Using OVER with analytic functions, Part 2

The final group of T-SQL functions that work with OVER compute and analyze percentiles and distributions.

Tamar E. Granor, Ph.D.

In my last article, I showed how you can use OVER with the LAG, LEAD, FIRST_VALUE and LAST_VALUE functions to put data from different records in a partition into a single result record. This article explores the last set of functions that work with SQL Server's OVER clause; these involve percentiles and distributions.

Dividing records into n-tiles

My discussion of OVER started (in the May, 2014 issue) with using it to assign ranks to records. The ROW_NUMBER, RANK and DENSE_RANK functions discussed there assign an integer to each result record representing its position in the set. Each of the three functions handles ties differently.

A fourth function in that group, NTILE, divides the records up as evenly as possible into a specified number of groups. The function takes a single parameter that indicates the number of groups to create. For example, the query in Listing 1 (SalesQuartiles.sql in this month's downloads) computes the total sales for each salesperson by year, and then divides each year's sales into four groups (quartiles) from lowest to highest. Figure 1 shows partial results; as you can see, when the number of records in the partition can't be divided evenly into the specified number of groups, earlier groups get an extra record.

Listing 1. The NTILE function divides each partition into a specified number of groups.

```
WITH csrAnnualSales
(SalesPersonID, OrderYear, TotalSales)

AS

(SELECT SalesPersonID, YEAR(OrderDate),
        SUM(SubTotal) AS TotalSales
   FROM [Sales].[SalesOrderHeader]
   WHERE SalesPersonID IS NOT NULL
   GROUP BY SalesPersonID, YEAR(OrderDate))

SELECT SalesPersonID, OrderYear, TotalSales,
       NTILE(4) OVER
         (PARTITION BY OrderYear
          ORDER BY TotalSales) AS Quartile
   FROM csrAnnualSales
```

SalesPersonID	OrderYear	TotalSales	Quartile
274	2011	28926.2465	1
278	2011	500091.8202	1
283	2011	599987.9444	1
280	2011	648485.5862	2
275	2011	875823.8318	2
281	2011	967597.2899	2
276	2011	1149715.3253	3
282	2011	1175007.4753	3
277	2011	1311627.2918	4
279	2011	1521289.1881	4
287	2012	116029.652	1
284	2012	441639.5961	1
274	2012	453524.5233	1
290	2012	996291.908	1
280	2012	1208264.3834	2

Figure 1. NTILE makes the groups as even as possible. Here, there are 10 records for 2011, so groups 1 and 2 have 3 records each, while groups 3 and 4 have 2 apiece.

If you change the parameter to NTILE() to 5 (as in Listing 2), you get quintiles instead of quartiles, as in Figure 2.

SalesPersonID	OrderYear	TotalSales	Quintile
274	2011	28926.2465	1
278	2011	500091.8202	1
283	2011	599987.9444	2
280	2011	648485.5862	2
275	2011	875823.8318	3
281	2011	967597.2899	3
276	2011	1149715.3253	4
282	2011	1175007.4753	4
277	2011	1311627.2918	5
279	2011	1521289.1881	5
287	2012	116029.652	1
284	2012	441639.5961	1
274	2012	453524.5233	1
290	2012	996291.908	2
280	2012	1208264.3834	2

Figure 2. Here, 5 was passed to NTILE(), so there are five groups for each year. As before, the group sizes are as even as possible.

Listing 2. The parameter to NTILE() determines how many groups the records in each partition are divided into.

```
NTILE(5) OVER
  (PARTITION BY OrderYear
   ORDER BY TotalSales) AS Quintile
```

Like the other ranking functions, NTILE dates back to SQL Server 2005.

Showing distribution of records

The analytical function group, added in SQL Server 2012, offers ways to rank the records relatively. The CUME_DIST() and PERCENT_RANK() functions both assign each record a value between 0 and 1 representing its position in the partition based on the specified order for the partition. The two functions differ in whether any record is assigned 0; that difference in the first record of the partition leads to different results throughout.

