Module 13: Using Window Ranking, Offset, and Aggregate Functions

Contents:

Module Overview

Microsoft® SQL Server® implements support for SQL windowing operations, which means you can define a set of rows and apply several different functions against those rows. After you have learned how to work with windows and window functions, you might find that some types of queries that appeared to require complex manipulations of data (for example, self-joins, temporary tables, and other constructs) aren't needed to write your reports.

Objectives

After completing this module, you will be able to:

- Describe the benefits of using window functions.
- Restrict window functions to rows defined in an OVER clause, including partitions and frames.
- Write queries that use window functions to operate on a window of rows and return ranking, aggregation, and offset comparison results.

Lesson 1: Creating Windows with OVER

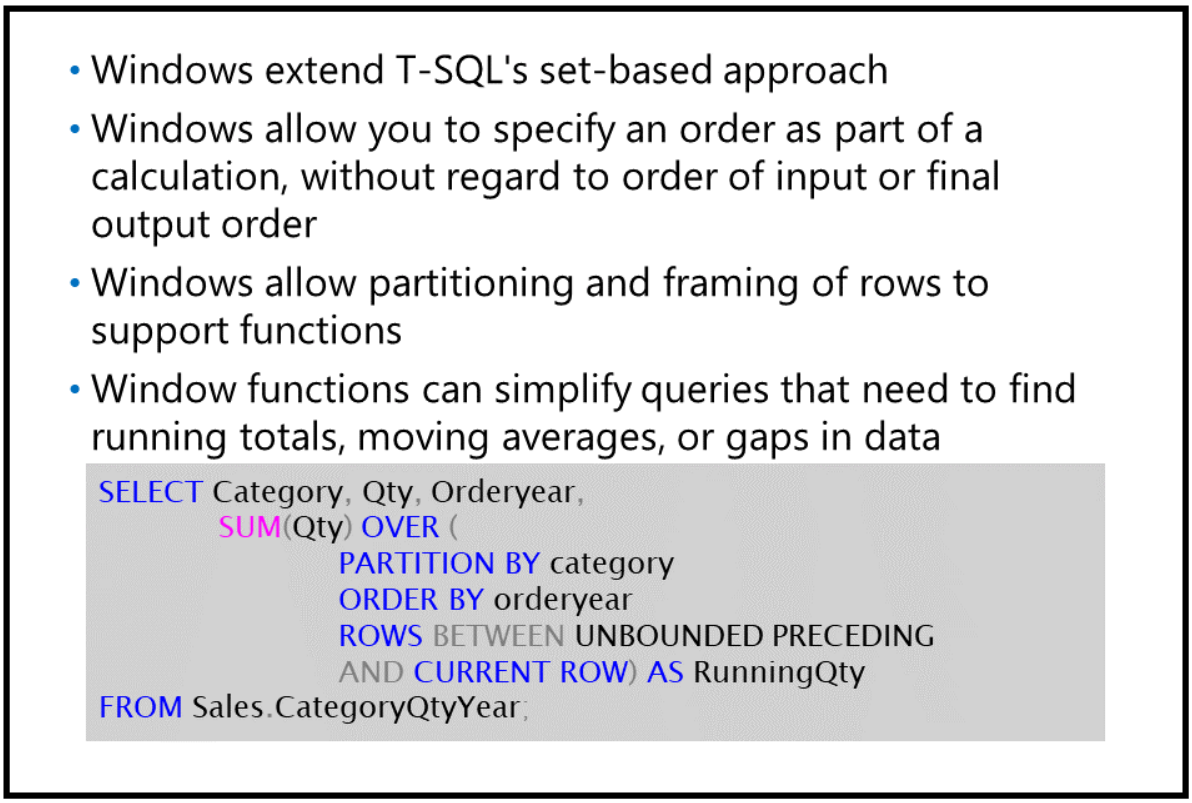
SQL Server provides a number of window functions, which perform calculations such as ranking, aggregations, and offset comparisons between rows. To use these functions, you will need to write queries that define windows, or sets, of rows. You will use the OVER clause and its related elements to define the sets for the window functions.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the T-SQL components used to define windows, and the relationships between them.
- Write queries that use the OVER clause, with partitioning, ordering, and framing to define windows.

SQL Windowing



SQL Server provides windows as a method for applying functions to sets of rows. There are many applications of this technique that solve common problems in writing T-SQL queries. For example, using windows allows the easy generation of row numbers in a result set and the calculation of running totals. Windows also provide an efficient way to compare values in one row with values in another without needing to join a table to itself using an inequality operator.

There are several core elements of writing queries that use windows:

1. Windows allow you to specify an order to rows that will be passed to a window function, without affecting the final order of the query output.
2. Windows include a partitioning feature, which enables you to specify that you want to restrict a function only to rows that have the same value as the current row.
3. Windows provide a framing option. It allows you to specify a further subset of rows within a window partition by setting upper and lower boundaries for the window frame, which presents rows to the window function.

Running Total Example

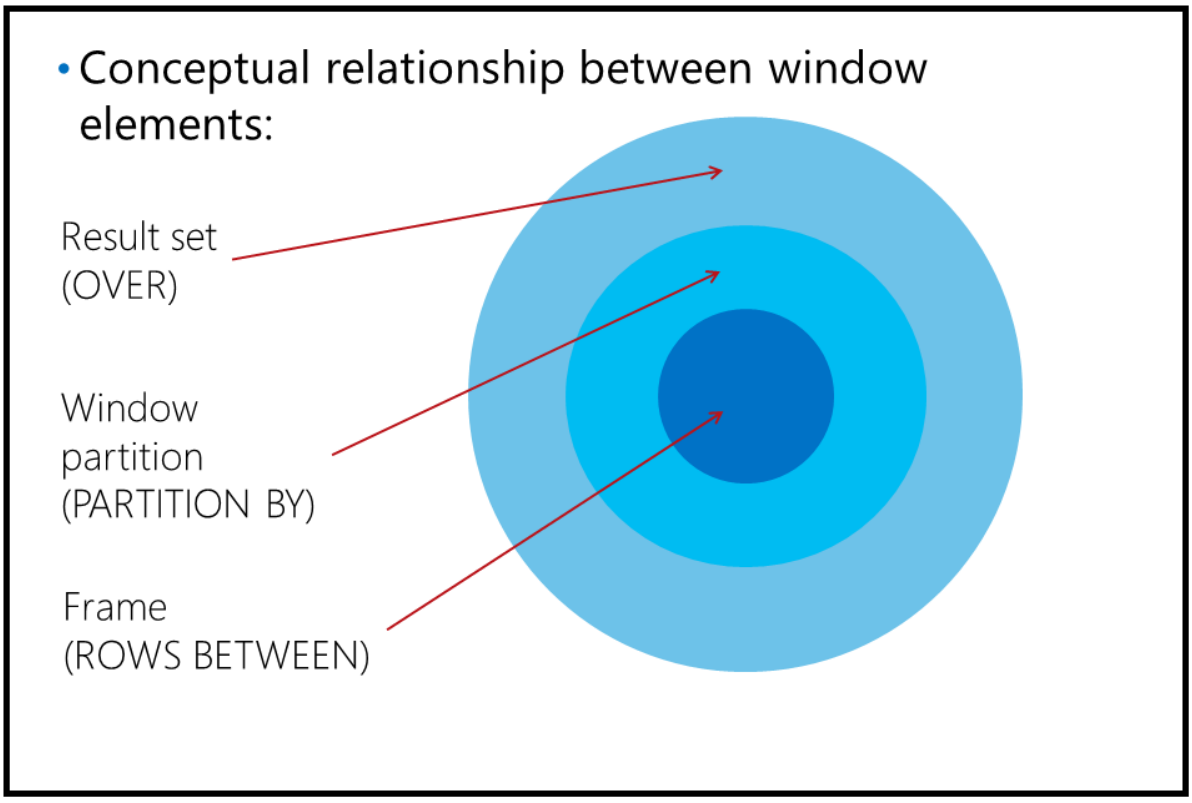
```
SELECT Category, Qty, Orderyear,
        SUM(Qty) OVER (PARTITION BY Category    ORDER BY Orderyear
                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningQty
FROM Sales.CategoryQtyYear;
```

