

Going OVER and Above with SQL

Tamar E. Granor Tomorrow's Solutions, LLC Voice: 215-635-1958 Email: tamar@tomorrowssolutionsllc.com

The SQL 2003 standard introduced the OVER keyword that lets you apply a function to a set of records. Introduced in SQL Server 2005, this capability was extended in SQL Server 2012. The functions allow you to rank records, aggregate them in a variety of ways, put data from multiple records into a single result record, and compute and use percentiles. The set of problems they solve range from removing exact duplicates to computing running totals and moving averages to comparing data from different periods to removing outliers.

In this session, we'll look at the OVER operator and the many functions you can use with it. We'll look at a variety of problems that can be solved using OVER. Over the last couple of years, I've been exploring aspects of SQL Server's T-SQL implementation that aren't included in VFP's SQL sub-language. I first noticed the OVER keyword as an easy way to solve a problem that's fairly complex with VFP's SQL, getting the top N records in each of a set of groups with a single query. At the time, I noticed that OVER had other uses, but I didn't stop to explore them.

When I finally returned to see what else OVER could do, I was blown away. In recent versions of SQL Server (2012 and later), OVER provides ways to compute running totals and moving averages, to put data from several records of the same table into a single result record, to divide records into percentile groups and more.

The more I looked at this capability, the more impressed I became, so I decided that while doing a session on a single aspect of a single command seems odd, there was good reason to do so.

Introduction

The formal name for the set of capabilities provided by the OVER clause is "window functions." They were introduced in the ANSI SQL 2003 standard and extended in the 2008 standard. Support for window functions was introduced in SQL Server 2005 and significantly enhanced in SQL Server 2012.

The basic idea with window functions is that you can define a set of records and apply a function to only that set of records in order to specify a field in a query. There are several ways to specify the set of records and they can be combined. The basic syntax for this is shown in **Listing 1**; it applies to all the functions except PERCENTILE_CONT and PERCENTILE_DISC. (See "Searching by percentile," later in this document, for the syntax for those two functions.) The three optional clauses inside the parentheses provide the definition for the set of records. **Table 1** shows the list of window functions.

Listing 1. Most of the window functions use this syntax.

```
<window function> OVER (
  [PARTITION BY <list of expressions>]
  [ORDER BY <list of <expression> ASC | DESC>>]
  [ROWS | RANGE <window frame>])
```

Table 1. SQL Server supports quite a few window functions. Support has improved over time.

Function	Version introduced	Group	Action	Comments	Example uses
ROW_NUMBER	2005	Ranking	Assigns a number to each row of each partition; within each partition, the number is unique.	ORDER BY must be included.	Assigning serial numbers, randomly ordering groups, deduping, paging

Function	Version	Group	Action	Comments	Example uses
RANK	introduced 2005	Ranking	Assigns a number	ORDER BY must	Top N for each
KAINK	2005	Kalikilig	to each row of	be included.	partition
			each partition.	be meruded.	purcicion
			Records with the		
			same value for		
			the ordering		
			expression are		
			assigned the		
			same rank. Skips		
			values after ties		
			(e.g., 1, 2, 3, 3, 5).		
DENSE_RANK	2005	Ranking	Assigns a number	ORDER BY must	Numbering
			to each row of	be included.	distinct values
			each partition.		
			Records with the		
			same value for		
			the ordering		
			expression are assigned the		
			same rank. No		
			values are		
			skipped after ties.		
			(e.g., 1, 2, 3, 3, 4)		
NTILE	2005	Ranking	Divides the	ORDER BY must	Determine
		8	records in each	be included.	quartiles,
			partition into a		quintiles, or
			specified number		deciles (or any
			of groups as		otheriles).
			evenly as		
			possible.		
SUM	2005	Aggregates	Computes the	ORDER BY and	Totaling on
			total of the	window frame	multiple levels
			specified	capability was	in a single
			expression for the	added in SQL	query; in 2012
			records in the	Server 2012.	and later,
			specified partition.		running totals.
AVG	2005	Aggregates	Computes the	ORDER BY and	Averaging on
117 G	2000	1.551 054103	average of the	window frame	multiple levels
			specified	capability was	in a single
			expression for the	added in SQL	query; in 2012
			records in the	Server 2012.	and later,
			specified		moving
			partition.		averages.
MIN	2005	Aggregates	Finds the		0
				Server 2012.	
			specified partition.		and later, "minimums to
MIN	2005	Aggregates	Finds the minimum value of the specified expression for the records in the specified	ORDER BY and window frame capability was added in SQL Server 2012.	Finding minimum valu on multiple levels in a sing query; in 2012 and later,

Function	Version introduced	Group	Action	Comments	Example uses
MAX	2005	Aggregates	Finds the maximum value of the specified expression for the records in the specified partition.	ORDER BY and window frame capability was added in SQL Server 2012.	Finding maximum value on multiple levels in a single query; in 2012 and later, "maximums to date."
COUNT COUNTBIG	2005	Aggregates	Counts the records in the specified partition.	COUNT returns an Int; COUNTBIG returns a BigInt. ORDER BY and window frame capability was added in SQL Server 2012.	Counting on multiple levels in a single query; in 2012 and later, running counts.
VAR VARP	2005	Aggregates	Computes the variance of the expression for the records in the specified partition.	VAR computes the variance for the sample; VARP computes the variance for the population. ORDER BY and window frame capability was added in SQL Server 2012.	Compute variance for multiple levels in a single query.
STDEV STDEVP	2005	Aggregates	Computes the standard deviation of the expression for the records in the specified partition.	STDEV computes the standard deviation for the sample; STDEVP computes the standard deviation for the population. ORDER BY and window frame capability was added in SQL Server 2012.	Compute standard deviation for multiple levels in a single query.
LAG	2012	Analytic	Provides the value of the specified expression for a prior row in the specified partition.	ORDER BY must be included. Optional parameters let you specify how far back to look and a default value.	Compare multiple data points (such as data from multiple reporting periods) in a single record. Fill in missing values in a sequence.

Function	Version	Group	Action	Comments	Example uses
LEAD	introduced 2012	Analytic	Provides the value of the specified expression for a subsequent row in the specified partition.	ORDER BY must be included. Optional parameters let you specify how far forward to look and a default value.	Compare multiple data points (such as data from multiple reporting periods) in a single record. Fill in missing values in a sequence.
FIRST_VALUE	2012	Analytic	Provides the value of the specified expression for the first record in the specified partition.	ORDER BY must be included.	Pull data from "minimum" record (such as year of fewest sales) of a partition into other records.
LAST_VALUE	2012	Analytic	Provides the value of the specified expression for the last record in the specified partition.	ORDER BY must be included. "Last" is interpreted as "current" unless window frame is specified.	Pull data from "maximum" record (such as year of most sales) of a partition into other records.
CUME_DIST	2012	Analytic	Computes relative position of a record in the specified partition, based on the specified order.	ORDER BY must be included. Each record is assigned a value between 0 and 1; 0 is not used. Records with the same value for the specified order are assigned the same value.	Remove outliers (say, top and bottom 5%).
PERCENT_RANK	2012	Analytic	Computes relative rank of a row in the specified partition, based on the specified order.	ORDER BY must be included. Each record is assigned a value between 0 and 1. Records with the same value for the specified order are assigned the same value.	Remove outliers (say, top and bottom 5%).
PERCENTILE_CONT	2012	Analytic	Returns the value at the specified percentile in the specified partition, based on the specified order.	ORDER BY must be included. Interpolates to provide a continuous set of values.	Find the median for each group, find all records at or above (or at or below) a given percentile.

Function	Version introduced	Group	Action	Comments	Example uses
PERCENTILE_DISC	2012	Analytic	Returns the value at the specified percentile in the specified partition, based on the specified order.	ORDER BY must be included. Does not interpolate; only returns values actually in the partition.	Find the median for each group, find all records at or above (or at or below) a given percentile.

The PARTITION BY clause lets you divide the data into groups, much like GROUP BY. However, GROUP BY consolidates all the records with matching values into a single result record. PARTITION BY simply indicates the groups of records to which the specified function should be applied. The original records still appear in the result set.

This ability to divide records into groups for a single calculation is one of the big benefits of OVER. It means that, in a single query, you can compute aggregated results based on different groupings. For example, you might compute a salesperson's total sales by day, week, month and year, and put each of those in a separate column of a single record. The result set would have one record for each day, but each record would include weekly, monthly and annual totals. (There are examples like this in "Aggregate functions," later in this document.)

