



SQL Server 2012 Window Function Basics

05 March 2013

by *Robert Sheldon*

Av rating: ★★★★★

Total votes: 30

Total comments: 3

✉ [send to a friend](#)

🖨 [printer friendly version](#)

For some time, Microsoft had a few window functions, but not the full set specified in the SQL 2003 standard. Now, in SQL Server 2012 we have the whole range, and extremely useful they are too. There's no longer an excuse to avoid them, particularly now you have Rob's gentle introduction.

When Microsoft released SQL Server 2005, they included support for window functions, built-in T-SQL functions that can be applied to a result set's partitioned rows—*windows*—in order to rank or aggregate data in each partition. However, support for window functions was fairly limited in SQL Server 2005, and those limitations carried into 2008. But SQL Server 2012 has pushed through those constraints by expanding the capabilities of existing window functions and by adding new functions that support windowing.

SQL Server 2012 now includes three types of window functions: ranking, aggregate, and analytic. Ranking functions return a ranking value for each row in a partition. Aggregate functions perform a calculation on a column's values within a partition, such as finding the total or average of those values. Ranking functions first appeared in SQL Server 2005 with the advent of window function support. Aggregate functions have been around forever.

Analytic functions are new to SQL Server 2012. An analytic function computes an aggregate value based on the values in a column within a partition. However, analytic functions go beyond simple aggregate ones by being able to take such actions as returning the first or last value in an ordered partitioned set, retrieving the previous or next value in that set, or calculating percentages based on the set's cumulative values.

NOTE: For more details about analytic functions, see the topic "[Analytic Functions \(Transact-SQL\)](#)" in SQL Server Books Online. Also note that SQL Server 2012 supports another window function, **NEXT VALUE FOR**, which is not considered a rank, aggregate, or analytic function. The function generates a sequence number based on a specified sequence object. A discussion of the **NEXT VALUE FOR** function is beyond the scope of this article, but you can find details about the function in the topic "[NEXT VALUE FOR \(Transact-SQL\)](#)" in SQL Server Books Online.

SQL Categories

[SQL Home](#)

[Learn SQL Server](#)

[SQL Training](#)

[Database Administration](#)

[T-SQL Programming](#)

[Performance](#)

[Backup and Recovery](#)

[SQL Tools](#)

[Editor's Corner](#)

[SSIS](#)

[Reporting Services](#)

Custom RSS feeds

- ☒ [SQL](#)
- ☒ [.NET](#)
- ☒ [SysAdmin](#)
- ☒ [Opinion](#)
- ☒ [Books](#)
- ☒ [Blogs](#)

If you update your feed, please remember to tell your RSS reader the new URL

[Click here for advanced RSS options](#)

[Get my feed](#)

The key to understanding SQL Server window functions is in the **OVER** clause, which can be defined on any window function used in a query's select list. The clause determines how to partition and sort a result set in order to apply the window function. In fact, you can think of a window function as one that supports the **OVER** clause.

To use the **OVER** clause, you first call the window function, followed by the **OVER** keyword. You then specify one or more of the supported subclauses (enclosed in parentheses) to qualify your partitioning and ordering strategy. The **OVER** clause supports three subclauses, as shown in the following syntax:

```
<window function> OVER
(
  [ PARTITION BY <expression> [, ... n] ]
  [ ORDER BY <expression> [ASC|DESC] [, ... n] ]
  [ ROWS|RANGE <window frame> ]
)
<=" " lang="EN-US">
```

The **PARTITION BY** subclause partitions the result set based on one or more columns or expressions. In most cases, you'll probably use a single column. All window functions support the **PARTITION BY** subclause, but it is optional in each case. When the subclause is not specified, the entire result set is treated as a single partition.

The **ORDER BY** subclause sorts one or more columns or expressions within the partition. The **ORDER BY** clause applies strictly within the context of the partition. All ranking functions and most aggregate functions can use the **ORDER BY** subclause. For ranking functions, the subclause is required. For aggregate functions, the subclause is optional. Only certain analytic functions can use the **ORDER BY** subclause. However, those that can must use it.

NOTE: Most, but not all, aggregate functions support the **OVER** clause. If they support the **OVER** clause, then they support the **ORDER BY** subclause. Unfortunately, Microsoft documentation is a bit inconsistent on which ones support the **OVER** clause and which do not. The easiest way to find out is to try them out, but know that the most common aggregate functions—such as **SUM**, **AVG**, **MIN**, **MAX**, and **COUNT**—do support the clause.