The partial results:

```
Category        Qty   Orderyear  RunningQty
--------------- ----- ---------- -----------
Beverages       1842  2006       1842
Beverages       3996  2007       5838
Beverages       3694  2008       9532
Condiments      962   2006       962
Condiments      2895  2007       3857
Condiments      1441  2008       5298
Confections     1357  2006       1357
Confections     4137  2007       5494
Confections     2412  2008       7906
Dairy Products  2086  2006       2086
Dairy Products  4374  2007       6460
Dairy Products  2689  2008       9149
```

During the next few topics of this lesson, you will learn how to use these query elements.

Windowing Components



In order to use windows and window functions in T-SQL, you will always use one of the subclauses that create and manipulate windows—the OVER subclause. Additionally, you may need to create partitions with the PARTITION BY option, and even further restrict which rows are applied to a function with framing options. Therefore, understanding the relationship between these components is vital.

The general relationship can be expressed as a sequence, with one element further manipulating the rows output by the previous element:

1. The OVER clause determines the result set that will be used by the window function. An OVER clause with no partition defined is unrestricted. It returns all rows to the function.
2. A PARTITION BY clause, if present, restricts the results to those rows with the same value in the partitioned columns as the current row. For example, PARTITION BY custid restricts the window to rows with the same custid as the current row. PARTITION BY builds on the OVER clause and cannot be used without OVER. (An OVER clause without a window partition clause is considered one partition).
3. A ROW or RANGE clause creates a window frame within the window partition, which allows you to set a starting and ending boundary around the rows being operated on. A frame requires an ORDER BY subclause within the OVER clause.

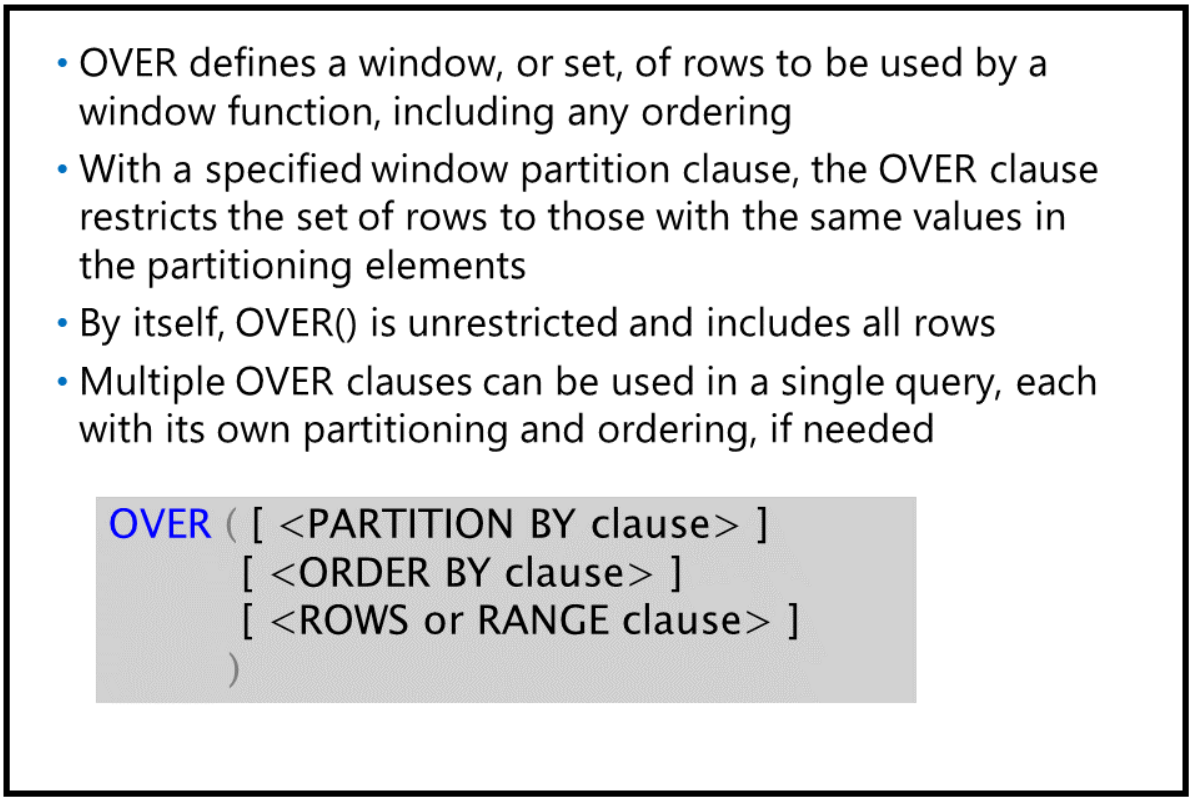
Windowing Example

```
SELECT Category, Qty, Orderyear,
        SUM(Qty) OVER (PARTITION BY category    ORDER BY Orderyear
                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningQty
FROM Sales.CategoryQtyYear;
```

The details of each component will be covered in future topics.

> **Note:** A single query can use multiple window functions, each with its own OVER clause. Each clause determines its own partitioning, ordering, and framing.

Using OVER



The OVER clause defines the window, or set, of rows that will be operated on by a window function, which we will look at in the next lesson. The OVER clause includes partitioning, ordering, and framing, where each is applicable.