The ORDER BY clause indicates the order in which records are processed by the specified function. For the ranking functions and some of the analytic functions, ORDER BY is required as it's the ordering that determines the results. For other functions, ORDER BY is optional, but if it's used, it has an impact on the results. For example, using ORDER BY with SUM lets you compute running totals. (See "Running totals, running counts and moving averages," later in this document for examples.)

Window frame specification using RANGE and ROWS lets you apply a function to a subset of a partition; it was added in SQL Server 2012. RANGE lets you limit the calculation to a group of rows based on their values for the ordering expression, while ROWS lets you limit the calculation to a set number of rows around the current row. See "Aggregating subsets within partitions," later in this document, for more explanation and examples.

Before digging into the window functions and showing how to use them, it's worth commenting on the choice of the keyword "OVER," which may seem odd. I'm fairly certain the term is drawn from mathematics, where it's used to refer to the domain of a function. That is, a function that applies to the integers is sometimes referred to as a function "over the integers."

The same idea applies here, because we're specifying the domain for the function using PARTITION, ORDER and window frame.

In the rest of this paper, we'll look at each group of functions; along the way, we'll explore PARTITION, ORDER and window frames in detail. VFP examples in this paper use the Northwind database; SQL Server examples use the AdventureWorks 2014 database; those

examples that work in earlier versions can be run against AdventureWorks 2008, but the results may vary from those shown here. You can download AdventureWorks 2014 from https://msftdbprodsamples.codeplex.com/releases/view/125550; the same page links to a ReadMe file with instructions for installation.

CTEs: A quick review

Before digging in, I want to quickly cover *common table expressions* or *CTEs*, as they're generally called. Many of the examples in the rest of this document use CTEs, so I want to be sure you understand what they are and how they work.

A CTE is a query executed before the main query, in order to collect some data to be used in the main query. It's very similar to a derived table (that is, a query in the FROM clause), but easier to read and more useful. A CTE is easier to read because it's isolated from the main query rather than embedded in it. It's more useful because you can refer to the same CTE multiple times in the main query.

Listing 2 shows the syntax of a query with a CTE. The key elements are the WITH clause that names the CTE, the AS clause that contains the CTE query, and the main query that presumably uses the CTE.

Listing 2. A CTE is analogous to a derived table, but more useful.

```
WITH CTEName [(list of field names)]
AS
(SELECT <rest of query>)
SELECT <field list>
   FROM <tables, presumably including CTEName, and join conditions>
   <rest of query>
```

Listing 3 shows a fairly simple use of a CTE; it's included in the materials for this session as AnnualProductSales.SQL. The CTE groups data and the main query joins the grouped data to an underlying look-up table to provide descriptions. In this case, you could, in fact, simply add the product name to the field list and the GROUP BY clause and get the same results, but I think this version is easier to maintain.

Listing 3. Here the CTE computes annual sales totals for each product, and the main query adds the product name.

```
WITH csrSalesByProduct (ProductID, nYear, TotalSales)
AS
(SELECT ProductID, YEAR(OrderDate), SUM(LineTotal)
   FROM [Sales].[SalesOrderHeader] SOH
    JOIN [Sales].[SalesOrderDetail] SOD
        ON SOH.SalesOrderID = SOD.SalesOrderDetailID
    GROUP BY ProductID, YEAR(OrderDate))
SELECT SBP.ProductID, Name, nYear, TotalSales
   FROM csrSalesByProduct SBP
    JOIN Production.Product
```

```
ON SBP.ProductID = Product.ProductID
ORDER BY nYear, TotalSales DESC;
```

Be aware that you can have multiple CTEs in a single query; they're comma-separated (following the terminating parenthesis for the CTE definition). Any CTE can list any preceding CTE in its own FROM clause.

Ranking records

As I indicated, my first foray into OVER was for finding the top N records in each group. 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, thanks to OVER, you can do it with a single query.

The VFP solution

Collecting the basic data you need to solve this problem in VFP is straightforward. **Listing 4** (EmployeeSalesByYear.PRG in the materials for this session) shows a query that provides each employee's sales by year; **Figure 1** shows part of the results.

Listing 4. 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
```

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 4 produces the total sales for each employee by year.

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 5** (TopnEmployeeSalesByYear-Loop.PRG in the materials for this session) shows the code; **Figure 2** shows the result.

Listing 5. 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
CREATE CURSOR csrTopEmployeeSalesByYear ;
  (FirstName C(10), LastName C(20), ;
  OrderYear N(4), TotalSales Y)
SELECT distinct OrderYear ;
  FROM csrEmpSalesByYear ;
  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 OrderYear, TotalSales DESC
ENDSCAN
USE IN csrYears
USE IN csrEmpSalesByYear
```

```
SELECT csrTopEmployeeSalesByYear
```

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 5 produces these results.

The first query is just a simpler version of Listing 4, omitting the Employees table and the ORDER BY clause; both of those are 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 6** (TopnEmployeeSalesByYear-Loop2.PRG in the materials for this session) 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 6. The code in Listing 5 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 ;
 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 GROUP BY and HAVING to keep only those that represent the top n records. Finally, it adds the employee name. **Listing 7** (TopNEmployeeSalesByYear-Trick.prg in the materials for this session) shows the code.

Listing 7. 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;</pre>
      ON Employees.EmployeeID = csrTop5.EmployeeID ;
 ORDER BY OrderYear, TotalSales DESC ;
 INTO CURSOR csrTopEmployeeSalesByYear
```

The first query here is just a variant of Listing 4. 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 8**. Records are matched up first by having the same year and then by having sales in the first instance of the table be the same or more than sales in the second instance. This join condition 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 8. 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 9** (included in the materials for this session 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 7 before applying GROUP BY. 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 fields ESBY2.EmployeeID and ESBY2.TotalSales to the field list, 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 9. This query demonstrates the intermediate results for the derived table in **Listing 7**.

```
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
Γ ς	1006	21065 2000

Figure 3. The query in Listing 9 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

Finding the top 5 salespeople for each year in SQL Server uses a couple of CTEs and the OVER clause with one of the ranking functions. To work through the steps involved, though, we'll start with a slightly easier TOP n by group problem: find the three longest-standing employees in each department.

To do this, we want to rank records within a group and then keep the first n. My first instinct was to 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 10** (included in the materials for this session as EmployeeOrderNumber.SQL) shows a query that lists AdventureWorks 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 position of each record. **Figure 4** shows partial results.

Listing 10. 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;
```

FirstName	LastName	HireDate	EmployeeOrderNumber
Guy	Gilbert	2006-06-30	1
Kevin	Brown	2007-01-26	2
Roberto	Tamburello	2007-11-11	3
Rob	Walters	2007-12-05	4
Thierry	D'Hers	2007-12-11	5
David	Bradley	2007-12-20	6
JoLynn	Dobney	2007-12-26	7
Ruth	Ellerbrock	2008-01-06	8
Gail	Erickson	2008-01-06	9
Barry	Johnson	2008-01-07	10
Jossef	Goldberg	2008-01-24	11
Terri	Duffy	2008-01-31	12
Sidney	Higa	2008-02-02	13

Figure 4. The query in Listing 10 applies a rank to each employee by hire date.

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 function doesn't know anything about ties. However, the RANK function is aware of ties and assigns them the same value, and then skips the appropriate number of values. **Listing 11** (EmployeeRank.SQL in the materials for this session) 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.

Listing 11. The RANK function is aware of ties, assigning them the same value.

FirstName	LastName	HireDate	EmployeeRank
Guy	Gilbert	2006-06-30	1
Kevin	Brown	2007-01-26	2
Roberto	Tamburello	2007-11-11	3
Rob	Walters	2007-12-05	4
Thierry	D'Hers	2007-12-11	5
David	Bradley	2007-12-20	6
JoLynn	Dobney	2007-12-26	7
Ruth	Ellerbrock	2008-01-06	8
Gail	Erickson	2008-01-06	8
Barry	Johnson	2008-01-07	10
Jossef	Goldberg	2008-01-24	11
Terri	Duffy	2008-01-31	12
Sidney	Higa	2008-02-02	13

Figure 5. Using RANK assigns the same EmployeeRank 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 12** (included in the materials for this session 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. We're now using the StartDate field of EmployeeDepartmentHistory rather than the HireDate field, because we want to know when the employee joined the department, not when she was hired. (Note that we also look only at records where EndDate is null, so that we consider only people's current assignments.) **Figure 6** shows partial results; note that the numbering begins again for each department and, as before, that ties have the same value.

Listing 12. 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;

FirstName	LastName	StartDate	Name	EmployeeRank
Roberto	Tamburello	2007-11-11	Engineering	1
Gail	Erickson	2008-01-06	Engineering	2
Jossef	Goldberg	2008-01-24	Engineering	3
Terri	Duffy	2008-01-31	Engineering	4
Michael	Sullivan	2010-12-30	Engineering	5
Sharon	Salavaria	2011-01-18	Engineering	6
Thierry	D'Hers	2007-12-11	Tool Design	1
Rob	Walters	2010-05-31	Tool Design	2
Ovidiu	Cracium	2010-12-05	Tool Design	3
Janice	Galvin	2010-12-23	Tool Design	4
Stephen	Jiang	2011-01-04	Sales	1
Brian	Welcker	2011-02-15	Sales	2
Michael	Blythe	2011-05-31	Sales	3
Linda	Mitchell	2011-05-31	Sales	3
Jillian	Carson	2011-05-31	Sales	3

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

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 13** (LongestStandingEmployeesByDept.SQL in the materials for this session).

Listing 13. Once we have the rank for an item within its group, we just need to filter to get the TOP n items by group.