The **ROWS/RANGE** subclause further defines how the data is displayed within the partition. Ranking functions cannot use the **ROWS/RANGE** subclause. Aggregate functions can use the subclause, but it is optional. Only two analytic functions—**FIRST_VALUE** and **LAST_VALUE**—can use the subclause, and it is optional for them as well.

The **ORDER BY** subclause must be included if the **ROWS/RANGE** subclause is specified. Also, if a window function supports the **ROWS/RANGE** subclause but the subclause is not defined, the function behaves as if the subclause is specified at its default value, which affects the behavior of the **ORDER BY** subclause. More on this later in the article.

The details of how the subclauses fit together can seem a bit convoluted. The best way to understand how the various components work is to see them in action. The rest of the article uses examples to demonstrate these concepts. The examples are based on the table shown in the following T-SQL script:





```
USE AdventureWorks2012;
GO

IF OBJECT_ID('RegionalSales', 'U') IS NOT NULL
DROP TABLE RegionalSales;
GO
```




PowerPoint Presentation Burnout
Phil's dread of Powerpoint sales presentations is already known to his readers, but we've never before heard the... [Read more...](#)

RECENT BLOG POSTS:


-  [Who's afraid of the big bad data type](#)
-  [Azure Explorer: Cause for Cerebration.](#)
-  [SQL Server JSON to Table and Table to JSON](#)
-  [View the blog](#)

Top Rated


Highway to Database Recovery

 Discover the best backup and recovery articles on Simple-Talk, all in one place. [Read more...](#)


Precision Indexing: Basics of Selective XML Indexes in SQL Server 2012

 Seldom has a SQL Server Service pack had such an effect on database development as when SQL Server 2012... [Read more...](#)

Working with Continuous Integration in a BI Environment Using Red Gate Tools with TFS

 Continuous integration is becoming increasingly popular for database development, and when we heard of ... [Read more...](#)

SQL Source Control: The Development Story

 Often, there is a huge difference between software being easy to use, and easy to develop. When your... [Read more...](#)

The PoSh DBA: Solutions using PowerShell and SQL Server

```

CREATE TABLE RegionalSales
(
    SalesID INT NOT NULL IDENTITY PRIMARY KEY,
    SalesGroup NVARCHAR(30) NOT NULL,
    Country NVARCHAR(30) NOT NULL,
    AnnualSales INT NOT NULL
);
GO

INSERT INTO RegionalSales
(SalesGroup, Country, AnnualSales)
VALUES
('North America', 'United States', 22000),
('North America', 'Canada', 32000),
('North America', 'Mexico', 28000),
('Europe', 'France', 19000),
('Europe', 'Germany', 22000),
('Europe', 'Italy', 18000),
('Europe', 'Greece', 16000),
('Europe', 'Spain', 16000),
('Europe', 'United Kingdom', 32000),
('Pacific', 'Australia', 18000),
('Pacific', 'China', 28000),
('Pacific', 'Singapore', 21000),
('Pacific', 'New Zealand', 18000),
('Pacific', 'Thailand', 17000),
('Pacific', 'Malaysia', 19000),
('Pacific', 'Japan', 22000);
GO

```

The script creates a simple table and populates the table with sales data broken into sales groups and countries. As for what the sales represent, use your imagination. They can be anything from hamburgers to paper airplanes to relational database management systems.

Ranking Functions

Nothing has changed with the ranking functions in SQL Server 2012, and you might already be well versed in how they work. I'm including a brief overview here in order to be complete, but you can skip ahead if necessary, though you might find this section to be a handy refresher.

The first example we'll look at uses all four of the ranking functions to rank our sample sales data based on the amount of sales:

```

SELECT
    SalesGroup,
    Country,
    AnnualSales,
    ROW_NUMBER() OVER(ORDER BY AnnualSales DESC) AS RowNumber,
    RANK() OVER(ORDER BY AnnualSales DESC) AS BasicRank,
    DENSE_RANK() OVER(ORDER BY AnnualSales DESC) AS DenseRank,

```

📖 PowerShell is worth using when it is the quickest way to providing a solution. For the DBA, it is much... [Read more...](#)