The easiest way to understand the difference between these functions, and between these two and the RANK function described in my May, 2014 article, is to look at the results. The query in Listing 3 (RankAndDistribution.sql in this month's downloads) computes sales by salesperson by year, and then applies a series of analytics to the data. Partial results are shown in Figure 3.

Listing 3. T-SQL offers several ways to show the distribution of data.

```
WITH csrAnnualSales
     (SalesPersonID, OrderYear, TotalSales)
AS
(SELECT SalesPersonID, YEAR(OrderDate),
 SUM(SubTotal) AS TotalSales
  FROM [Sales].[SalesOrderHeader]
  WHERE SalesPersonID IS NOT NULL
  GROUP BY SalesPersonID, YEAR(OrderDate))

SELECT SalesPersonID, OrderYear, TotalSales,
       CUME_DIST() OVER
         (PARTITION BY OrderYear
          ORDER BY TotalSales) AS CumeDist,
       PERCENT_RANK() OVER
         (PARTITION BY OrderYear
          ORDER BY TotalSales) AS PctRank,
       RANK() OVER
         (PARTITION BY OrderYear
          ORDER BY TotalSales) AS Rank,
       COUNT(SalesPersonID) OVER
         (PARTITION BY OrderYear
          ORDER BY TotalSales
          RANGE BETWEEN UNBOUNDED PRECEDING
          AND UNBOUNDED FOLLOWING)
        AS GroupCount,
       CAST(1.00 * RANK() OVER
         (PARTITION BY OrderYear
          ORDER BY TotalSales) /
         COUNT(SalesPersonID) OVER
            (PARTITION BY OrderYear
             ORDER BY TotalSales
             RANGE BETWEEN UNBOUNDED PRECEDING
             AND UNBOUNDED FOLLOWING)
        AS decimal(5,2)) AS ComputedDist
FROM csrAnnualSales
```

Consider the results for 2011. There are 10 records, each with a different value for TotalSales. CUME_DIST divides them into ten evenly-spaced groups. PERCENT_RANK does the same, but the first record always has a rank of 0. The query also demonstrates that you can actually compute CUME_DIST by dividing the RANK of a row by the number of rows in the partition (that is COUNT applied to the same partition).

SalesPersonID	OrderYear	TotalSales	CumeDist	PctRank	Rank	GroupCount	ComputedDist
274	2011	28926.2465	0.1	0	1	10	0.10
278	2011	500091.8202	0.2	0.111111111111111	2	10	0.20
283	2011	599987.9444	0.3	0.222222222222222	3	10	0.30
280	2011	648485.5862	0.4	0.333333333333333	4	10	0.40
275	2011	875823.8318	0.5	0.444444444444444	5	10	0.50
281	2011	967597.2899	0.6	0.555555555555556	6	10	0.60
276	2011	1149715.3253	0.7	0.666666666666667	7	10	0.70
282	2011	1175007.4753	0.8	0.777777777777778	8	10	0.80
277	2011	1311627.2918	0.9	0.888888888888889	9	10	0.90
279	2011	1521289.1881	1	1	10	10	1.00
287	2012	116029.652	0.0714285714285714	0	1	14	0.07
284	2012	441639.5961	0.142857142857143	0.0769230769230769	2	14	0.14
274	2012	453524.5233	0.214285714285714	0.153846153846154	3	14	0.21
290	2012	996291.908	0.285714285714286	0.230769230769231	4	14	0.29
280	2012	1208264.3834	0.357142857142857	0.307692307692308	5	14	0.36

Figure 3. CUME_DIST and PERCENT_RANK give similar but not identical results.

One thing this example doesn't show is what happens when there are ties in the data. That I used RANK (rather than RECORD_NUMBER) when computing the equivalent of CUME_DIST should give you a hint, though. Both CUME_DIST and PERCENT_RANK assign the same result to records with the same sort value. An updated version of a query that appeared in my May, 2014 article demonstrates. The query in Listing 4 (EmployeeRankByDept.sql in this month's downloads) ranks employees in each department by how long they've been working there. As you can

see in the partial results in Figure 4 when multiple employees have the same start date, those employees share the same result both for CUME_DIST and for PERCENT_RANK.