```
WHERE EndDate IS NULL)
SELECT FirstName, LastName, StartDate, Department
FROM EmpRanksByDepartment
WHERE EmployeeRank <= 3
ORDER BY Department, StartDate;</pre>
```

Figure 7 shows part of the result. Note that there are many more than three records 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.

FirstName	LastName	StartDate	Department
Diane	Margheim	2008-12-29	Research and Developm
Gigi	Matthew	2009-01-16	Research and Developm
Dylan	Miller	2009-02-08	Research and Developm
Stephen	Jiang	2011-01-04	Sales
Brian	Welcker	2011-02-15	Sales
Michael	Blythe	2011-05-31	Sales
Linda	Mitchell	2011-05-31	Sales
Jillian	Carson	2011-05-31	Sales
Garrett	Vargas	2011-05-31	Sales
Tsvi	Reiter	2011-05-31	Sales
Pamela	Ansman-Wolfe	2011-05-31	Sales
Shu	lto	2011-05-31	Sales
José	Saraiva	2011-05-31	Sales
David	Campbell	2011-05-31	Sales
Vamsi	Kuppa	2008-12-07	Shipping and Receiving

Figure 7. The query in Listing 13 provides the three longest-standing employees in each department. When there are ties, it may produce more than three results.

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 14** (TopSalesPeopleByYear.SQL in the materials for this session) shows the complete query.

Listing 14. Finding the top five salespeople by year requires cascading CTEs, plus the OVER clause.

```
WITH TotalSalesBySalesPerson AS
(SELECT BusinessEntityID, YEAR(OrderDate) AS nYear, SUM(SubTotal) AS TotalSales
FROM 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;</pre>
```

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.

FirstName	LastName	nYe	TotalSales
Tsvi	Reiter	2011	1521289.1881
Jillian	Carson	2011	1311627.2918
José	Saraiva	2011	1175007.4753
Linda	Mitchell	2011	1149715.3253
Shu	lto	2011	967597.2899
Jillian	Carson	2012	4317306.5741
Linda	Mitchell	2012	3834908.674
Michael	Blythe	2012	3375456.8947
Jae	Pak	2012	3014278.0472
Tsvi	Reiter	2012	2674436.3518
Linda	Mitchell	2013	4111294.9056
Jae	Pak	2013	4106064.0146
Michael	Blythe	2013	3985374.8995
Jillian	Carson	2013	3396776.2674
Ranjit	Varkey Chudukatil	2013	2646078.409

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

With the basics covered, let's look at some other uses for OVER with the ranking functions.

Deduping

One of the most straightforward uses for OVER is identifying and removing duplicate records. Deduping data is a big question and generally more of a business problem than a code problem. That is, usually, the problem is having multiple similar, but not identical, records; for example, some organizations have a record for me alone, as well as one for me and my husband as a couple. No code alone is going to solve that problem.

However, OVER is very handy for the narrower problem of having records that are identical in some list of fields. To identify the duplicates, we can partition on the fields that must be the same to indicate a match, and assign each record in the partition a unique value. Then, we can delete all the extras.

Because I don't want to delete records from the sample AdventureWorks database, this example uses a copy of the Person table, created as shown in **Listing 15**.

Listing 15. This code creates a temporary table #People containing the primary key and name information from the AdventureWorks Person table.

```
CREATE TABLE #People
  (PersonID INT, FirstName nVarchar(50),
  MiddleName nVarchar(50), LastName nVarchar(50));
INSERT INTO #People
  SELECT BusinessEntityID, FirstName, MiddleName, LastName
   FROM Person.Person;
```

The first step in deduping is matching records with the exact same name and assigning each a different number. You can do that with the query in **Listing 16**. In this case, we order the duplicates by their primary key, PersonID, because we need some ordering; you could use any of the fields in the query in the ORDER BY. Partial results are shown in **Figure 9**; note that the two records for Aaron Con are assigned 1 and 2 respectively in the RecNo column.

Listing 16. This query matches records by name and assigns each a unique number within those with the same name.

PersonID	FirstName	MiddleName	LastName	RecNo
1305	A .	Francesca	Leonetti	1
2321	Α.	Scott	Wright	1
222	A. Scott	NULL	Wright	1
5508	Aaron	NULL	Alexander	1
5504	Aaron	NULL	Bryant	1
5500	Aaron	NULL	Butler	1
5519	Aaron	NULL	Chen	1
5488	Aaron	NULL	Coleman	1
727	Aaron	NULL	Con	1
2272	Aaron	NULL	Con	2
15693	Aaron	NULL	Edwards	1
5496	Aaron	NULL	Flores	1
5502	Aaron	NULL	Foster	1
5503	Aaron	NULL	Gonzales	1
5512	Aaron	NULL	Griffin	1

Figure 9. The instances of each name are numbered, starting at 1, providing a way to identify duplicates.

To dedupe the #People table, we need to delete all the records where RecNo is greater than 1. The solution uses a CTE and the SQL DELETE command. The CTE is similar to the query in **Listing 16**, but doesn't include the name fields in its field list. The full query to dedupe the table is shown in **Listing 17**.

Listing 17. To remove exact duplicates from a table, use ROW_NUMBER to number each copy and then delete all those whose number is not 1.

The materials for this session include DeDupe.SQL, which creates the #People table, dedupes it, and then deletes it. To demonstrate that it works, it shows the count for the table before and after deduping.

Which function to use?

As indicated above, there are three functions that return similar, but not identical results: ROW_NUMBER, RANK, and DENSE_RANK. We've seen examples for ROW_NUMBER (deduping) and RANK (finding the top N), but when else would you use these and how do you know which one to use?

Another use for ROW_NUMBER is randomly ordering groups. The idea is to assign each record a random number and then use ROW_NUMBER against that field to generate a random ordering.

The first step, assigning each record a random number, is a little harder than you might expect. Calling SQL Server's RAND function with no parameters in a query produces the same result on each row. That is, SQL Server collapses it to a single call. There are a variety of solutions (see, for example, <u>http://tinyurl.com/n58svm5</u>), but most of them boil down to using CHECKSUM(NEWID())). **Listing 18** (included in the materials for this session as RandomOrderInDept.SQL) demonstrates the one that looks best to me. The CTE calls RAND, passing CHECKSUM(NEWID()) as a seed, thus ensuring that the function is called for each row. The main query applies ROW_NUMBER to the random field. **Figure 10** shows partial results; note that the results will change each time you run the query.

Listing 18. You can use ROW_NUMBER to randomly order records within groups.