Most Viewed

Beginning SQL Server 2005 Reporting Services Part 1

📖 Steve Joubert begins an in-depth tour of SQL Server 2005 Reporting Services with a step-by-step guide... [Read more...](#)

Ten Common Database Design Mistakes

📖 If database design is done right, then the development, deployment and subsequent performance in... [Read more...](#)

SQL Server Index Basics

📖 Given the fundamental importance of indexes in databases, it always comes as a surprise how often the... [Read more...](#)

Reading and Writing Files in SQL Server using T-SQL

📖 SQL Server provides several "standard" techniques by which to read and write to files but, just... [Read more...](#)

Concatenating Row Values in Transact-SQL

📖 It is an interesting problem in Transact SQL, for which there are a number of solutions and... [Read more...](#)

Why Join

Over 400,000 Microsoft professionals subscribe to

```
NTILE(3) OVER(ORDER BY AnnualSales DESC) AS NTileRank
FROM
    RegionalSales;
```

the Simple-Talk technical journal. Join today, it's fast, simple, free and secure.

Join Simple-Talk!

Notice that for each function, I include an **ORDER BY** subclause that sorts the **AnnualSales** column in descending order. Because I have not included a **PARTITION BY** subclause, the result set is treated as a single partition. Each ranking function ranks the data, based on the sorted **AnnualSales** values, in different ways. The following table shows the results returned by the **SELECT** statement.

SalesGroup	Country	AnnualSales	Row Number	BasicRank	DenseRank	NTileRank
North America	Canada	32000	1	1	1	1
Europe	United Kingdom	32000	2	1	1	1
North America	Mexico	28000	3	3	2	1
Pacific	China	28000	4	3	2	1
North America	United States	22000	5	5	3	1
Pacific	Japan	22000	6	5	3	1
Europe	Germany	22000	7	5	3	2
Pacific	Singapore	21000	8	8	4	2
Pacific	Malaysia	19000	9	9	5	2
Europe	France	19000	10	9	5	2
Pacific	Australia	18000	11	11	6	2
Europe	Italy	18000	12	11	6	3
Pacific	New Zealand	18000	13	11	6	3
Pacific	Thailand	17000	14	14	7	3
Europe	Greece	16000	15	15	8	3
Europe	Spain	16000	16	15	8	3

The **ROW_NUMBER** function simply numbers each row in the partition. Because the statement returns 16 rows and there is only one partition, the rows are numbered 1 through 16.

The **RANK** function also numbers each row consecutively, based on the sorted **AnnualSales** values, but also takes into account duplicate values by assigning those values the same rank and then skipping the rank value that would have been assigned to the second duplicate. For example, the first two rows are ranked 1, so the third row is ranked 3 because the 2 has been skipped.

The **DENSE_RANK** function takes a slightly different approach. It accounts for duplicates but doesn't skip ranking values. Consequently, the first two rows receive a ranking value of 1, as with **RANK**, but the third row receives a value of 2.

The **NTILE** function is a different animal all together. If you refer back to the **SELECT** statement, you'll notice that I passed a value of 3 in as an argument to the function. As a result, the function divides the partition into three groups. That division is based on the total number of rows divided by the number specified in the function's argument. In this case, the 16 rows in the result set are divided into one group of six rows and two groups of five rows. The function assigns a 1 to each row in the first group, a 2 to each row in the second group, and a 3 to each

row in the third group.

You might have also noticed in the example above that the result set is sorted based on the **AnnualSales** values (in descending order). However, that occurs only because we did not specify an **ORDER BY** clause on the outer statement itself. As you'll recall, the **ORDER BY** subclause specified in the **OVER** clause is specific to the partition, so SQL Server defaults to a sort order based on the subclause. But the subclause is specific to the function associated with the **OVER** clause. That means, we can override the subclause's sorting in the result set, without impacting the functions' results.