Used alone, the OVER clause does not restrict the result set passed to the window function. Used with a PARTITION BY subclause, OVER restricts the set to those rows with the same values in the partitioning elements.

OVER Example

```
SELECT Category, Qty, Orderyear,
        ROW_NUMBER() OVER (ORDER BY Qty DESC) AS Running
FROM Sales.CategoryQtyYear
ORDER BY Running;
```

The partial result, further ordered by the Running column for display purposes:

```
Category        Qty         Orderyear   Running
--------------- ----------- ----------- --
Dairy Products  4374        2007        1
Confections     4137        2007        2
Beverages       3996        2007        3
Beverages       3694        2008        4
Seafood         3679        2007        5
Condiments      2895        2007        6
Seafood         2716        2008        7
```

```
Dairy Products  2689        2008        8
Grains/Cereals  2636        2007        9
```

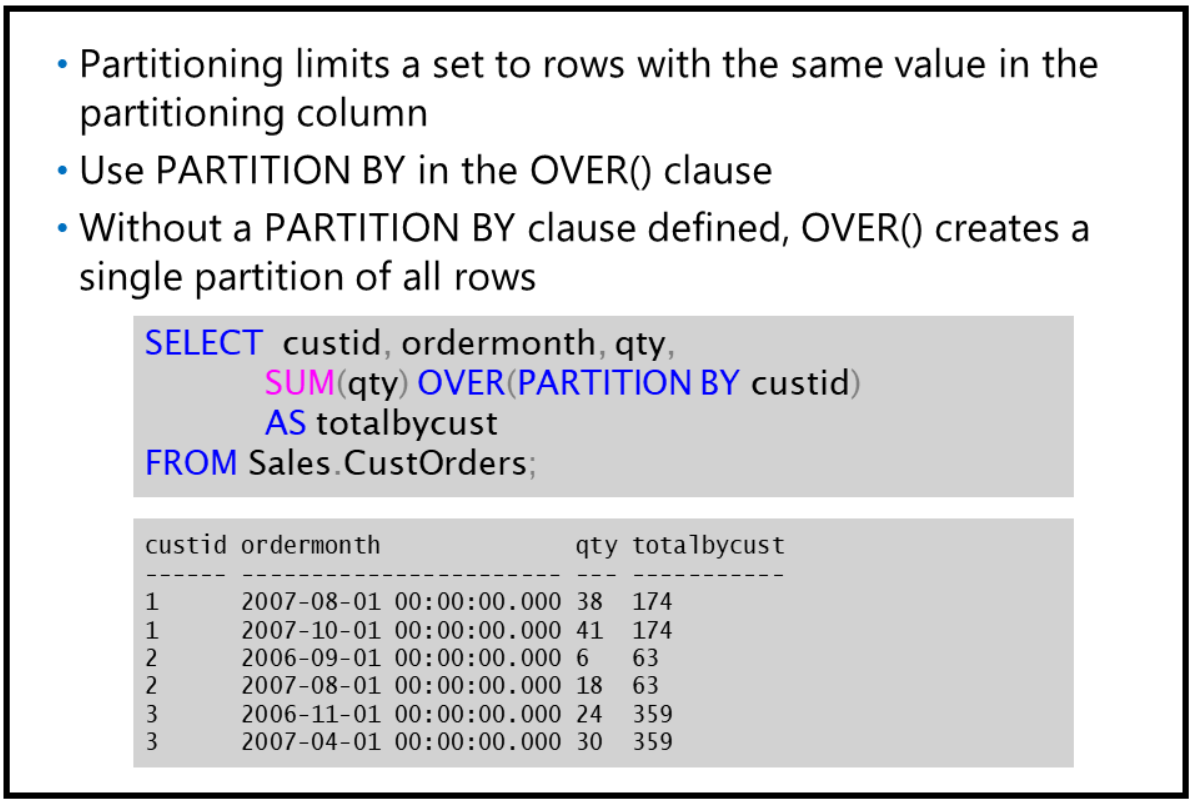
The next topics will build on this basic use of OVER to define a window of rows.

For further reading on the OVER clause, see Microsoft Docs:

OVER Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402778

Partitioning Windows



Partitioning a window limits a set to rows with the same value in the partitioning column.

PARTITION BY Code Snippet

```
<function_name>() OVER(PARTITION BY Category)
```

As you have learned, if no partition is defined, then the OVER() clause returns all rows from the underlying query's result set to the window function.

PARTITION BY Example

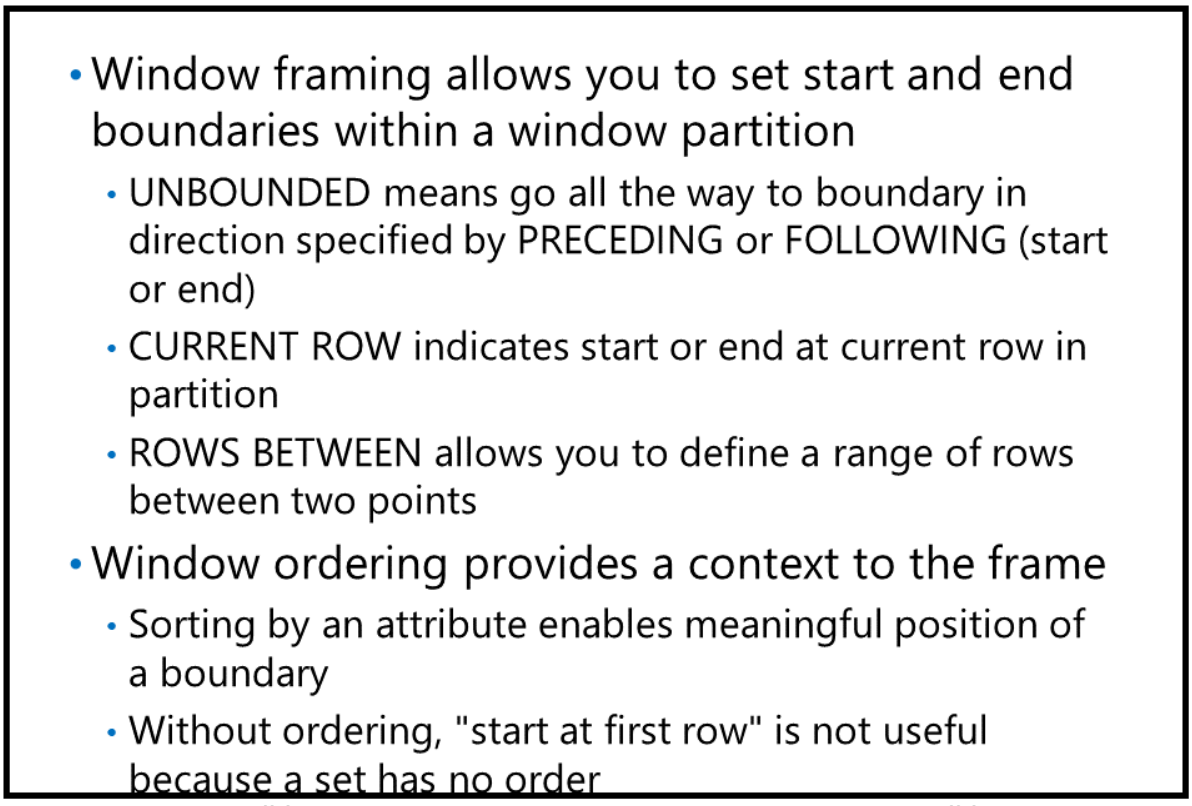
```
SELECT Category, Qty, Orderyear,
        ROW_NUMBER() OVER (PARTITION BY Category ORDER BY Qty DESC) AS Running
FROM Sales.CategoryQtyYear
ORDER BY Category;
```