```
JOIN HumanResources.Department
```

```
ON EmpsByDept.DepartmentID = Department.DepartmentID
```

JOIN Person.Person

```
ON EmpsByDept.BusinessEntityID = Person.BusinessEntityID
```

ORDER BY Name, EmployeeRand;

FirstName	LastName	StartDate	Name	EmployeeRand
Chris	Norred	2009-03-06	Document Control	1
Zainal	Arifin	2009-01-04	Document Control	2
Tengiz	Kharatishvili	2008-12-16	Document Control	3
Karen	Berge	2009-02-09	Document Control	4
Sean	Chai	2009-01-22	Document Control	5
Jossef	Goldberg	2008-01-24	Engineering	1
Sharon	Salavaria	2011-01-18	Engineering	2
Michael	Sullivan	2010-12-30	Engineering	3
Gail	Erickson	2008-01-06	Engineering	4
Roberto	Tamburello	2007-11-11	Engineering	5
Terri	Duffy	2008-01-31	Engineering	6
Ken	Sánchez	2009-01-14	Executive	1
Laura	Norman	2013-11-14	Executive	2
Gary	Altman	2009-12-02	Facilities and Maintenance	1
Magnus	Hedlund	2009-12-21	Facilities and Maintenance	2

Figure 10. Applying ROW_NUMBER to a randomly generated field lets you randomly order each group.

Another use for ROW_NUMBER is paging records, as you might for a website. You can use ROW_NUMBER to number the records in a CTE and then return only the rows in the range for the specified page.

DENSE_RANK is useful when you want to number distinct values for the ordering criteria. Suppose you want to get a list of the current job titles in each department and number them alphabetically. You only want to list each job title once, so this calls for SELECT DISTINCT. Your first attempt might be the query in **Listing 19**. As the partial results in **Figure 11** indicate, it doesn't work; once a row number is added, each row is different, so DISTINCT doesn't remove any records.

Listing 19. ROW_NUMBER and DISTINCT don't mix, because the unique values returned by ROW_NUMBER make rows that should be the same different from each other.

```
SELECT DISTINCT Name, JobTitle,
    ROW_NUMBER() OVER
        (PARTITION BY Department.DepartmentID ORDER BY JobTitle) AS JobNum
    FROM [HumanResources].[EmployeeDepartmentHistory]
    JOIN [HumanResources].[Department]
    ON EmployeeDepartmentHistory.DepartmentID = Department.DepartmentID
```

JOIN [HumanResources]. [Employee]

```
ON EmployeeDepartmentHistory.BusinessEntityID = Employee.BusinessEntityID WHERE EndDate IS null;
```

Name	JobTitle	JobNum
Document Control	Control Specialist	1
Document Control	Control Specialist	2
Document Control	Document Control Assistant	3
Document Control	Document Control Assistant	4
Document Control	Document Control Manager	5
Engineering	Design Engineer	1
Engineering	Design Engineer	2
Engineering	Design Engineer	3
Engineering	Engineering Manager	4
Engineering	Senior Design Engineer	5
Engineering	Vice President of Engineering	6
Executive	Chief Executive Officer	1
Executive	Chief Financial Officer	2
Facilities and Maintenance	Facilities Administrative Assistant	1
Facilities and Maintenance	Facilities Manager	2

Figure 11. When you use ROW_NUMBER with SELECT DISTINCT, rows that should be the same are different.

The solution is to use DENSE_RANK instead, as in **Listing 20** (NumberedJobTitles.SQL in the materials for this session). Since rows that otherwise match are given the same value by DENSE_RANK, they can then be removed by DISTINCT. **Figure 12** shows partial results; as you can see, each job title is listed only once for each department.

Listing 20. DENSE_RANK assigns the same value to matching rows, which allows DISTINCT to remove duplicates.

```
SELECT DISTINCT Name, JobTitle,
DENSE_RANK() OVER
(PARTITION BY Department.DepartmentID ORDER BY JobTitle) AS JobSerial
FROM [HumanResources].[EmployeeDepartmentHistory]
JOIN [HumanResources].[Department]
ON EmployeeDepartmentHistory.DepartmentID = Department.DepartmentID
JOIN [HumanResources].[Employee]
ON EmployeeDepartmentHistory.BusinessEntityID = Employee.BusinessEntityID
WHERE EndDate IS null;
```

Name	JobTitle	JobSerial
Document Control	Control Specialist	1
Document Control	Document Control Assistant	2
Document Control	Document Control Manager	3
Engineering	Design Engineer	1
Engineering	Engineering Manager	2
Engineering	Senior Design Engineer	3
Engineering	Vice President of Engineering	4
Executive	Chief Executive Officer	1
Executive	Chief Financial Officer	2
Facilities and Maintenance	Facilities Administrative Assistant	1
Facilities and Maintenance	Facilities Manager	2
Facilities and Maintenance	Janitor	3
Facilities and Maintenance	Maintenance Supervisor	4
Finance	Accountant	1
Finance	Accounts Manager	2

Figure 12. The JobSerial field, created with DENSE_RANK, numbers each distinct job in each department.

Dividing into percentiles

The final function in the Ranking group, NTILE, divides the records in each partition 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 21** (SalesQuartiles.SQL in the materials for this session) computes the total sales for each salesperson by year, and then divides each year's sales into four groups (quartiles) from highest to lowest. **Figure 13** 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 21. 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 DESC) AS Quartile
   FROM csrAnnualSales
```

SalesPersonID	OrderYear	TotalSales	Quartile
279	2011	1521289.1881	1
277	2011	1311627.2918	1
282	2011	1175007.4753	1
276	2011	1149715.3253	2
281	2011	967597.2899	2
275	2011	875823.8318	2
280	2011	648485.5862	3
283	2011	599987.9444	3
278	2011	500091.8202	4
274	2011	28926.2465	4
277	2012	4317306.5741	1
276	2012	3834908.674	1
275	2012	3375456.8947	1
289	2012	3014278.0472	1

Figure 13. 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 22**), you get quintiles instead of quartiles, as in **Figure 14**.

Listing 22. 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 DESC) AS Quintile

SalesPersonID	OrderYear	TotalSales	Quintile
279	2011	1521289.1881	1
277	2011	1311627.2918	1
282	2011	1175007.4753	2
276	2011	1149715.3253	2
281	2011	967597.2899	3
275	2011	875823.8318	3
280	2011	648485.5862	4
283	2011	599987.9444	4
278	2011	500091.8202	5
274	2011	28926.2465	5
277	2012	4317306.5741	1
276	2012	3834908.674	1
275	2012	3375456.8947	1
289	2012	3014278.0472	2
279	2012	2674436.3518	2

Figure 14. Here, 5 was passed to NTILE(), so there are five groups for each year. As before, the group sizes are as even as possible.

Later in this paper (see "Searching by percentile"), we'll look at functions that let you ask where the diving point is between various percentiles.

Aggregate functions

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 23** computes the yearly, monthly and daily number sold for each product; **Figure 15** shows a portion of the results. 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 23. Combine OVER with the aggregate functions to 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 15. 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 24** (included in the materials for this session as SalesByYearMonthDay.SQL). **Figure 16** shows partial results.

Listing 24. 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
OPDER BY BroductID, OrderDate
```

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 16. The query in Listing 24 results in one record per date, each holding yearly, monthly and daily totals.

Computing percentages

You can use OVER to compute what percent of a total a particular record represents. **Listing 25** 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 17** shows partial results. The query is included in the materials for this session as SalesByYearMonthDayWithPcts.SQL.

Listing 25. 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, OrderDate)
 / SUM(SOD.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	ProductID	Yearly	Monthly	Daily	PctOfYear	PctOfMonth
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 17. 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.

Counting groups

Use OVER with COUNT to put the size of a group into the records in the group. For example, suppose you're preparing a staff directory to be sorted alphabetically with a break after the employees beginning with each initial letter. In order to have a good layout, you might want to know how many staff members begin with a given letter. The query in **Listing 26** (CountByInitial.SQL in the materials for this session) gives you what you need; **Figure 18** shows partial results.

Listing 26. By partitioning by the first letter of the last name, COUNT tells how many records begin with the same letter.

```
SELECT LEFT(LastName,1) AS Initial, LastName, FirstName,
        COUNT(*) OVER (PARTITION BY LEFT(LastName,1)) AS CountByInitial
FROM Person.Person
        JOIN HumanResources.Employee
        ON Person.BusinessEntityID = Employee.BusinessEntityID
        ORDER BY LastName, FirstName;
```

Initial	LastName	FirstName	CountByInitial
Α	Abbas	Syed	13
Α	Abercrombie	Kim	13
Α	Abolrous	Hazem	13
Α	Ackerman	Pilar	13
Α	Adams	Jay	13
Α	Ajenstat	François	13
Α	Alberts	Amy	13
Α	Alderson	Greg	13
Α	Alexander	Sean	13
Α	Altman	Gary	13
Α	Anderson	Nancy	13
Α	Ansman-Wolfe	Pamela	13
Α	Arifin	Zainal	13
В	Bacon	Dan	20
B	Baker	Brvan	20

Figure 18. The CountByInitial column indicates how many names begin with the same letter as the current record. Having that value can be useful for layout.

Running totals, running counts 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; OVER with aggregate functions was restricted to PARTITION.

The ability to include ORDER BY with OVER and aggregate functions lets you compute running totals, running counts 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 27** (included in the materials for this session 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 19**; look at the rows for customer 11019 to see the monthly running total change for a customer.

Listing 27. 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(SubTotal) OVER
(PARTITION BY CustomerID, YEAR(OrderDate)) AS Yearly,
SUM(SubTotal) OVER
(PARTITION BY CustomerID, YEAR(OrderDate), MONTH(OrderDate)) AS Monthly,
SUM(SubTotal) OVER
(PARTITION BY CustomerID, OrderDate) AS Daily,
SUM(SubTotal) OVER
(PARTITION BY CustomerID, YEAR(OrderDate) ORDER BY OrderDate)
AS YearlyRunning,
SUM(SubTotal) OVER
(PARTITION BY CustomerID, YEAR(OrderDate), MONTH(OrderDate)
ORDER BY OrderDate) AS MonthlyRunning
```