Let's look at an example to demonstrate what that looks like. The following **SELECT** statement is the same as the preceding one only now it includes an **ORDER BY** clause that sorts first by the **SalesGroup** column and then by the **Country** column:

```
SELECT
    SalesGroup,
    Country,
    AnnualSales,
    ROW_NUMBER() OVER(ORDER BY AnnualSales DESC) AS RowNumber,
    RANK() OVER(ORDER BY AnnualSales DESC) AS BasicRank,
    DENSE_RANK() OVER(ORDER BY AnnualSales DESC) AS DenseRank,
    NTILE(3) OVER(ORDER BY AnnualSales DESC) AS NTileRank
FROM
    RegionalSales
ORDER BY
    SalesGroup, Country;
```

As you can see, nothing has changed but the addition of the **ORDER BY** clause. However, as the following table shows, the results in the ranking columns are in a different order:

SalesGroup	Country	AnnualSales	Row Number	BasicRank	DenseRank	NTileRank
Europe	France	19000	10	9	5	2
Europe	Germany	22000	7	5	3	2
Europe	Greece	16000	15	15	8	3
Europe	Italy	18000	12	11	6	3
Europe	Spain	16000	16	15	8	3
Europe	United Kingdom	32000	2	1	1	1
North America	Canada	32000	1	1	1	1
North America	Mexico	28000	3	3	2	1
North America	United States	22000	5	5	3	1
Pacific	Australia	18000	11	11	6	2
Pacific	China	28000	4	3	2	1
Pacific	Japan	22000	6	5	3	1
Pacific	Malaysia	19000	9	9	5	2
Pacific	New Zealand	18000	13	11	6	3

Pacific	Singapore	21000	8	8	4	2
Pacific	Thailand	17000	14	14	7	3

The data in this result set is the same as the one in the preceding example. It's just messier looking. For instance, the **RowNumber** value for France is 10, the **BasicRank** value is 9, the **DenseRank** value is 5, and the **NTileRank** value is 2, just as they were in the preceding example. However, because we specifically sorted our entire result set, the ranking columns themselves are no longer in order, but they still base their data on the sorted **AnnualSales** column, as defined in the **ORDER BY** subclause of the **OVER** clause. What this points to is that the **ORDER BY** subclause is specific to the window function associated with the **OVER** clause.

Now let's return to our original example, only this time we'll add a **PARTITION BY** subclause to each rank:

```
SELECT
    SalesGroup,
    Country,
    AnnualSales,
    ROW_NUMBER() OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS RowNumber,
    RANK() OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS BasicRank,
    DENSE_RANK() OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS DenseRank,
    NTILE(3) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS NTileRank
FROM
    RegionalSales;
```

For each ranking function, we partition the results by the **SalesGroup** values, which breaks our result set into three distinct groups. The following table shows the results now returned by the **SELECT** statement:

SalesGroup	Country	AnnualSales	Row Number	BasicRank	DenseRank	NTileRank
Europe	United Kingdom	32000	1	1	1	1
Europe	Germany	22000	2	2	2	1
Europe	France	19000	3	3	3	2
Europe	Italy	18000	4	4	4	2
Europe	Greece	16000	5	5	5	3
Europe	Spain	16000	6	5	5	3
North America	Canada	32000	1	1	1	1
North America	Mexico	28000	2	2	2	2
North America	United States	22000	3	3	3	3
Pacific	China	28000	1	1	1	1
Pacific	Japan	22000	2	2	2	1
Pacific	Singapore	21000	3	3	3	1
Pacific	Malaysia	19000	4	4	4	2

Pacific	Australia	18000	5	5	5	2
Pacific	New Zealand	18000	6	5	5	3
Pacific	Thailand	17000	7	7	6	3

As you can see, the ranking functions now each apply to the individual partitions, which are based on the sales group. For example, the Europe sales group includes six rows, one for each country. As a result, the **ROW_NUMBER** function ranks the rows 1 through 6. However, because the last two values in that group are duplicates, the **RANK** and **DENSE_RANK** functions assign the first four rows 1 through 4 and assign a 5 to the last two rows. The **RANK** function doesn't skip any numbers because the duplicates come at the end of the sorted **AnnualSales** column. And when the **NTILE** function breaks the Europe partition into three groups, each group contains only three rows.

Now let's see what happens if we add an **ORDER BY** clause to the statement itself, as shown in the following example:

```
SELECT
    SalesGroup,
    Country,
    AnnualSales,
    ROW_NUMBER() OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS RowNumber,
    RANK() OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS BasicRank,
    DENSE_RANK() OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS DenseRank,
    NTILE(3) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS NTileRank
FROM
    RegionalSales
ORDER BY
    SalesGroup, Country;
```

Once again, I've sorted the result set first by **SalesGroup** and then by **Country**. As the following table demonstrates, the ranking values are now displayed in a different order:

SalesGroup	Country	AnnualSales	Row Number	BasicRank	DenseRank	NTileRank
Europe	France	19000	3	3	3	2
Europe	Germany	22000	2	2	2	1
Europe	Greece	16000	5	5	5	3
Europe	Italy	18000	4	4	4	2
Europe	Spain	16000	6	5	5	3
Europe	United Kingdom	32000	1	1	1	1
North America	Canada	32000	1	1	1	1
North America	Mexico	28000	2	2	2	2
North America	United States	22000	3	3	3	3

Pacific	Australia	18000	5	5	5	2
Pacific	China	28000	1	1	1	1
Pacific	Japan	22000	2	2	2	1
Pacific	Malaysia	19000	4	4	4	2
Pacific	New Zealand	18000	6	5	5	3
Pacific	Singapore	21000	3	3	3	1
Pacific	Thailand	17000	7	7	6	3

Although the result set itself has been sorted, the ranking data is still based on the defined partition and sort order within the partition. Each country receives the same ranking values as they did in the earlier example, only now they're in a different order.

Aggregate Functions

Aggregate functions work a bit differently from ranking functions, not only because of the functions themselves, but also in terms of how the subclauses work. For example, aggregate functions don't require an **ORDER BY** subclause and can include a **ROWS/RANGE** subclause. But first, let's look at aggregate functions that use only the **PARTITION BY** subclause. In the following example, the **SELECT** statement uses several aggregate functions to calculate **AnnualSales** values for each partition:

```
SELECT
  SalesGroup,
  Country,
  AnnualSales,
  COUNT(AnnualSales) OVER(PARTITION BY SalesGroup) AS CountryCount,
  SUM(AnnualSales) OVER(PARTITION BY SalesGroup) AS TotalSales,
  AVG(AnnualSales) OVER(PARTITION BY SalesGroup) AS AverageSales
FROM
  RegionalSales
ORDER BY
  SalesGroup, AnnualSales DESC;
```

As before the statement partitions the data by sales group. In addition, for each function, I pass in the **AnnualSales** column as the argument because I want to perform my calculations on that column. The **SELECT** statement returns the results shown in the following table:

SalesGroup	Country	AnnualSales	CountryCount	TotalSales	AverageSales
Europe	United Kingdom	32000	6	123000	20500
Europe	Germany	22000	6	123000	20500
Europe	France	19000	6	123000	20500
Europe	Italy	18000	6	123000	20500
Europe	Greece	16000	6	123000	20500
Europe	Spain	16000	6	123000	20500
North America	Canada	32000	3	82000	27333
North America	Mexico	28000	3	82000	27333
North America	United States	22000	3	82000	27333

Pacific	China	28000	7	143000	20428
Pacific	Japan	22000	7	143000	20428
Pacific	Singapore	21000	7	143000	20428
Pacific	Malaysia	19000	7	143000	20428
Pacific	Australia	18000	7	143000	20428
Pacific	New Zealand	18000	7	143000	20428
Pacific	Thailand	17000	7	143000	20428

Notice how the aggregate functions have been applied to each group. For example, the **COUNT** function returns a 6 for each row in the Europe group, the **SUM** function returns 123000 for each row in that group, and the **AVG** function returns 20500 for those rows.

If you wanted to get at the aggregate calculations only, you can simplify your statement in order to eliminate duplicate values, as shown in the following example:

```
SELECT DISTINCT
    SalesGroup,
    COUNT(AnnualSales) OVER(PARTITION BY SalesGroup) AS CountryCount,
    SUM(AnnualSales) OVER(PARTITION BY SalesGroup) AS TotalSales,
    AVG(AnnualSales) OVER(PARTITION BY SalesGroup) AS AverageSales
FROM
    RegionalSales
ORDER BY
    TotalSales DESC;
```

Now the **SELECT** statement returns only the data we need:

SalesGroup	CountryCount	TotalSales	AverageSales
Pacific	7	143000	20428
Europe	6	123000	20500
North America	3	82000	27333

So far, everything we've looked at to this point has existed in SQL Server since the 2005 release. However, SQL Server 2012 now supports the **ORDER BY** subclause for aggregate functions. For example, the following **SELECT** statement adds an **ORDER BY** subclause that sorts **AnnualSales** in descending order:

```
SELECT
    SalesGroup,
    Country,
    AnnualSales,
    COUNT(AnnualSales) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS CountryCount,
    SUM(AnnualSales) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS TotalSales,
    AVG(AnnualSales) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS AverageSales
```

```
FROM  
  RegionalSales;
```