The partial result:

Category	Qty	Orderyear	Running
Beverages	3996	2007	1
Beverages	3694	2008	2
Beverages	1842	2006	3
Condiments	2895	2007	1
Condiments	1441	2008	2
Condiments	962	2006	3
Confections	4137	2007	1
Confections	2412	2008	2
Confections	1357	2006	3

Note: If you intend to add framing to the window partition, an ORDER BY subclause will also be needed in the OVER clause, as discussed in the next topic.

Ordering and Framing



As you have learned, you use window partitions to define a subset of rows within the outer window defined by OVER. In a similar approach, window framing allows you to further restrict the rows available to the window function. You can think of a frame as a moving window over the data, starting and ending at positions you define.

To define window frames, use the ROW or RANGE subclauses to provide a starting and an ending boundary. For example, to set a frame that extends from the first row in the partition to the current row (such as to create a moving window for a running total), follow these steps:

1. Define an OVER clause with a PARTITION BY element.
2. Define an ORDER BY subclause to the OVER clause. This will cause the concept of "first row" to be meaningful.
3. Add the ROWS BETWEEN subclause, setting the starting boundary using UNBOUNDED PRECEDING. UNBOUNDED means go all the way to the boundary in the direction specified as PRECEDING (before). Add the CURRENT ROW element to indicate the ending boundary is the row being calculated.

> **Note:** Since OVER returns a set, and sets have no order, an ORDER BY subclause is required for the framing operation to be useful. This can be (and typically is) different from ORDER BY, which determines the display order for the final result set.

Framing Example

```
SELECT Category, Qty, Orderyear,
        SUM(Qty) OVER (PARTITION BY Category    ORDER BY Orderyear
                ROWS BETWEEN UNBOUNDED PRECEDING
                AND CURRENT ROW) AS RunningQty
FROM Sales.CategoryQtyYear;
```

The partial results:

```
Category        Qty         Orderyear   RunningQty
--------------- ----------- ----------- -----------
Beverages       1842        2006        1842
Beverages       3996        2007        5838
Beverages       3694        2008        9532
Condiments      962         2006        962
Condiments      2895        2007        3857
Condiments      1441        2008        5298
Confections     1357        2006        1357
Confections     4137        2007        5494
Confections     2412        2008        7906
Dairy Products  2086        2006        2086
Dairy Products  4374        2007        6460
Dairy Products  2689        2008        9149
```

Demonstration: Using OVER and Partitioning

In this demonstration, you will see how to use OVER, PARTITION BY, and ORDER BY clauses.

Demonstration Steps

Use OVER, PARTITION BY, and ORDER BY Clauses

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod13\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and then press Enter.
5. Wait for the script to finish, and then press any key.
6. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod13\Demo** folder.
8. In Solution Explorer, open the **11 - Demonstration A.sql** script file.
9. Select the code under the comment **Step 1**, and then click **Execute**.
10. Select the code under the comment **Step 2**, and then click **Execute**.
11. Select the code under the comment **Step 3**, and then click **Execute**.
12. Select the code under the comment **Step 4**, and then click **Execute**.
13. Select the code under the comment **Step 5**, and then click **Execute**.
14. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Sequencing Activity

Put the following elements into the logical order in which they are processed in a windowed query by numbering each to indicate the correct order.

The OVER clause

The PARTITION BY clause

The ROW or RANGE clause

Check answer Show solution Reset

Correct

Lesson 2: Exploring Window Functions

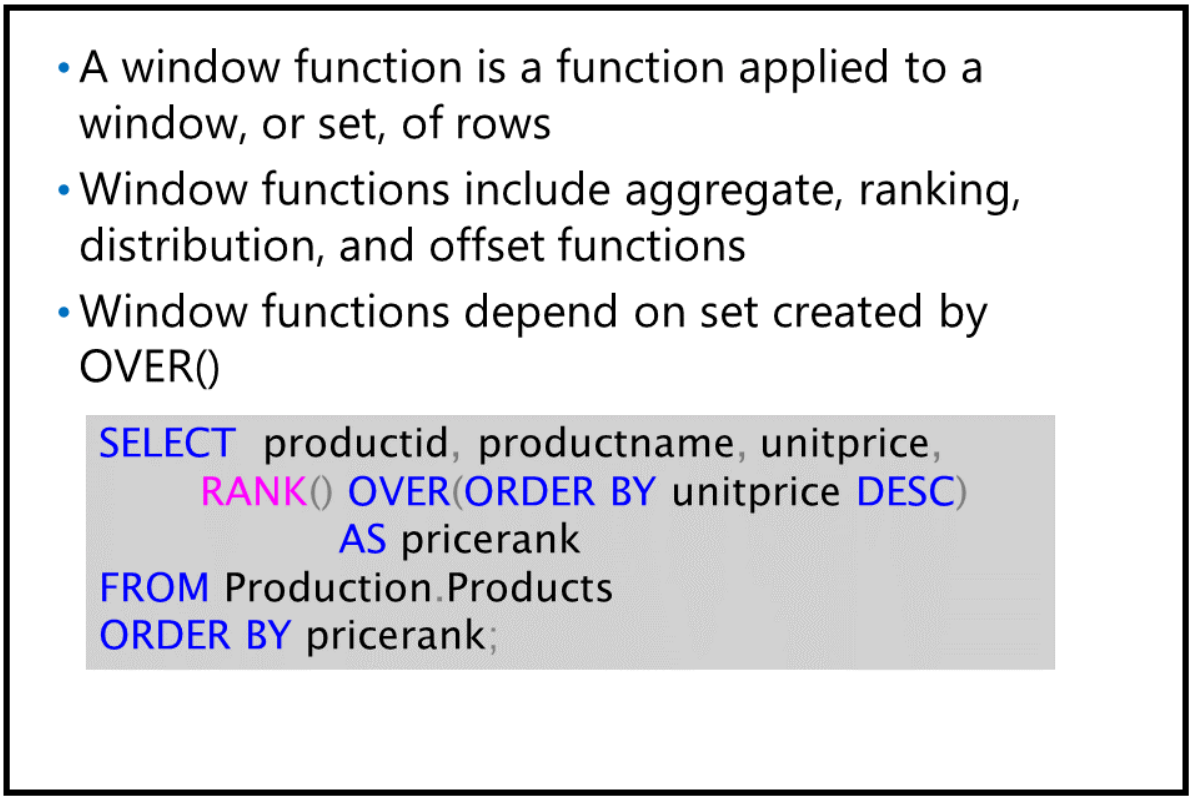
SQL Server provides window functions to operate on a window of rows. In addition to window aggregate functions, which you will find to be conceptually similar to grouped aggregate functions, you can use window ranking, distribution, and offset functions in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use window aggregate functions.
- Write queries that use window ranking functions.
- Write queries that use window offset functions.