CustomerID	OrderDate	Yearly	Monthly	Daily	YearlyRunning	MonthlyRunning
11017	2011-06-14 00:00:00.000	3374.99	3374.99	3374.99	3374.99	3374.99
11017	2013-06-03 00:00:00.000	2316.97	2316.97	2316.97	2316.97	2316.97
11017	2014-03-16 00:00:00.000	742.35	742.35	742.35	742.35	742.35
11018	2011-06-19 00:00:00.000	3399.99	3399.99	3399.99	3399.99	3399.99
11018	2013-06-18 00:00:00.000	2341.97	2341.97	2341.97	2341.97	2341.97
11018	2014-03-26 00:00:00.000	791.32	791.32	791.32	791.32	791.32
11019	2013-07-15 00:00:00.000	342.47	37.97	37.97	37.97	37.97
11019	2013-08-04 00:00:00.000	342.47	101.96	53.99	91.96	53.99
11019	2013-08-13 00:00:00.000	342.47	101.96	47.97	139.93	101.96
11019	2013-09-28 00:00:00.000	342.47	8.99	8.99	148.92	8.99
11019	2013-10-08 00:00:00.000	342.47	156.26	7.28	156.20	7.28
11019	2013-10-25 00:00:00.000	342.47	156.26	148.98	305.18	156.26
11019	2013-12-23 00:00:00.000	342.47	37.29	37.29	342.47	37.29
11019	2014-01-22 00:00:00.000	540.23	2.29	2.29	2.29	2.29
11019	2014-02-08 00:00:00.000	540.23	48.77	24.99	27.28	24.99

FROM Sales.SalesOrderHeader SOH
ORDER BY CustomerID, OrderDate;

Figure 19. Include ORDER BY when using OVER with SUM() to get a running total.

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 use ORDER BY with COUNT, you get a running count of records. The query in **Listing 28** (included in the session materials as RunningOrderCount.SQL) shows the total number of orders placed in a year and the running total through the year; **Figure 20** shows partial results. As with SUM, when you use only PARTITION, you get the count for the whole partition. When you add ORDER BY, you get a running count that changes on each value of the ordering expression (OrderDate, in the example).

Listing 28. Use COUNT with ORDER BY to get a running count.

```
SELECT DISTINCT OrderDate, YEAR(OrderDate) AS OrderYear,
COUNT(SalesOrderNumber) OVER (PARTITION BY YEAR(OrderDate)) AS OrdersThisYear,
COUNT(SalesOrderNumber) OVER (PARTITION BY YEAR(OrderDate) ORDER BY OrderDate)
AS RunningOrdersThisYear
FROM [Sales].[SalesOrderHeader]
ORDER BY OrderDate
```

OrderDate	OrderYear	OrdersThisYear	RunningOrdersThisYear
2011-05-31 00:00:00.000	2011	1607	43
2011-06-01 00:00:00.000	2011	1607	47
2011-06-02 00:00:00.000	2011	1607	52
2011-06-03 00:00:00.000	2011	1607	54
2011-06-04 00:00:00.000	2011	1607	59
2011-06-05 00:00:00.000	2011	1607	63
2011-06-06 00:00:00.000	2011	1607	66
2011-06-07 00:00:00.000	2011	1607	69
2011-06-08 00:00:00.000	2011	1607	75
2011-06-09 00:00:00.000	2011	1607	78
2011-06-10 00:00:00.000	2011	1607	82
2011-06-11 00:00:00.000	2011	1607	86
2011-06-12 00:00:00.000	2011	1607	88
2011-06-13 00:00:00.000	2011	1607	95
2011-06-14 00:00:00.000	2011	1607	99

Figure 20. Using COUNT with OVER lets you compute counts and running counts by partition.

When you apply ORDER BY to AVG, you get a moving average, that is, the average of all the records in the partition up to this point. The last record in the group will show the average for the whole group. (This type of moving average is called a *cumulative moving average*.) **Listing 29** (included in the materials for this session as SalesWithMovingAverage.SQL) demonstrates by computing the moving average of sales for a customer within a year. **Figure 21** shows partial results.

Listing 29. 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,
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	MovingAvg
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
11004	2011-06-25 00:00:00.000	3399.99	3399.99	3399.99
11004	2013-06-24 00:00:00.000	4796.02	2376.96	2376.96
11004	2013-10-01 00:00:00.000	4796.02	2419.06	2398.01

Figure 21. 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.

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 30** shows the minimum and maximum quantity in a single order to date for each product. **Figure 22** shows partial results. The query is included in the materials for this session as SalesByYearMonthDayWithMinMax.SQL.

Listing 30. 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	ProductID	Yearly	Monthly	Daily	MinOrder	MaxOrder
2011-05-31 00:00:00.000	707	331	24	24	1	4
2011-07-01 00:00:00.000	707	331	58	58	1	5
2011-08-01 00:00:00.000	707	331	96	55	1	5
2011-08-31 00:00:00.000	707	331	96	41	1	9
2011-10-01 00:00:00.000	707	331	141	77	1	10
2011-10-31 00:00:00.000	707	331	141	64	1	10
2011-12-01 00:00:00.000	707	331	12	12	1	10
2012-01-01 00:00:00.000	707	1278	61	31	1	10
2012-01-29 00:00:00.000	707	1278	61	30	1	10
2012-02-29 00:00:00.000	707	1278	27	27	1	10
2012-03-30 00:00:00.000	707	1278	93	93	1	10
2012-04-30 00:00:00.000	707	1278	52	52	1	10
2012-05-30 00:00:00.000	707	1278	162	162	1	10
2012-06-30 00:00:00.000	707	1278	214	214	1	14
2012-07-31 00:00:00.000	707	1278	197	197	1	24

Figure 22. 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.

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 for a product in a given year, 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 or a self-join 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 31** (SalesByYearWithTwoYearTotal.SQL in the materials for this session) shows the query; **Figure 23** 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.

Listing 31. 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;

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 23. The TwoYear column here is computed using the ROWS clause.

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 frame* or just *frame*. You can specify UNBOUNDED PRECEDING as the start point to indicate that the frame begins with the first row of the partition, or UNBOUNDED FOLLOWING as the end point to say that the frame ends with the last row of the partition. Also, note that you can specify a frame where all the rows 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 for the next two years, not including the current year. It's easy to see why someone might want to use ROW BETWEEN 3 PRECEDING and 1 PRECEDING to compute, say, the average sales of a product over the last three years (or months) for comparison to the sales for the current year (or month).

It's important to realize that ROWS is unaware of the data in other fields of the specified rows. It simply uses rows within the partition in the way you specify. For example, you might think that you could compute the number of each product sold by date and include the total sales for the week with that date in the middle with the query in **Listing 32** (included in the materials for this session as ProductSalesWithWeekly-WRONG.SQL), specifying 3 rows before and 3 rows after the current row. However, as the partial results

in **Figure 24** show, the WeekSales column is wrong; it doesn't notice that some dates are missing because the product wasn't sold every day.