The data is still partitioned by the **SalesGroup** column, only now we've added the **ORDER BY** subclause. However, as the following table shows, the statement's results might not be quite what you expect:

SalesGroup	Country	AnnualSales	CountryCount	TotalSales	AverageSales
Europe	United Kingdom	32000	1	32000	32000
Europe	Germany	22000	2	54000	27000
Europe	France	19000	3	73000	24333
Europe	Italy	18000	4	91000	22750
Europe	Greece	16000	6	123000	20500
Europe	Spain	16000	6	123000	20500
North America	Canada	32000	1	32000	32000
North America	Mexico	28000	2	60000	30000
North America	United States	22000	3	82000	27333
Pacific	China	28000	1	28000	28000
Pacific	Japan	22000	2	50000	25000
Pacific	Singapore	21000	3	71000	23666
Pacific	Malaysia	19000	4	90000	22500
Pacific	Australia	18000	6	126000	21000
Pacific	New Zealand	18000	6	126000	21000
Pacific	Thailand	17000	7	143000	20428

In fact, when you use the **ORDER BY** subclause with an aggregate function, the aggregated data changes with each row. The result set now shows moving averages and cumulative totals. For example, Germany shows a count of 2, a total of 54000, and an average of 27000. Because Germany is the second row in the partition, as determined by the **ORDER BY** subclause, the aggregated totals reflect only those two rows. The same thing goes for France. Because it comes in at number 3, the aggregated totals reflect only the first three rows. To complicate matters, duplicated rows are grouped together, as is the case with Greece and Spain.

There's a reason for this behavior. As you'll recall from earlier in the article, when a window function supports the **ROWS/RANGE** subclause but the subclause has not been specified, the function operates as if it has been specified with its default value. And that default setting impacts the **ORDER BY** subclause.

The default setting for the **ROWS/RANGE** subclause is **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**. This means that, for each row in the partition, the window function is applied to the current row and the preceding rows only. So aggregations don't operate on the entire set of values within the partition, but only on the value in the current row and the previous rows, as we saw in the example above.

NOTE: The main difference between a **ROWS** clause and a **RANGE** clause is in the way duplicate data is treated. **ROWS** treats duplicates as distinct values. **RANGE** treats them as a single entity, as the above result set indicates.

The way to get around the default behavior, of course, is to add a **ROWS/RANGE** subclause that overrides that behavior, as shown in the following example:

```
SELECT  
  SalesGroup,  
  Country,
```

```

AnnualSales,
COUNT(AnnualSales) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC
ROWS 2 PRECEDING) AS CountryCount,
SUM(AnnualSales) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC
ROWS 2 PRECEDING) AS TotalSales,
AVG(AnnualSales) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC
ROWS 2 PRECEDING) AS AverageSales
FROM
RegionalSales;

```

Notice that I've added the subclause **ROWS 2 PRECEDING** to each instance of the **OVER** clause. Now for each partition, the aggregate function applies only to the current row and the two preceding rows, as shown in the following results:

SalesGroup	Country	AnnualSales	CountryCount	TotalSales	AverageSales
Europe	United Kingdom	32000	1	32000	32000
Europe	Germany	22000	2	54000	27000
Europe	France	19000	3	73000	24333
Europe	Italy	18000	3	59000	19666
Europe	Greece	16000	3	53000	17666
Europe	Spain	16000	3	50000	16666
North America	Canada	32000	1	32000	32000
North America	Mexico	28000	2	60000	30000
North America	United States	22000	3	82000	27333
Pacific	China	28000	1	28000	28000
Pacific	Japan	22000	2	50000	25000
Pacific	Singapore	21000	3	71000	23666
Pacific	Malaysia	19000	3	62000	20666
Pacific	Australia	18000	3	58000	19333
Pacific	New Zealand	18000	3	55000	18333
Pacific	Thailand	17000	3	53000	17666

Because the **ROWS** subclause is included, each aggregated value never calculates more than three rows, so the **CountryCount** value will never exceed 3 and the total and average sales will never represent the total amounts within the group, unless that group has fewer than four rows. If you don't want to aggregate your partitioned data in this way—with totals and averages that don't reflect the entire partition—your best bet is to go with the **PARTITION BY** subclause only and not the other subclauses.