Defining Window Functions



A window function is applied to a window, or set, of rows. Earlier in this course, you learned about group aggregate functions such as SUM, MIN, and MAX, which operated on a set of rows defined by a GROUP BY clause. In

window operations, you can use these functions, in addition to others, to operate on a set of rows defined in a window by an OVER clause and its elements.

SQL Server window functions can be found in the following categories, which will be discussed in the next topics:

- Aggregate functions, such as SUM, which operate on a window and return a single row.
- Ranking functions, such as RANK, which depend on a sort order and return a value representing the rank of a row, with respect to other rows in the window.
- Distribution functions, such as CUME_DIST, which calculate the distribution of a value in a window of rows.
- Offset functions, such as LEAD, which return values from other rows relative to the position of the current row.

When used in windowing scenarios, these functions depend on the result set returned by the OVER clause and any further restrictions you provide within OVER, such as partitioning and framing.

RANK Example

```
SELECT  productid, productname, unitprice,
        RANK() OVER(ORDER BY unitprice DESC) AS pricerank
FROM Production.Products
ORDER BY pricerank;
```

The partial result:

```
productid   productname   unitprice             pricerank
----------- ------------- --------------------- ---------
38          Product QDOMO 263.50                1
29          Product VJXYN 123.79                2
9           Product AOZBW 97.00                 3
20          Product QHFFP 81.00                 4
18          Product CKEDC 62.50                 5
59          Product UKXRI 55.00                 6
51          Product APITJ 53.00                 7
62          Product WUXYK 49.30                 8
43          Product ZZZHR 46.00                 9
28          Product OFBNT 45.60                 10
27          Product SMIOH 43.90                 11
63          Product ICKNK 43.90                 11
8           Product WVJFP 40.00                 13
```

RANK with PARTITION Example

```
SELECT  categoryid, productid, unitprice,
        RANK() OVER(PARTITION BY categoryid ORDER BY unitprice DESC) AS pricerank
FROM Production.Products
ORDER BY categoryid, pricerank;
```

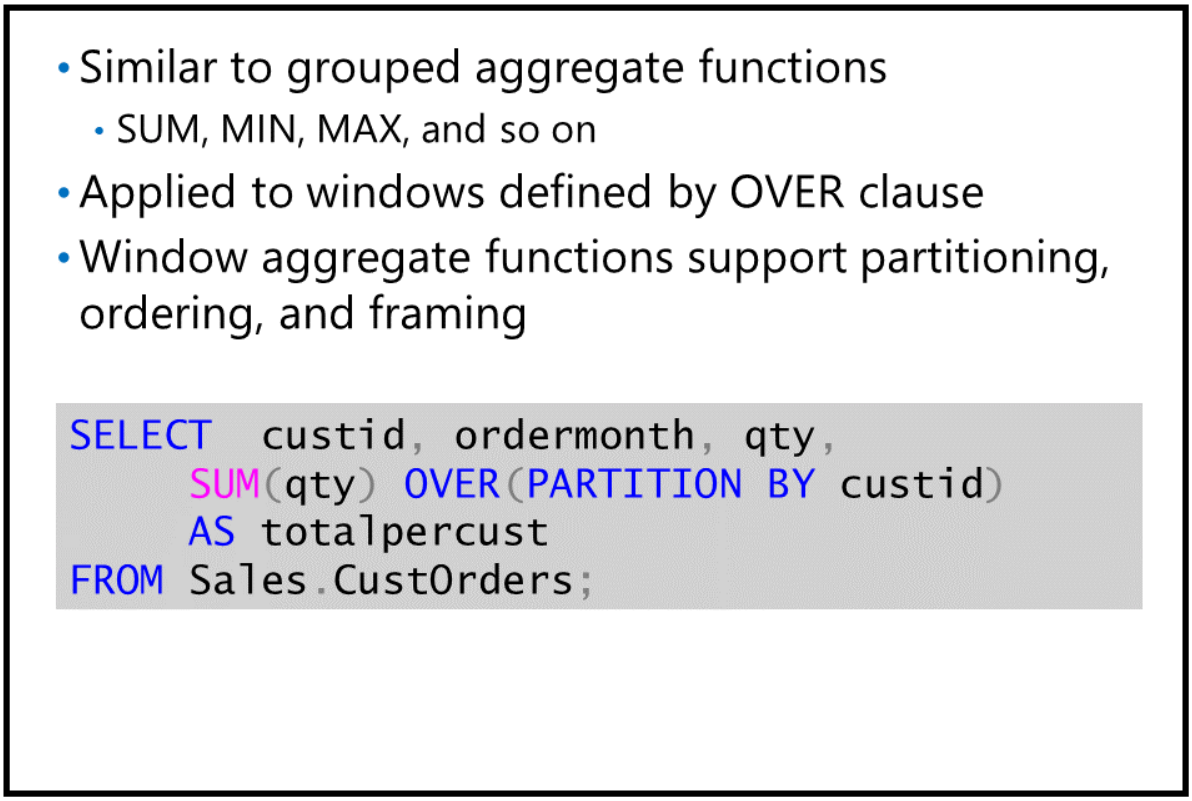
The partial result, edited for space:

categoryid	productid	unitprice	pricerank
1	38	263.50	1
1	43	46.00	2
1	2	19.00	3
2	63	43.90	1
2	8	40.00	2
2	61	28.50	3
2	6	25.00	4
3	20	81.00	1
3	62	49.30	2
3	27	43.90	3
3	26	31.23	4

Notice that the addition of partitioning allows the window function to operate at a more granular level than when OVER returns an unrestricted set.

> **Note:** Repeating values and gaps in the pricerank column are expected when using RANK in case of ties. Use DENSE_RANK if gaps are not desired. See the next topics for more information.