Listing 32. This attempt to calculate daily and weekly sales by product is flawed. ROWS simply counts records forward and backward from the current record without paying any attention to their contents.

```
WITH csrSalesByProduct (ProductID, OrderDate, NumSold)
AS
(SELECT ProductID, OrderDate, SUM(OrderQty)
FROM [Sales].[SalesOrderHeader] SOH
JOIN [Sales].[SalesOrderDetail] SOD
ON SOH.SalesOrderID = SOD.SalesOrderDetailID
GROUP BY ProductID, OrderDate)
SELECT OrderDate, ProductID, NumSold AS TodaysSales,
SUM(NumSold) OVER (
PARTITION BY ProductID ORDER BY ORDERDATE
ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING) AS WeekSales
FROM csrSalesByProduct
ORDER BY ProductID, OrderDate;
```

OrderDate	ProductID	TodaysSales	WeekSales
2011-05-31 00:00:00.000	707	3	6
2011-06-05 00:00:00.000	707	1	7
2011-06-06 00:00:00.000	707	1	8
2011-06-08 00:00:00.000	707	1	9
2011-06-09 00:00:00.000	707	1	7
2011-06-13 00:00:00.000	707	1	7
2011-06-16 00:00:00.000	707	1	7
2011-06-20 00:00:00.000	707	1	8
2011-06-29 00:00:00.000	707	1	8
2011-06-30 00:00:00.000	707	1	8
2011-07-01 00:00:00.000	707	2	8
2011-07-06 00:00:00.000	707	1	8
2011-07-08 00:00:00.000	707	1	8
2011-07-21 00:00:00.000	707	1	8
2011-07-22 00:00:00.000	707	1	7

Figure 24. It's easy to misuse ROWS by assuming that it takes data values into consideration. Here, the WeekSales column is wrong, containing the units of the product sold in seven consecutive records, whether or not those records represent consecutive days.

On the other hand, the RANGE keyword lets you specify rows based on value rather than position. You can't specify a number at either end with RANGE, though. You can start with CURRENT ROW or UNBOUNDED PRECEDING and end with CURRENT ROW or UNBOUNDED FOLLOWING.

You can also specify RANGE 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 33**, where we list each order, but include the daily sales total for the salesperson. Partial results are shown in **Figure 25**. The query is included in the materials for this session as SalesWithDailyTotal.SQL.

Listing 33. 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;
```

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

RANGE doesn't let you narrow down to a group of 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 34** (included in the materials for this session as SalesWithReverseRunningTotalByDay.SQL) computes such a reverse running total of sales for the salesperson. **Figure 26** 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 third and fourth rows shown have the same value because they're for the same day.

Listing 34. 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 the 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	SalesPersonID	SubTotal	ReverseRunningTotal
2011-07-01 00:00:00.000	274	20544.7015	1092123.8561
2011-08-01 00:00:00.000	274	2039.994	1071579.1546
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

Figure 26. 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 35** (included in the materials for this session as SalesWithReverseRunningTotal.SQL). Partial results are shown in **Figure 27**.

Listing 35. 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	SalesPersonID	SubTotal	ReverseRunningTotal
2011-07-01 00:00:00.000	274	20544.7015	1092123.8561
2011-08-01 00:00:00.000	274	2039.994	1071579.1546
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

Figure 27. 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.)

I should also note that although many of these examples order by date, you can order by pretty much anything. My examples use ORDER BY OrderDate or one of its components (month, year) simply because it's easy to think of business examples where we want to calculate things based on the date or the month or the year.

Analytical functions

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, containing CUME_DIST, PERCENTILE_CONT, PERCENTILE_DISC, and PERCENT_RANK, looks at percentiles and distributions.

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, along with the prior year's sales and the next year's sales. 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 36** (included in the materials for this session 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 28** show, you get the null value for the previous year in the first 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 36. 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 28. In VFP, 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 in 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 37** (SalesByYearWithPrevAndFoll.SQL in the materials for this session) 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 29** 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 37. 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
```

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

FROM csrYearlySales ORDER BY ProductID, OrderYear;

Figure 29. 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, so you can do a calculation based on data from a preceding or following record. 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 38** (included in the materials for this session 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 30** shows, we don't actually have five years' sales data, so every record contains some nulls.

Listing 38. 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 30. 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 38.

It's important to keep in mind that, like ROWS, LAG and LEAD are about position, not range. LAG returns the value of the expression for the preceding record in the partition, even if that record doesn't represent the immediately preceding value of the ordering expression. In the examples above, if some product hadn't been sold during a particular year (perhaps the materials to produce it weren't available), the PrevYear and FollYear columns wouldn't necessarily represent the immediately preceding and following year. That's easier to see if you consider showing sales for three days at a time, as in **Listing 39** (included in the materials for this session as SalesByDayWithPrevAndFoll-WRONG.SQL), which is the same query as in **Listing 37**, except that it computes daily sales. Partial results are shown in **Figure 31**, where it's clear that PrevDate and FollDate show sales for the first previous day and the next day on which the product was sold, not the previous day and next day.

Listing 39. When looking at daily sales (rather than yearly), the real meaning of LAG and LEAD becomes more apparent.

```
WITH csrDailySales (OrderDate, ProductID, NumSold)
AS
(SELECT OrderDate , ProductID, SUM(OrderQty) AS NumSold
  from Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
    ON soh.SalesOrderID = sod.SalesOrderID
  group by OrderDate, ProductID)
SELECT OrderDate, ProductID,
   LAG(NumSold) OVER (PARTITION BY ProductID ORDER BY OrderDate) AS PrevDate,
```

NumSold AS CurrYear, LEAD(NumSold) OVER (PARTITION BY ProductID ORDER BY OrderDate) AS FollDate FROM csrDailySales ORDER BY ProductID, OrderDate

OrderDate	ProductID	PrevDate	CurrDate	FollDate
2011-05-31 00:00:00.000	707	NULL	24	58
2011-07-01 00:00:00.000	707	24	58	55
2011-08-01 00:00:00.000	707	58	55	41
2011-08-31 00:00:00.000	707	55	41	77
2011-10-01 00:00:00.000	707	41	77	64
2011-10-31 00:00:00.000	707	77	64	12
2011-12-01 00:00:00.000	707	64	12	31
2012-01-01 00:00:00.000	707	12	31	30
2012-01-29 00:00:00.000	707	31	30	27
2012-02-29 00:00:00.000	707	30	27	93
2012-03-30 00:00:00.000	707	27	93	52
2012-04-30 00:00:00.000	707	93	52	162
2012-05-30 00:00:00.000	707	52	162	214

Figure 31. These results make it clear that LAG and LEAD operate based on position, not data.

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 40** (PayHistoryWithOrig.SQL in the materials for this session) 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 32** to see an employee with multiple records.

Listing 40. FIRST_VALUE lets you include data from the first record in the partition with each record in the result.

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

While the example in **Listing 40** doesn't seem terribly useful, a small extension of the idea does. The query in **Listing 41** (PayHistoryWithPctInc.SQL in the materials for this session) computes the percentage increase from the original pay rate and includes only those records that represent changes in pay in the result. (The query also includes the date for the original pay rate.) The CTE here is required in order to be able to use the computed increase in the WHERE clause. **Figure 33** shows partial results; for example, David Bradley started in December, 2012 at \$24/hour. He got a raise of nearly 20% in July, 2009 and an additional raise at the end of April, 2012, making his pay rate more than 56% higher than when he started.