Analytic Functions

Analytic functions work much the same way as aggregate functions, except that the **ORDER BY** subclause is required for those functions that support the clause. In our next example, we'll try out the **FIRST_VALUE** and **LAST_VALUE** analytic functions. The **FIRST_VALUE** function retrieves the first value from a sorted list, and the

LAST_VALUE function retrieves the last value. In the following example, the **SELECT** statement uses both of these functions to calculate **AnnualSales** values in each sales group:

```
SELECT
    SalesGroup,
    Country,
    AnnualSales,
    FIRST_VALUE(AnnualSales) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS HighestSales,
    LAST_VALUE(AnnualSales) OVER(PARTITION BY SalesGroup
        ORDER BY AnnualSales DESC) AS LowestSales
FROM
    RegionalSales;
```

Once again, I use the **PARTITION BY** subclause to partition the result set by sales group. In addition, I use the **ORDER BY** subclause to specify that the **AnnualSales** values be sorted in descending order. The following table shows the results returned by the **SELECT** statement.

SalesGroup	Country	AnnualSales	HighestSales	LowestSales
Europe	United Kingdom	32000	32000	32000
Europe	Germany	22000	32000	22000
Europe	France	19000	32000	19000
Europe	Italy	18000	32000	18000
Europe	Greece	16000	32000	16000
Europe	Spain	16000	32000	16000
North America	Canada	32000	32000	32000
North America	Mexico	28000	32000	28000
North America	United States	22000	32000	22000
Pacific	China	28000	28000	28000
Pacific	Japan	22000	28000	22000
Pacific	Singapore	21000	28000	21000
Pacific	Malaysia	19000	28000	19000
Pacific	Australia	18000	28000	18000
Pacific	New Zealand	18000	28000	18000
Pacific	Thailand	17000	28000	17000

As to be expected, the result set is grouped by the **SalesGroup** column, with the **AnnualSales** values in each group sorted. The **HighestSales** column displays the first of the sorted values, and the **LowestSales** column displays the last of the sorted values. But these values are running totals because the **FIRST_VALUE** and **LAST_VALUE** functions support the **ROWS/RANGE** subclause, which impacts the **ORDER BY** operation. For example, the highest amount of sales for the France row is 32000 and the lowest amount is 19000. These calculations are based only on the first three rows in this partition as a result of the **ROWS/RANGE** default settings being applied.

As before, we can override the subclause's default behavior. Only this time, let's change it to incorporate all **AnnualSales** in each calculation, regardless of row:

```
SELECT
    SalesGroup,
    Country,
```

```

AnnualSales,
FIRST_VALUE(AnnualSales) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC) AS HighestSales,
LAST_VALUE(AnnualSales) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
AS LowestSales
FROM
RegionalSales;

```

Notice that our **ROWS** clause specifies unbounded preceding and following values, which means all values should be included in the calculations. The following table shows what the results look like:

SalesGroup	Country	AnnualSales	HighestSales	LowestSales
Europe	United Kingdom	32000	32000	16000
Europe	Germany	22000	32000	16000
Europe	France	19000	32000	16000
Europe	Italy	18000	32000	16000
Europe	Greece	16000	32000	16000
Europe	Spain	16000	32000	16000
North America	Canada	32000	32000	22000
North America	Mexico	28000	32000	22000
North America	United States	22000	32000	22000
Pacific	China	28000	28000	17000
Pacific	Japan	22000	28000	17000
Pacific	Singapore	21000	28000	17000
Pacific	Malaysia	19000	28000	17000
Pacific	Australia	18000	28000	17000
Pacific	New Zealand	18000	28000	17000
Pacific	Thailand	17000	28000	17000

As you can see, the **HighestSales** and **LowestSales** columns now display duplicate values for each row in a sales group, providing both the first and last value, respectively, in the sorted **AnnualSales** column. You can, of course, simplify your **SELECT** statement to retrieve only the necessary distinct values, as we did in an earlier example.

Now let's look at two other analytic functions: **LAG** and **LEAD**. The **LEAD** function retrieves a value from a row previous to the current one. The **LAG** function retrieves a value from a row after the current one. The following **SELECT** statement demonstrates how to use these functions:

```

SELECT
SalesGroup,
Country,
AnnualSales,
LAG(AnnualSales, 1) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC) AS PreviousSale,
LEAD(AnnualSales, 1) OVER(PARTITION BY SalesGroup
ORDER BY AnnualSales DESC) AS NextSale
FROM
RegionalSales;

```

First, notice that we pass a second argument into the functions, in this case, 1. The argument indicates that we go one row up or down to retrieve the value from the sorted **AnnualSales** values. The following table shows the results returned by the **SELECT** statement.