Window Aggregate Functions



Window aggregate functions are similar to the aggregate functions you have already used in this course. They aggregate a set of rows and return a single value. However, when used in the context of windows, they operate on the set returned by the OVER clause, not on a set defined by a grouped query using GROUP BY.

Window aggregate functions provide support for windowing elements you have learned about in this module, such as partitioning, ordering, and framing. Unlike other window functions, ordering is not required with aggregate functions, unless you are also specifying a frame.

Window Aggregate Example

```
SELECT  custid,
        ordermonth,
        qty,
        SUM(qty) OVER ( PARTITION BY custid ) AS totalpercust
FROM    Sales.CustOrders;
```

The partial result, edited for space:

```
custid      ordermonth              qty         totalpercust
----------- ----------------------- ----------- ------------
1           2007-08-01 00:00:00.000 38          174
1           2007-10-01 00:00:00.000 41          174
1           2008-01-01 00:00:00.000 17          174
2           2006-09-01 00:00:00.000 6           63
2           2007-08-01 00:00:00.000 18          63
```

```
3           2006-11-01 00:00:00.000 24          359
3           2007-04-01 00:00:00.000 30          359
3           2007-05-01 00:00:00.000 80          359
4           2007-02-01 00:00:00.000 40          650
4           2007-06-01 00:00:00.000 96          650
```

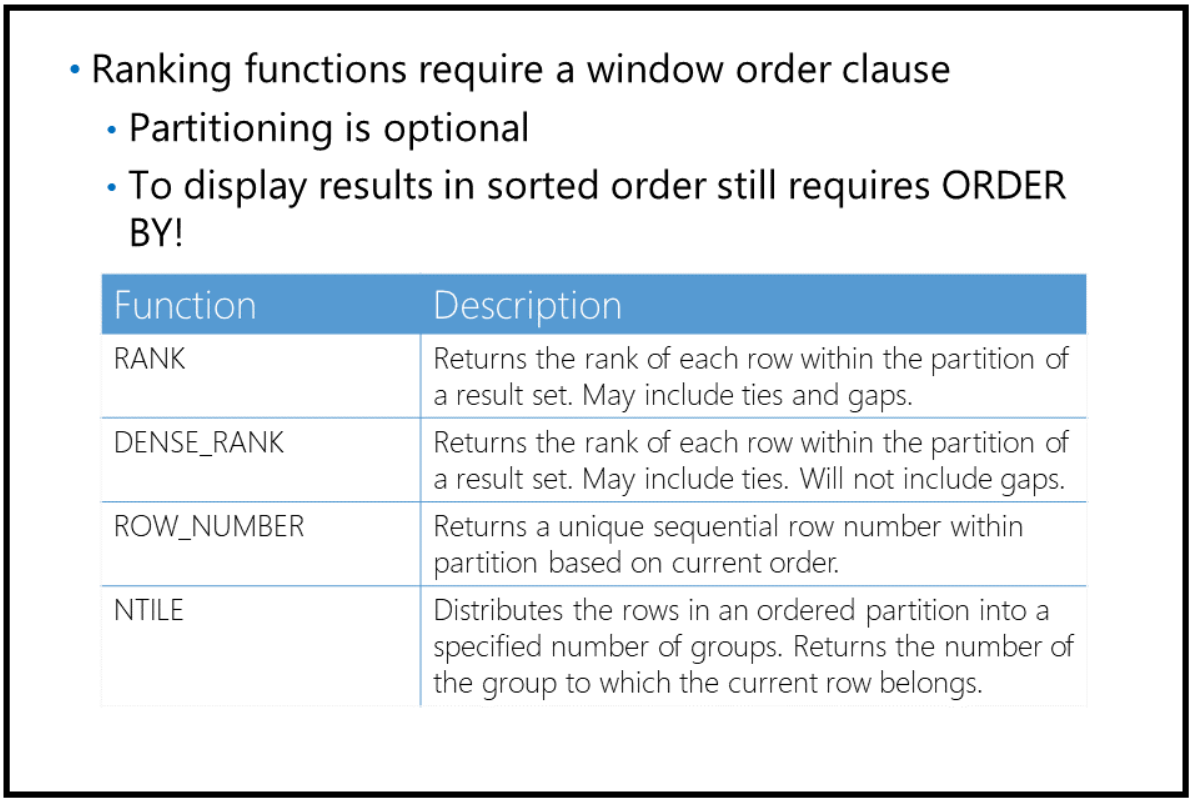
Further Window Aggregate Example


```
SELECT  custid, ordermonth, qty,
        SUM(qty) OVER ( PARTITION BY custid ) AS custtotal,
        CAST(100. * qty/SUM(qty) OVER ( PARTITION BY custid )AS NUMERIC(8,2)) AS
OfTotal
FROM    Sales.CustOrders;
```

The result:

```
custid ordermonth              qty custtotal  OfTotal
------ ----------------------- --- ---------- -------
1      2007-08-01 00:00:00.000 38  174         21.84
1      2007-10-01 00:00:00.000 41  174         23.56
1      2008-01-01 00:00:00.000 17  174         9.77
1      2008-03-01 00:00:00.000 18  174         10.34
1      2008-04-01 00:00:00.000 60  174         34.48
2      2006-09-01 00:00:00.000 6   63          9.52
2      2007-08-01 00:00:00.000 18  63          28.57
2      2007-11-01 00:00:00.000 10  63          15.87
2      2008-03-01 00:00:00.000 29  63          46.03
3      2006-11-01 00:00:00.000 24  359         6.69
3      2007-04-01 00:00:00.000 30  359         8.36
3      2007-05-01 00:00:00.000 80  359         22.28
3      2007-06-01 00:00:00.000 83  359         23.12
3      2007-09-01 00:00:00.000 102 359         28.41
3      2008-01-01 00:00:00.000 40  359         11.14
```


Window Ranking Functions



Window ranking functions return a value representing the rank of a row with respect to other rows in the window. To accomplish this, ranking functions require an ORDER BY element within the OVER clause, to establish the position of each row within the window.

> **Note:** Remember that the ORDER BY element within the OVER clause affects only the processing of rows by the window function. To control the display order of the results, add an ORDER BY clause to the end of the SELECT statement, as with other queries.

The primary difference between RANK and DENSE_RANK is the handling of rows when there are tie values.

