
2011-05-31 00:00:00.000	710	66	5	5	1	3
2011-07-01 00:00:00.000	710	66	13	13	1	4
2011-08-01 00:00:00.000	710	66	19	6	1	4

Figure 6. The last two columns show running minimums and maximums for the quantity of a product in an individual order.

Aggregating subsets within partitions

SQL Server 2012 also introduced another way of narrowing down which records are aggregated. The ROWS and RANGE clauses let you specify that a calculation is applied only to some records within a partition. Let's look at an example first.

Suppose you want to compute yearly orders for each product as well as a two-year moving total. That is, each record in the result should show you sales in a given year for a product, plus the sales for that product across the year you're looking at and the prior year. Your initial reaction may be that you'd need a loop of some sort to compute the two–year (or three-year or five-year totals) after getting yearly totals, but OVER with the ROWS clause makes this fairly easy. Listing 8 (SalesByYear-WithTwoYearTotal.sql in this month's downloads) shows the query; Figure 7 shows partial results. The query uses a CTE to compute the number of items sold each year for each product. Then, the ROWS clause in the fourth field in the main query indicates that the field TwoYear should be computed as the sum of NumSold for the current record and the preceding record within the partition. Note that for the first row of each product, Yearly and TwoYear are the same.

OrderYear	ProductID	Yearly	TwoYear
2011	707	331	331
2012	707	1278	1609
2013	707	2940	4218
2014	707	1717	4657
2011	708	341	341
2012	708	1387	1728
2013	708	3088	4475
2014	708	1716	4804
2011	709	608	608
2012	709	499	1107
2011	710	66	66
2012	710	24	90
2011	711	360	360
2012	711	1519	1879

Figure 7. The TwoYear column here is computed using the ROWS clause.

Listing 8. The ROWS clause lets you apply a function to a subset of a partition.

```
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,
       NumSold AS Yearly,
       SUM(NumSold) OVER (
         PARTITION BY productID
         ORDER BY OrderYear
         ROWS BETWEEN 1 PRECEDING
              AND CURRENT ROW) AS TwoYear
   FROM csrYearlySales
   ORDER BY ProductID, OrderYear
```

As the example demonstrates, the ROWS clause lets you specify a number of rows near the current row. In addition to the PRECEDING and CURRENT ROW items shown, you can also specify FOLLOWING. For example, to have three-year totals including the year before and the year after the current year, you'd specify ROW BETWEEN 1 PRECEDING and 1 FOLLOWING.

The documentation refers to the group of rows as a "window." You can specify UNBOUNDED PRECEDING as the start point to indicate that the window begins with the first row of the partition, or UNBOUNDED FOLLOWING as the end point to say that the window ends with the last row of the partition. Also, note that you can specify a window where all the rows in the window are before the current row or all the rows are after the current row. That is, either PRECEDING or FOLLOWING can be used for either of the start and end points of the window. For example, in the product orders query, you might specify ROW BETWEEN 1 FOLLOWING and 2 FOLLOWING to compute a total (or an average) for the next two years, not including the current year.

The RANGE keyword lets you specify rows based on value rather than position. You can't specify a number at either end with range. You can start with CURRENT ROW or UNBOUNDED PRECEDING and end with CURRENT ROW or UNBOUNDED FOLLOWING.

You can also specify just CURRENT ROW, which says to apply the function to all records in the partition that have the same ORDER BY value as the current record. This offers a way to compute an aggregate while still looking at individual records, as in Listing 9, where we list each order, but include the daily sales total for the salesperson. Partial results are shown in Figure 8. The query is included in this month's downloads as SalesWithDailyTotal.sql.

Orderdate	SalesPerso...	SubTotal	SPDayTotal
2011-10-01 00:00:00.000	274	4194.589	6341.551
2011-10-01 00:00:00.000	274	2146.962	6341.551
2012-01-01 00:00:00.000	274	61206.4782	61206.4782
2012-01-29 00:00:00.000	274	6101.382	18307.746
2012-01-29 00:00:00.000	274	12206.364	18307.746
2012-02-29 00:00:00.000	274	33406.7043	33406.7043
2012-04-30 00:00:00.000	274	40708.4413	44670.6854
2012-04-30 00:00:00.000	274	3962.2441	44670.6854
2012-05-30 00:00:00.000	274	2927.7262	3575.7202
2012-05-30 00:00:00.000	274	647.994	3575.7202
2012-06-30 00:00:00.000	274	2458.9178	55616.5989
2012-06-30 00:00:00.000	274	4254.45	55616.5989
2012-06-30 00:00:00.000	274	48693.9751	55616.5989
2012-06-30 00:00:00.000	274	209.256	55616.5989
2012-07-31 00:00:00.000	274	53.994	523.788

Figure 8. The last column here shows the daily total for the salesperson, using RANGE CURRENT ROW.