SalesGroup	Country	AnnualSales	PreviousSale	NextSale
Europe	United Kingdom	32000	NULL	22000
Europe	Germany	22000	32000	19000
Europe	France	19000	22000	18000
Europe	Italy	18000	19000	16000
Europe	Greece	16000	18000	16000
Europe	Spain	16000	16000	NULL
North America	Canada	32000	NULL	28000
North America	Mexico	28000	32000	22000
North America	United States	22000	28000	NULL
Pacific	China	28000	NULL	22000
Pacific	Japan	22000	28000	21000
Pacific	Singapore	21000	22000	19000
Pacific	Malaysia	19000	21000	18000
Pacific	Australia	18000	19000	18000
Pacific	New Zealand	18000	18000	17000
Pacific	Thailand	17000	18000	NULL

As the results show, each row displays **AnnualSales** values from the previous and next rows within each partition, unless it's a first or last row, in which case **NULL** is returned. For example, the United Kingdom row returns a **NULL** in the **PreviousSale** column because no rows precede this row. However, the **NextSale** column displays the amount 22000, the **AnnualSales** amount from the row that follows.

SQL Server supports four other analytic functions that mostly have to do with calculating percentages. It also supports additional ways to configure the **ROWS/RANGE** subclause. In fact, the subclause supports a number of variations. But the key to understanding the subclause is to become familiar with the **OVER** clause and how the subclauses work together to partition and present data. For a complete discussion of the clause, see the topic "[OVER Clause \(Transact-SQL\)](#)" in SQL Server Books Online. And be willing to try out window functions in different scenarios. They're the type of feature you need to mess with for a while before you become fully comfortable with using them.

This article has been viewed 9728 times.

Thank this author by sharing:     10



Author profile: [Robert Sheldon](#)

After being dropped 35 feet from a helicopter and spending the next year recovering, Robert Sheldon left the Colorado Rockies and emergency rescue work to pursue safer and less painful interests—thus his entry into the world of technology. He is now a technical consultant and the author of numerous books, articles, and training material related to Microsoft Windows, various relational database management systems, and business intelligence design and implementation. He has

also written news stories, feature articles, restaurant reviews, legal summaries, and the novel 'Dancing the River Lightly'. You can find more information at <http://www.rhsheldon.com>.

[Search for other articles by Robert Sheldon](#)

Rate this article: Avg rating: ★★☆☆☆ from a total of 30 votes.



Poor



OK



Good



Great



Must read

SUBMIT

Have Your Say

Do you have an opinion on this article? Then [add your comment below](#):

You must be logged in to post to this forum

[Click here to log in.](#)

Subject: window functions or OLAP functions

Posted by: AlexK (not signed in)

Posted on: Thursday, March 07, 2013 at 3:55 PM

Message:Hi Robert,

I think that these "window functions" you are writing about are in fact OLAP functions - they were implemented more than a decade ago by IBM and Oracle. I've used them long ago. Example here, written in 2001:

<http://www.ibm.com/developerworks/data/library/techarticle/lyle/0110lyle.html>

IMO calling OLAP functions "window functions" hides this fact. I understand that you are using standard Microsoft nomenclature...

Subject: window functions or OLAP functions

Posted by: Anonymous (not signed in)

Posted on: Monday, March 18, 2013 at 12:38 AM

Message:good to know that these called OLAP function, idea of DB2.

Subject: No, WINDOW is the ANSI Standard terminology

Posted by: Celko ([view profile](#))

Posted on: Monday, March 18, 2013 at 9:29 AM

Message:No, WINDOW is the ANSI Standard terminology, not OLAP. The full version of this feature set was done by IBM and Oracle standards

committee members working together. The WINDOW or OVER() can be factored out and put into a clause at the front of the SELECT.

The whole feature set is large and tricky, and there are some vendor extensions that are useful. The hard part is [ROW | RANGE] subclauses.

[About](#) | [Site map](#) | [Become an author](#) | [Newsletters](#) | [Contact us](#) | [Help](#)

redgate

[Privacy policy](#) | [Terms and conditions](#) | ©2005-2013 Red Gate Software