RANK and DENSE_RANK Example

```
SELECT CatID, CatName, ProdName, UnitPrice,
        RANK() OVER(PARTITION BY CatID ORDER BY UnitPrice DESC) AS PriceRank,
        DENSE_RANK() OVER(PARTITION BY CatID ORDER BY UnitPrice DESC) AS DensePriceRank
FROM Production.CategorizedProducts
ORDER BY CatID;
```

The partial results follow. Note the rank numbering of the rows following the products with a unitprice of 18.00:

```
CatID CatName    ProdName      UnitPrice PriceRank DensePriceRank
----- ---------- ------------- --------- --------- --------------
1     Beverages  Product QDOMO 263.50    1         1
1     Beverages  Product ZZZHR 46.00     2         2
```

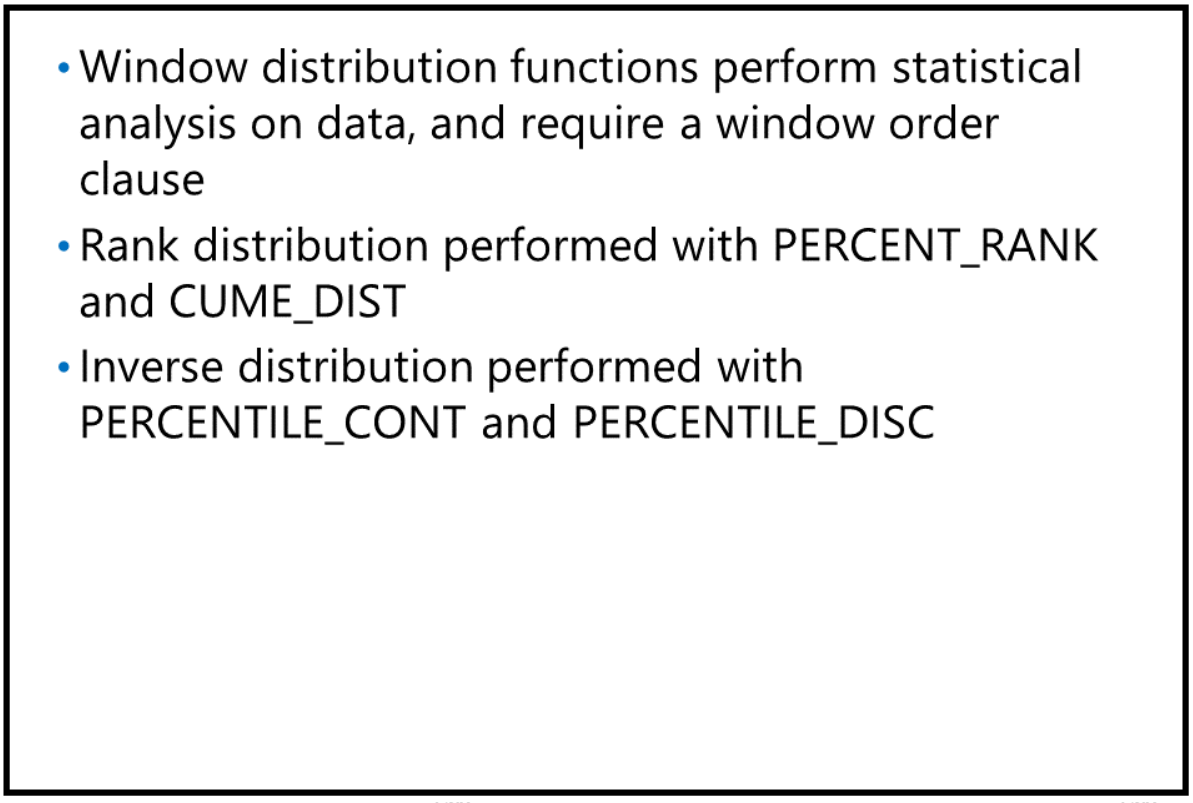
```
1    Beverages Product RECZE 19.00    3        3
1    Beverages Product HHYDP 18.00    4        4
1    Beverages Product LSOFL 18.00    4        4
1    Beverages Product NEVTJ 18.00    4        4
1    Beverages Product JYGFE 18.00    4        4
1    Beverages Product TOONT 15.00    8        5
1    Beverages Product XLXQF 14.00    9        6
1    Beverages Product SWNJY 14.00    9        6
1    Beverages Product BWRLG 7.75     11       7
1    Beverages Product QOGNU 4.50     12       8
```

Go to Ranking Functions (Transact-SQL) in the SQL Server Technical Documentation:

Ranking Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402779

Window Distribution Functions



Window distribution functions perform statistical analysis on the rows within the window or window partition. Partitioning a window is optional for distribution functions, but ordering is required.

Distribution functions return a rank of a row, but instead of being expressed as an ordinal number, as with RANK, DENSE_RANK, or ROW_NUMBER, it is expressed as a ratio between 0 and 1. SQL Server provides rank distribution with the PERCENT_RANK and CUME_DIST functions. It provides inverse distribution with the PERCENTILE_CONT and PERCENTILE_DISC functions.

These functions are listed here for completeness only and are beyond the scope of this course. For more information, see the SQL Server Technical Documentation:

Analytic Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402780

Window Offset Functions

- Window offset functions allow comparisons between rows in a set without the need for a self-join
- Offset functions operate on a position relative to the current row, or to the start or end of the window frame

Function	Description
LAG	Returns an expression from a previous row that is a defined offset from the current row. Returns NULL if no row at specified position.
LEAD	Returns an expression from a later row that is a defined offset from the current row. Returns NULL if no row at specified position.
FIRST_VALUE	Returns the first value in the current window frame. Requires window ordering to be meaningful.
LAST_VALUE	Returns the last value in the current window frame. Requires window ordering to be meaningful.

Windows offset functions give access to values located in rows other than the current row. This can enable queries that perform comparisons between rows, without the need to join the table to itself.

Offset functions operate on a position that is either relative to the current row, or relative to the starting or ending boundary of the window frame. LAG and LEAD operate on an offset to the current row. FIRST_VALUE and LAST_VALUE operate on an offset from the window frame.

> **Note:** Since FIRST_VALUE and LAST_VALUE operate on offsets from the window frame, it is important to remember to specify framing options other than the default of RANGE BETWEEN UNBOUND PRECEDING AND CURRENT ROW.

Window Offset Function Example

```
SELECT employee, orderyear ,totalsales AS currsales,
   LEAD(totalsales, 1,0) OVER (PARTITION BY employee ORDER BY orderyear) AS nextsales
FROM Sales.OrdersByEmployeeYear
ORDER BY employee, orderyear;
```

The partial results:

```
employee orderyear currsales nextsales
-------- --------- --------- ---------
1           2006   38789.00  97533.58
1           2007   97533.58  65821.13
1           2008   65821.13  0.00
2           2006   22834.70  74958.60
2           2007   74958.60  79955.96
2           2008   79955.96  0.00
3           2006   19231.80  111788.61
3           2007   111788.61 82030.89
3           2008   82030.89  0.00
```

Demonstration: Exploring Windows Functions

In this demonstration, you will see how to use window aggregate, ranking, and offset functions.