The formula for PERCENT_RANK is much less obvious. It's one less than rank divided by one less than the group size, that is (RANK-1)/(COUNT-1). Subtracting one from the rank ensures that PERCENT_RANK always begins with 0. The SQL Server documentation describes this as the "relative rank of a row within a group of rows."

FirstName	LastName	StartDate	Name	EmployeeRank	CumeDist	PctRank
Ovidiu	Cracium	2010-12-05	Tool Design	3	0.75	0.666666666666667
Janice	Galvin	2010-12-23	Tool Design	4	1	1
Stephen	Jiang	2011-01-04	Sales	1	0.0555555555555556	0
Brian	Welcker	2011-02-15	Sales	2	0.111111111111111	0.0588235294117647
Michael	Blythe	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Linda	Mitchell	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Jillian	Carson	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Garrett	Vargas	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Tsvi	Reiter	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Pamela	Ansman-...	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Shu	Ito	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
José	Saraiva	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
David	Campbell	2011-05-31	Sales	3	0.611111111111111	0.117647058823529
Amy	Alberts	2012-04-16	Sales	12	0.666666666666667	0.647058823529412
Jae	Pak	2012-05-30	Sales	13	0.777777777777778	0.705882352941177

Figure 4. Records with the same value for the ordering expression are assigned the same result by both CUME_DIST and PERCENT_RANK.

Listing 4. Both CUME_DIST and PERCENT_RANK assign the same value to ties.

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

This query also helps to understand exactly what these two functions compute. CUME_DIST is the fraction of records in the partition with the same value as or a lower value than the current record for the ordering expression. So, there are 11 Sales employees who started on 31-May-2011 or earlier; that's divided by 18 (the total number of employees in the Sales department, which you can't tell from this figure). That gives the result 0.61111 shown for all nine employees who started that day.

You can also consider this the percentile into which the record falls. Though I was taught that you never have a 100th percentile, a little research shows that some methods for computing percentile do, in fact, result in a 100th percentile. Note though that, if there is a tie for the greatest value, then no record in that partition has PERCENT_RANK = 1.

You're likely to want to multiply both CUME_DIST and PERCENT_RANK by 100 to get the familiar percentage values we're used to dealing with.

Searching by percentile

The last two analytical functions, PERCENT_CONT and PERCENT_DISC, let you find the cut-off value for a particular percentile. Each accepts a decimal value indicating which percentile is desired; for example, specify .5 to return the median, that is, the value at the 50th percentile, and specify .99 to return the value at the 99th percentile.

The syntax for these functions is a little different than for any of the other functions you can use with OVER. The syntax for PERCENTILE_DISC is shown in Listing 5; the syntax for PERCENTILE_CONT is identical except, of course, for the function name.

Listing 5. The two percentile functions use a different syntax than the other functions that work with OVER.

```
PERCENTILE_DISC( number )
 WITHIN GROUP ( ORDER BY order_by_expression
                [ ASC | DESC ] )
 OVER ( [ PARTITION BY <partition_by_expr> ] )
```

As usual, the PARTITION BY clause lets you break the results up into groups and apply the function separately to each group. While the PARTITION BY clause is optional here, if you want to apply the function to the whole result set as one group, you still have to include the OVER keyword; follow it with empty parentheses.

The WITHIN GROUP clause sets the order used to determine percentiles.

The expression you pass to the function must be a number between 0 and 1. (That's a difference from the other functions that work with OVER.)

The difference between the two functions is in whether they return only values in the data (PERCENTILE_DISC—"DISC" stands for "discrete") or can interpolate between values to give a more accurate answer (PERCENTILE_CONT—"CONT" stands for "continuous").

The query in Listing 6 (TenurePercentile.sql in this month's downloads) shows the number of people in each department, and their average tenure in the department in days (that is, how many days they've been in that department). Then, it computes the 25th, 50th and 75th percentiles for tenure in the department, using each of the two methods. Figure 5 shows partial results.