Listing 9. This query uses RANGE CURRENT ROW to compute the daily total for each order's salesperson.

```
SELECT Orderdate, SalesPersonID, SubTotal,
       SUM(SubTotal) OVER
```

```
        (PARTITION BY SalesPersonID
         ORDER BY OrderDate
         RANGE CURRENT ROW) AS SPDayTotal
FROM Sales.SalesOrderHeader SOH
WHERE SalesPersonID IS NOT NULL
ORDER BY SalesPersonID, OrderDate
```

RANGE doesn't let you narrow down to specific values, so you can't ask for a function to be applied to, say, all records with the same value as this row and the one immediately following, or with the same value as this row and the next possible value. To do calculations like that, you have to figure out clever ways to partition and order your data.

You can, though, ask for the function to apply to records from this row's value to the end, giving you a "reverse running total." The query in Listing 10 (included in this month's downloads as SalesWithReverseRunningTotalByDay.sql) computes such a reverse running total of sales for the salesperson. Figure 9 shows partial results. Note that it's still a daily computation because RANGE uses the value of the ORDER BY expression to choose records; for example, the first two rows shown have the same value because they're for the same day.

Listing 10. The RANGE specified for the last column produces a reverse running total, where the first row for each salesperson contains the total for that salesperson, and each subsequent row shows that total only from that date to the end.

```
SELECT Orderdate, SalesPersonID, SubTotal,
       SUM(SubTotal) OVER
         (PARTITION BY SalesPersonID
          ORDER BY OrderDate
          RANGE BETWEEN CURRENT ROW AND
          UNBOUNDED FOLLOWING)
        AS ReverseRunningTotal
   FROM Sales.SalesOrderHeader SOH
   WHERE SalesPersonID IS NOT NULL
   ORDER BY SalesPersonID, OrderDate
```

Orderdate	SalesPerso...	SubTotal	ReverseRunningT...
2011-10-01 00:00:00.000	274	4194.589	1069539.1606
2011-10-01 00:00:00.000	274	2146.962	1069539.1606
2012-01-01 00:00:00.000	274	61206.4782	1063197.6096
2012-01-29 00:00:00.000	274	6101.382	1001991.1314
2012-01-29 00:00:00.000	274	12206.364	1001991.1314
2012-02-29 00:00:00.000	274	33406.7043	983683.3854
2012-04-30 00:00:00.000	274	3962.2441	950276.6811
2012-04-30 00:00:00.000	274	40708.4413	950276.6811
2012-05-30 00:00:00.000	274	2927.7262	905605.9957
2012-05-30 00:00:00.000	274	647.994	905605.9957
2012-06-30 00:00:00.000	274	4254.45	902030.2755
2012-06-30 00:00:00.000	274	48693.9751	902030.2755
2012-06-30 00:00:00.000	274	209.256	902030.2755
2012-06-30 00:00:00.000	274	2458.9178	902030.2755
2012-07-31 00:00:00.000	274	469.794	846413.6766

Figure 9. The reverse running total here declines each day for the salesperson.

To compute a reverse running total that declines with each record rather than each day, change RANGE to ROWS, as in Listing 11 (included in this month's downloads as SalesWithReverseRunningTotal.sql). Partial results are shown in Figure 10.

Listing 11. Using ROWS rather than RANGE results in a complete reverse running total that declines with each record.

```
SELECT Orderdate, SalesPersonID, SubTotal,
       SUM(SubTotal) OVER
         (PARTITION BY SalesPersonID
          ORDER BY OrderDate
          ROWS BETWEEN CURRENT ROW AND
          UNBOUNDED FOLLOWING)
         AS ReverseRunningTotal
   FROM Sales.SalesOrderHeader SOH
   WHERE SalesPersonID IS NOT NULL
   ORDER BY SalesPersonID, OrderDate
```

Orderdate	SalesPerso...	SubTotal	ReverseRunningT...
2011-10-01 00:00:00.000	274	4194.589	1067392.1986
2011-10-01 00:00:00.000	274	2146.962	1069539.1606
2012-01-01 00:00:00.000	274	61206.4782	1063197.6096
2012-01-29 00:00:00.000	274	6101.382	989784.7674
2012-01-29 00:00:00.000	274	12206.364	1001991.1314
2012-02-29 00:00:00.000	274	33406.7043	983683.3854
2012-04-30 00:00:00.000	274	3962.2441	909568.2398
2012-04-30 00:00:00.000	274	40708.4413	950276.6811
2012-05-30 00:00:00.000	274	2927.7262	904958.0017
2012-05-30 00:00:00.000	274	647.994	905605.9957
2012-06-30 00:00:00.000	274	4254.45	850668.1266
2012-06-30 00:00:00.000	274	48693.9751	899362.1017
2012-06-30 00:00:00.000	274	209.256	899571.3577
2012-06-30 00:00:00.000	274	2458.9178	902030.2755
2012-07-31 00:00:00.000	274	469.794	846359.6826

Figure 10. Use ROWS BETWEEN CURRENT ROW and UNBOUNDED FOLLOWING to compute reverse running totals.

While these examples of ROWS and RANGE use the SUM() function, they actually can be applied to any of the functions you can use with OVER, so you can find, for example, the largest sale or the average sale for each salesperson on a daily basis. (You might then use that to compute the ratio of a given sale to the largest or average sale for that day.)

What's Next?

In my next article, I'll look at the analytic functions added to OVER in SQL SERVER 2012. These functions let you find the first and last values in a partition, let you access the preceding and following records in the partition, and let you explore the distribution of values.

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.*