Demonstration Steps

Use Window Aggregate, Ranking, and Offset Functions

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Select the code under the comment **Step 8**, and then click **Execute**.
10. Select the code under the comment **Step 9**, and then click **Execute**.
11. Select the code under the comment **Step 10**, and then click **Execute**.
12. Select the code under the comment **Step 11**, and then click **Execute**.
13. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Categorize Activity

Place each windowing function into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Window Aggregate Functions

- MIN()
- MAX()
- SUM()

Window Ranking Function

- DENSERANK()
- ROW_NUMBER()
- NTITLE()
- RANK()

Window Distribution Functions

- PERCENTILE_CONT()
- CUME_DIST()
- PERCENTILE_DISC()
- PERCENT_RANK()

Check answer Show solution Reset

Correct

Lab: Using Window Ranking, Offset, and Aggregate Functions

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. To fill these requests, you will need to calculate ranking values, as well as the difference between two consecutive rows, and running totals. You will use window functions to achieve these calculations.

Objectives

After completing this lab, you will be able to:

- Write queries that use ranking functions.
- Write queries that use offset functions.
- Write queries that use window aggregation functions.

Lab Setup

Estimated Time: 60 minutes

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Ranking Functions

Scenario

The sales department would like to rank orders by their values for each customer. You will provide the report by using the RANK function. You will also practice how to add a calculated column to display the row number in the SELECT clause.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column
3. Add an Additional Column Using the RANK Function
4. Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value
5. Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

6. Filter Only Orders with the Top Two Ranks

Detailed Steps ▲

Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab13\Starter** folder as Administrator.

Detailed Steps ▲

Task 2: Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab13\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the orderid, orderdate, and val columns in addition to a calculated column named rowno from the view Sales.OrderValues. Use the ROW_NUMBER function to return rowno. Order the row numbers by the orderdate column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

Detailed Steps ▲

Task 3: Add an Additional Column Using the RANK Function

1. Copy the previous T-SQL statement and modify it by including an additional column named rankno. To create rankno, use the RANK function, with the rank order based on the orderdate column.
2. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\53 - Lab Exercise 1 - Task 2 Result.txt. Notice the different values in the rowno and rankno columns for some of the rows.
3. What is the difference between the RANK and ROW_NUMBER functions?

Detailed Steps ▲

Task 4: Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

1. Write a SELECT statement to retrieve the orderid, orderdate, custid, and val columns, as well as a calculated column named orderrankno from the Sales.OrderValues view. The orderrankno column should display the rank per each customer independently, based on val ordering in descending order.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

Detailed Steps ▲

Task 5: Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

1. Write a SELECT statement to retrieve the custid and val columns from the Sales.OrderValues view. Add two calculated columns:
   - orderyear as a year of the orderdate column.
   - orderrankno as a rank number, partitioned by the customer and order year, and ordered by the order value in descending order.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\55 - Lab Exercise 1 - Task 4 Result.txt.

Detailed Steps ▲

Task 6: Filter Only Orders with the Top Two Ranks

1. Copy the previous query and modify it to filter only orders with the first two ranks based on the orderrankno column.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\56 - Lab Exercise 1 - Task 5 Result.txt.

Result: After this exercise, you should know how to use ranking functions in T-SQL statements.

Exercise 2: Writing Queries That Use Offset Functions

Scenario

You need to provide separate reports to analyze the difference between two consecutive rows. This will enable business users to analyze growth and trends.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)
2. Add a Column to Display the Running Sales Total
3. Analyze the Sales Information for the Year 2007

Detailed Steps ▲

Task 1: Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Define a CTE named OrderRows based on a query that retrieves the orderid, orderdate, and val columns from the Sales.OrderValues view. Add a calculated column named rowno using the ROW_NUMBER function, ordering by the orderdate and orderid columns.
3. Write a SELECT statement against the CTE and use the LEFT JOIN with the same CTE to retrieve the current row and the previous row based on the rowno column. Return the orderid, orderdate, and val columns for the current row and the val column from the previous row as prevval. Add a calculated column named diffprev to show the difference between the current val and previous val.
4. Execute the T-SQL code and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

Detailed Steps ▲

Task 2: Add a Column to Display the Running Sales Total

1. Write a SELECT statement that uses the LAG function to achieve the same results as the query in the previous task. The query should not define a CTE.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

Detailed Steps ▲

Task 3: Analyze the Sales Information for the Year 2007

1. Define a CTE named SalesMonth2007 that creates two columns: monthno (the month number of the orderdate column) and val (aggregated val column). Filter the results to include only the order year 2007 and group by monthno.
2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
   - **avglast3months**. This column should contain the average sales amount for the last three months before the current month, using a window aggregate function. You can assume that there are no missing months.
   - **ytdval**. This column should contain the cumulative sales value up to the current month.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\63 - Lab Exercise 2 - Task 3 Result.txt.

Result: After this exercise, you should be able to use the offset functions in your T-SQL statements.

Exercise 3: Writing Queries That Use Window Aggregate Functions

Scenario

To better understand the cumulative sales value of a customer through time and to provide the sales analyst with a year-to-date analysis, you will have to write different SELECT statements that use the window aggregate functions.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase
2. Add a Column to Display the Running Sales Total
3. Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

Detailed Steps ▲

Task 1: Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid, orderid, orderdate, and val columns from the Sales.OrderValues view. Add a calculated column named percoftotalcust containing a percentage value of each order sales amount compared to the total sales amount for that customer.

3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

Detailed Steps ▲

Task 2: Add a Column to Display the Running Sales Total

1. Copy the previous SELECT statement and modify it by adding a new calculated column named runval. This column should contain a running sales total for each customer based on order date, using orderid as the tiebreaker.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

Detailed Steps ▲

Task 3: Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

1. Copy the SalesMonth2007 CTE in the final task in exercise 2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
   - **avglast3months**. This column should contain the average sales amount for the last three months before the current month using a window aggregate function. You can assume that there are no missing months.
   - **ytdval**. This column should contain the cumulative sales value up to the current month.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\74 - Lab Exercise 3 - Task 3 Result.txt.
3. Close SQL Server Management Studio without saving any changes.

Result: After this exercise, you should have a basic understanding of how to use window aggregate functions in T-SQL statements.

Module Review and Takeaways

In this module, you have learned how to:

- Describe the benefits of using window functions.

- Restrict window functions to rows defined in an OVER clause, including partitions and frames.
- Write queries that use window functions to operate on a window of rows and return ranking, aggregation, and offset comparison results.

Review Question(s)

Check Your Knowledge

Discovery

What results will be returned by a ROW_NUMBER function if there is no ORDER BY clause in the query?

Show solution Reset

An unordered set.

Check Your Knowledge

Discovery

Which ranking function would you use to return the values 1,1,3? Which would return 1,1,2?

Show solution Reset

RANK, DENSE_RANK.

Check Your Knowledge

Discovery

Can a window frame extend beyond the boundaries of the window partition defined in the same OVER() clause?

Show solution Reset

No.