Listing 6. PERCENTILE_CONT and PERCENTILE_DISC return the value that represents a specified percentile.

```
WITH csrTenure (DepartmentID, DeptName,
                BusinessEntityID, DaysInDept)
AS

(SELECT Department.DepartmentID,
        Department.Name AS DeptName,
        EDH.BusinessEntityID,
        DATEDIFF(DD,StartDate,GETDATE())
  FROM
    HumanResources.EmployeeDepartmentHistory
     EDH
   JOIN HumanResources.Department
     ON EDH.DepartmentID =
        Department.DepartmentID
    WHERE EndDate IS null)

SELECT DISTINCT DeptName,
       COUNT(BusinessEntityID) OVER
         (PARTITION BY DepartmentID)
         AS DeptSize,
       AVG(DaysInDept) OVER
         (PARTITION BY DepartmentID)
         AS AvgTenure,
       PERCENTILE_CONT(.25)
         WITHIN GROUP (ORDER BY DaysInDept)
         OVER (PARTITION BY DepartmentID)
         AS Cont25Pctile,
       PERCENTILE_CONT(.5)
         WITHIN GROUP (ORDER BY DaysInDept)
         OVER (PARTITION BY DepartmentID)
         AS ContMedian,
       PERCENTILE_CONT(.75)
         WITHIN GROUP (ORDER BY DaysInDept)
         OVER (PARTITION BY DepartmentID)
         AS Cont75Pctile,
       PERCENTILE_DISC(.25)
         WITHIN GROUP (ORDER BY DaysInDept)
         OVER (PARTITION BY DepartmentID)
         AS Disc25Pctile,
       PERCENTILE_DISC(.5)
         WITHIN GROUP (ORDER BY DaysInDept)
         OVER (PARTITION BY DepartmentID)
         AS DiscMedian,
       PERCENTILE_DISC(.75)
         WITHIN GROUP (ORDER BY DaysInDept)
         OVER (PARTITION BY DepartmentID)
         AS Disc75Pctile
  FROM csrTenure
 ORDER BY DeptName
```

The most obvious use for these functions is computing medians, but you might use them to build a table of percentiles for a standardized test, as well. I can imagine using them in political discussions about income and taxation, too.

DeptName	DeptSize	AvgTenure	Cont25Pctile	ContMedian	Cont75Pctile	Disc25Pctile	DiscMedian	Disc75Pctile
Document Control	5	2214	2198	2216	2234	2198	2216	2234
Engineering	6	2234	1775	2576.5	2593.5	1509	2573	2598
Executive	2	1341	900.25	1341.5	1782.75	459	459	2224
Facilities and Maintenance	7	1904	1817.5	1846	1892.5	1809	1846	1902
Finance	10	2210	2190.75	2211	2230.5	2189	2208	2232
Human Resources	6	2230	2210	2240.5	2253	2201	2237	2256
Information Services	10	2221	2203.25	2218.5	2241.25	2203	2216	2246
Marketing	9	2033	1501	2177	2226	1501	2177	2226
Production	179	2179	2181	2211	2245	2181	2211	2245
Production Control	6	2061	2180.5	2203	2245.75	2176	2194	2257
Purchasing	12	1777	1737	1848	1873.5	1533	1846	1869
Quality Assurance	6	2154	2185.5	2214	2236.5	2179	2205	2241
Research and Development	4	2194	2178	2210.5	2226.5	2115	2199	2222
Sales	18	1168	992	1357	1357	992	1357	1357
Shipping and Receiving	6	2233	2222.5	2239.5	2257.25	2218	2236	2262

Figure 5. Because PERCENTILE_CONT interpolates, the values it returns may not be in the original data. PERCENTILE_DISC always returns an actual data value.

Story OVER

I've now looked at each group of functions that works with OVER through SQL Server 2014 (though there are a few aggregate functions I haven't demonstrated, in particular, the ones for standard deviation and variance). They provide a wide range of capabilities and make many queries much simpler than they'd otherwise be.

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.*