Listing 41. 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, OrigDate, Inc)
AS
(SELECT FirstName, LastName, Rate, RateChangeDate,
        FIRST VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID ORDER BY RateChangeDate),
        FIRST VALUE(RateChangeDate) OVER
          (PARTITION BY EPH.BusinessEntityID ORDER BY RateChangeDate),
        CAST((100.00 * Rate/FIRST VALUE(Rate) OVER
          (PARTITION BY EPH.BusinessEntityID ORDER BY RateChangeDate)-100)
          AS DECIMAL(5,2))
   FROM Person.Person
    JOIN [HumanResources].[EmployeePayHistory] EPH
      ON Person.BusinessEntityID = EPH.BusinessEntityID)
SELECT *
   FROM csrPayHikes
   WHERE Inc <> 0
   ORDER BY LastName, FirstName, RateChangeDate;
```

FirstName	LastName	Rate	RateChangeDate	OrigRate	OrigDate	Inc
David	Bradley	28.75	2009-07-15 00:00:00.000	24.00	2007-12-20 00:00:00.000	19.79
David	Bradley	37.50	2012-04-30 00:00:00.000	24.00	2007-12-20 00:00:00.000	56.25
John	Frum	7.00	2011-12-01 00:00:00.000	6.50	2009-03-03 00:00:00.000	7.69
John	Frum	9.50	2013-07-14 00:00:00.000	6.50	2009-03-03 00:00:00.000	46.15
Marc	Ingle	7.25	2011-12-01 00:00:00.000	6.50	2009-01-16 00:00:00.000	11.54
Marc	Ingle	9.50	2013-07-14 00:00:00.000	6.50	2009-01-16 00:00:00.000	46.15
David	Johnson	7.25	2011-12-01 00:00:00.000	6.50	2008-12-02 00:00:00.000	11.54
David	Johnson	9.50	2013-07-14 00:00:00.000	6.50	2008-12-02 00:00:00.000	46.15
Russell	King	7.25	2011-12-01 00:00:00.000	6.50	2009-02-21 00:00:00.000	11.54
Russell	King	9.50	2013-07-14 00:00:00.000	6.50	2009-02-21 00:00:00.000	46.15
Reed	Koch	7.25	2011-12-01 00:00:00.000	6.50	2009-02-02 00:00:00.000	11.54
Reed	Koch	9.50	2013-07-14 00:00:00.000	6.50	2009-02-02 00:00:00.000	46.15
David	Lawrence	7.25	2011-12-01 00:00:00.000	6.50	2009-02-14 00:00:00.000	11.54
David	Lawrence	9.50	2013-07-14 00:00:00.000	6.50	2009-02-14 00:00:00.000	46.15
George	Li	7.25	2011-12-01 00:00:00.000	6.50	2008-12-21 00:00:00.000	11.54

Figure 33. The Inc column uses FIRST_VALUE in a computation to figure out how much of a cumulative raise each employee has received.

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 40**, so we have the query shown in **Listing 42** (included in the materials for this session as PayHistoryWithLast.SQL). We get results like those shown in **Figure 34**. 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 42. 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 CurrRate
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 34. 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 frame notation (described in "Aggregating subsets within partitions," earlier in this paper). The default frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. To get a value from the last record of the partition, we need RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. The query in **Listing 43** (included in the materials for this session as PayHistoryWithOrigAndCurr.SQL) shows the pay rate represented by the particular record, the original pay rate and the current pay rate. **Figure 35** shows partial results; the last three records in the figure demonstrate the correct results for an employee with multiple pay rates.

Listing 43. 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	OrigRate	CurrRate
Mary	Baker	13.45	2009-12-25 00:00:00.000	13.45	13.45
Angela	Barbariol	11.00	2009-01-20 00:00:00.000	11.00	11.00
David	Barber	13.4615	2009-01-12 00:00:00.000	13.4615	13.4615
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 35. 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 let you work around a limitation of the MIN and MAX aggregate functions. The issue is that MIN and MAX give you the minimum or maximum value for the specified expression, but they don't give you a way to reach into other fields of the record that provides the minimum or maximum.

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 44** (MinMaxProductsSold.SQL in the materials for this session).

Listing 44. 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 group. But with FIRST_VALUE and LAST_VALUE, you can get exactly what you want, as in **Listing 45** (included in the materials for this session as SalesByYearWithWorstAndBest.SQL). **Figure 36** shows partial results.

Listing 45. 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(OrderOty) 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;
```

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

In addition, FIRST_VALUE and LAST_VALUE can answer questions more simply, that is, with less code. Suppose you want to get a list of AdventureWorks employees, with their current department, and their last previous department. Without these functions, you need two CTEs to get the name of the previous department, so you can join it to the current data, as in **Listing 46** (EmployeeWithPriorDept-TwoCTE.SQL in the materials for this session). The first CTE finds the latest date an employee's assignment to another department ended. The second CTE uses that date to find the appropriate record in EmployeeDepartmentHistory and joins it to Department to get the name of the department. Then, the main query joins that data with other employee data.

Listing 46. To find each employee's previous department and join it to the current data, you can use a pair of CTEs.

```
WITH LastXfer (BusinessEntityID, LastEndDate)
AS
(SELECT BusinessEntityID, MAX(EndDate)
 FROM HumanResources.EmployeeDepartmentHistory EDH
 WHERE EndDate IS NOT NULL
 GROUP BY BusinessEntityID),
PriorDept (BusinessEntityID, DeptName)
AS
(SELECT EDH.BusinessEntityID, Name
  FROM HumanResources.EmployeeDepartmentHistory EDH
    JOIN HumanResources.Department
      ON EDH.DepartmentID = Department.DepartmentID
    JOIN LastXfer
      ON EDH.BusinessEntityID = LastXfer.BusinessEntityID
    AND EDH.EndDate = LastXfer.LastEndDate
  )
SELECT Person.BusinessEntityID, FirstName, LastName, Name AS DeptName,
       PriorDept.DeptName AS PriorDeptName
  FROM Person.Person
    JOIN HumanResources.EmployeeDepartmentHistory EDH
      ON Person.BusinessEntityID = EDH.BusinessEntityID
      AND EDH.EndDate IS NULL
    JOIN HumanResources.Department
      ON EDH.DepartmentID = Department.DepartmentID
    LEFT JOIN PriorDept
      ON Person.BusinessEntityID = PriorDept.BusinessEntityID;
```

Using LAST_VALUE, we can use a single CTE, as in **Listing 47** (included in the materials for this session as EmployeeWithPriorDept-LastValue.SQL). The CTE uses LAST_VALUE to get the name of the last department an employee left. The CTE requires DISTINCT because it provides one record for each previous assignment for each employee; DISTINCT reduces that to one record. **Figure 37** shows partial results.

Listing 47. With LAST_VALUE, you can get the name of the employee's last prior department with a single CTE.

WITH PriorDept (BusinessEntityID, DeptName)

AS
(SELECT DISTINCT BusinessEntityID,
LAST_VALUE(Name) OVER
(PARTITION BY BusinessEntityID ORDER BY EndDate
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
FROM HumanResources.EmployeeDepartmentHistory EDH
JOIN HumanResources.Department
ON EDH.DepartmentID = Department.DepartmentID
WHERE EndDate IS NOT NULL)
<pre>SELECT Person.BusinessEntityID, FirstName, LastName, Name AS DeptName, PriorDept.DeptName AS PriorDeptName FROM Person.Person JOIN HumanResources.EmployeeDepartmentHistory EDH ON Person.BusinessEntityID = EDH.BusinessEntityID AND EDH.EndDate IS NULL JOIN HumanResources.Department ON EDH.DepartmentID = Department.DepartmentID LEFT JOIN PriorDept CN DepartmentE = Department_DepartmentID LEFT JOIN PriorDept</pre>
ON Person.BusinessEntityID = PriorDept.BusinessEntityID;

BusinessEntityID	FirstName	LastName	DeptName	PriorDeptName
1	Ken	Sánchez	Executive	NULL
2	Terri	Duffy	Engineering	NULL
3	Roberto	Tamburello	Engineering	NULL
4	Rob	Walters	Tool Design	Engineering
5	Gail	Erickson	Engineering	NULL
6	Jossef	Goldberg	Engineering	NULL
7	Dylan	Miller	Research and Development	NULL
8	Diane	Margheim	Research and Development	NULL
9	Gigi	Matthew	Research and Development	NULL
10	Michael	Raheem	Research and Development	NULL
11	Ovidiu	Cracium	Tool Design	NULL
12	Thierry	D'Hers	Tool Design	NULL
13	Janice	Galvin	Tool Design	NULL
14	Michael	Sullivan	Engineering	NULL
15	Sharon	Salavaria	Engineering	NULL

Figure 37. LAST_VALUE makes it easier to find the name of an employee's last previous department.

Showing distribution of records

The analytical function group also 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 earlier in this paper, is to look at the results. The query in **Listing 48** (RankAndDistribution.SQL in the materials for this session) computes

sales by salesperson by year, and then applies a series of analytics to the data. Partial results are shown in **Figure 38**.

Listing 48. 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;
```

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	0.1111111111111111	2	10	0.20
283	2011	599987.9444	0.3	0.22222222222222222	3	10	0.30
280	2011	648485.5862	0.4	0.333333333333333333	4	10	0.40
275	2011	875823.8318	0.5	0.444444444444444	5	10	0.50
281	2011	967597.2899	0.6	0.5555555555555556	6	10	0.60
276	2011	1149715.3253	0.7	0.666666666666666	7	10	0.70
282	2011	1175007.4753	0.8	0.7777777777777778	8	10	0.80
277	2011	1311627.2918	0.9	0.8888888888888888	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 38. CUME_DIST and PERCENT_RANK give similar but not identical results.

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 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).

One thing this example doesn't show is what happens when there are ties in the data. You should get a hint, though, from the fact that I used RANK (rather than RECORD_NUMBER) when computing the equivalent of CUME_DIST. Both CUME_DIST and PERCENT_RANK assign the same result to records with the same sort value. An updated version of the query in **Listing 12** demonstrates. The query in **Listing 49** (EmployeeRankByDeptWithDist.SQL in the materials for this session) ranks employees in each department by how long they've been working there. As you can see in the partial results in **Figure 39**, when multiple employees have the same start date, those employees share the same result both for CUME_DIST and for PERCENT_RANK.

Listing 49. 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) AS CumeDist,
PERCENT_RANK() OVER
(PARTITION BY Department.DepartmentID ORDER BY StartDate) AS PctRank
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
```

FirstName	LastName	StartDate	Name	EmployeeRank	CumeDist	PctRank
Ovidiu	Cracium	2010-12-05	Tool Design	3	0.75	0.666666666666666
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	0.1111111111111111	0.0588235294117647
Michael	Blythe	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Linda	Mitchell	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Jillian	Carson	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Garrett	Vargas	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Tsvi	Reiter	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Pamela	Ansman	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Shu	lto	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
José	Saraiva	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
David	Campbell	2011-05-31	Sales	3	0.6111111111111111	0.117647058823529
Amy	Alberts	2012-04-16	Sales	12	0.666666666666666	0.647058823529412
Jae	Pak	2012-05-30	Sales	13	0.7777777777777778	0.705882352941177

Figure 39. Records with the same value for the ordering expression are assigned the same result by both CUME_DIST and PERCENT_RANK.

This query also helps to explain 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 or before 31-May-2011; 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.

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."

You can also consider PERCENT_RANK as the percentile into which the record falls (divided by 100). 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/percentile values we're used to dealing with.

One way you might use these functions is to eliminate outliers from a calculation. For example, you might want to get a list of those products whose sales in a given year were in the middle 50%, that is, between the 25th and 75th percentiles. **Listing 50** (Middle50PctInSales.SQL in the materials for this session) shows how to do that using PERCENT_RANK. It uses two CTEs. The first computes yearly total sales for each product. The second uses PERCENT_RANK to rank the sales for each year. The main query then simply keeps those records whose rank falls between 0.25 and 0.75, and adds some more information about each product. **Figure 40** shows partial results, ordered from lowest to highest sales by year.

Listing 50. PERCENT_RANK makes it possible to keep only the middle range of values.

```
WITH csrProductSales (ProductID, nYear, TotalSales)
AS
(SELECT ProductID, YEAR(OrderDate), SUM(LineTotal)
 FROM [Sales]. [SalesOrderHeader] SOH
    JOIN [Sales]. [SalesOrderDetail] SOD
      ON SOH.SalesOrderID = SOD.SalesOrderID
 GROUP BY ProductID, YEAR(OrderDate)),
csrRankedProductSales (ProductID, nYear, TotalSales, PctRank)
AS
(SELECT ProductID, nYear, TotalSales,
       PERCENT_RANK() OVER (PARTITION BY nYear ORDER BY TotalSales)
  FROM csrProductSales)
SELECT Product.ProductID, Name, ProductNumber, nYear, TotalSales
  FROM csrRankedProductSales
    JOIN Production.Product
      ON csrRankedProductSales.ProductID = Product.ProductID
 WHERE PctRank BETWEEN 0.25 and 0.75
 ORDER BY nYear, TotalSales;
```

ProductID	Name	ProductNumber	nYear	TotalSales
715	Long-Sleeve Logo Jersey, L	LJ-0192-L	2011	15594.637060
733	ML Road Frame - Red, 52	FR-R72R-52	2011	19629.390000
722	LL Road Frame - Black, 58	FR-R38B-58	2011	20001.049600
726	LL Road Frame - Red, 48	FR-R38R-48	2011	20968.954800
730	LL Road Frame - Red, 62	FR-R38R-62	2011	21888.645800
769	Road-650 Black, 48	BK-R50B-48	2011	32158.515800
738	LL Road Frame - Black, 52	FR-R38B-52	2011	32323.124800
759	Road-650 Red, 58	BK-R50R-58	2011	33137.253400
729	LL Road Frame - Red, 60	FR-R38R-60	2011	33292.814200
725	LL Road Frame - Red, 44	FR-R38R-44	2011	34028.567000
767	Road-650 Black, 62	BK-R50B-62	2011	41106.972700
757	Road-450 Red, 48	BK-R68R-48	2011	50738.052000
747	HL Mountain Frame - Black, 38	FR-M94B-38	2011	52173.413900
732	ML Road Frame - Red, 48	FR-R72R-48	2011	52464.006000
742	HL Mountain Frame - Silver, 46	FR-M94S-46	2011	53472.022600

Figure 40. Only those products whose sales fell between the 25% and 75% percentile for the year are included here.

The next section of this document shows another way to filter based on percentile information.

Searching by percentile

The last two analytical functions, PERCENT_CONT and PERCENT_DISC, let you find the cutoff 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 51**; the syntax for PERCENTILE_CONT is identical except, of course, for the function name.

Listing 51. 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 another 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 52** (TenurePercentile.SQL in the materials for this session) 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 41** shows partial results.

```
Listing 52. 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;
```

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 41. Because PERCENTILE_CONT interpolates, the values it returns may not be in the original data. PERCENTILE_DISC always returns an actual data value. The difference is particularly striking in the data for the Executive department, which has only two employees.

These functions also let you find all the records above or below a certain percentile. For example, suppose you want a list of the customers in the top 10% of spending each month. That is, for each month, find the 90th percentile of customer spending and get a list of the customers who spent that much or more. There are three parts to the solution to this problem. First, compute customer spending by month. Second, find the 90th percentile of spending for each month. Finally, check the individual customer totals for each month against the 90th percentile value for that month. **Listing 53** (included in the materials for this session as CustomersAbove90thPercentile.SQL) shows the code. The query reflects the three tasks. The first CTE computes customer totals by month. The second CTE finds the cutoff for the 90th percentile for each month, using the computed totals. Finally, the main query joins the two CTE results, matching them by month and year, and keeps only those where the customer's total is at least the cutoff amount.

Listing 53. PERCENTILE_CONT lets us find customers whose purchases were in the 90th percentile or above for each month.

```
WITH csrSalesByCustomer (CustomerID, nMonth, nYear, TotalSales)
AS
(SELECT CustomerID, MONTH(OrderDate), YEAR(OrderDate), SUM(SubTotal)
FROM [Sales].[SalesOrderHeader]
GROUP BY CustomerID, MONTH(OrderDate), Year(OrderDate)),
csrNinetiethPctile (nMonth, nYear, Cutoff)
AS
(SELECT DISTINCT nMonth, nYear,
        PERCENTILE_CONT(.9) WITHIN GROUP (ORDER BY TotalSales)
        OVER (PARTITION BY nMonth, nYear )
FROM csrSalesByCustomer)
SELECT CustomerID, SBC.nMonth, SBC.nYear, SBC.TotalSales, NPtile. Cutoff
FROM csrSalesByCustomer SBC
        JOIN csrNinetiethPctile NPtile
```

```
ON SBC.nMonth = NPtile.nMonth
AND SBC.nYear = NPtile.nYear
WHERE SBC.TotalSales >= NPtile.Cutoff
ORDER BY nYear, nMonth, TotalSales DESC;
```

Figure 42 shows partial results; the surprising data for June, 2011 is because every customer who made a purchase that month spent the same amount. (Sounds like someone populating these sample tables got a little lazy.)

CustomerID	nMonth	nYear	TotalSales	Cutoff
29497	5	2011	42813.4333	33743.19188
29958	5	2011	39373.781	33743.19188
29992	5	2011	38510.8973	33743.19188
29614	5	2011	35944.1562	33743.19188
29646	5	2011	33997.3702	33743.19188
11611	6	2011	3578.27	3578.27
13260	6	2011	3578.27	3578.27
13591	6	2011	3578.27	3578.27
16515	6	2011	3578.27	3578.27
16518	6	2011	3578.27	3578.27
16521	6	2011	3578.27	3578.27
16524	6	2011	3578.27	3578.27
27578	6	2011	3578.27	3578.27
27616	6	2011	3578.27	3578.27
27645	6	2011	3578.27	3578.27

Figure 42. Only customers in the top 10% of sales for the month are included in this result.

In fact, you can use a query analogous to this one to find the products in the middle 50% of sales for each year. That is, you can rewrite the example in **Listing 50** using PERCENTILE_CONT; Middle50PercentInSales-Percentile.SQL in the materials for this session does just that.

You might also use these to build a table of percentiles for a standardized test, or to crunch data for political discussions about income and taxation.

OVER and out

The more time I spend with the functions that work with OVER, the more amazed I am at the number of problems you can solve with them. The ability to apply these functions to groups within a query and to narrow the set of records they use makes them extremely powerful. In preparing this session, I spent many hours trying different variations of functions and clauses to fully understand what OVER offers. I hope this paper gives you a jumpstart on that process, but I still recommend trying lots of variations on your own database.