**Chapter 1 – Partitions, Frames and the OVER() clause**

The goal of this chapter is to describe the OVER() clause, its various configurations in terms of how partitions and window frames are created and how data is sorted in the partition so that the window function can operate on the values. (The OVER() clause together with partitions and window frames is what gives power to the window functions.)

We will cover window frames and how they are defined to present a set of partition rows that are processed by the window functions.

Several diagrams illustrate the concepts and then several examples are shown to illustrate the various ways partitions and window frames work on real data sets using window functions. We will discuss ROWS and RANGE clauses and how they operate on data in the partition.

We will also look at a case where we use subqueries in the ORDER BY clause instead of columns so as to eliminate expensive sort steps in a query plan.

A final short example is presented to show the new named windows feature in SQL Server 2022.

**What are Partitions and Window Frames?**

A TSQL query will generate an initial data result set. The result set can be divided into sections called partitions, like a partition for each set of rows by specific years. The window frame is like a smaller window into the partition defined by some row boundaries or ranges. For example, rows prior to and including the current row being processed or all rows after and including the current row within the partition. The entire data set can also be a partition by itself, and a window can be defined that uses all rows in the single partition. It all depends on how you include and define the PARTITION BY, ORDER BY and ROWS/RANGE window frame clauses.

These conditions can be specified with the OVER() clause. Let’s see how this works.

**What is an OVER() Clause?**

As stated earlier, the OVER() clause allows you to create partitions and windows into a data set generated by a query. It allows you to divide the data set into sections called partitions. The windows allow you to control how the windows functions are applied in terms of the data they will operate on.

Smaller windows called window frames can be created to further carve up the larger window defined by the PARTITION BY clause using ROWS and FRAMES clauses. These windows grow or move (as rows are processed) based on the boundaries defined by ROWS and RANGE frame specifications included in the OVER() clause. You can also specify how the rows are sorted via an ORDER BY clause so that the window function can process them.

This capability allows you to create queries such as three-month rolling averages and year to date, quarter to date and month to date totals. Each of the functions we discuss in this book can utilize the OVER() clause in this manner.

This is why they are called window functions!

**History of OVER() clause and window functions**

Below is a brief history of the window functions:

* Aggregate/ranking window functions without the ORDER BY clause support was introduced 2005.
* Seven years later, aggregate functions with support for the ORDER BY clause was introduced in 2012.
* Window Frames support (which we will discuss shortly) was also introduced in 2012.
* Ranking functions and some window capability were introduced in 2015.
* Batch mode window aggregate operator was introduced in 2016
* STRING\_AGG function was introduced 2017
* Named WINDOW capability introduced in SQL Server 2022.

The capability of the window functions has grown over the years and delivers a rich and powerful set of tools to analyze and solve complex data analysis problems.

**The Window Functions**

The window functions (which are the focus of this book) that can be used with the OVER() clause are assigned to three categories and are listed below. Please refer to table 1.1 below:

| **Aggregate Functions** | **Analytical Functions** | **Ranking Functions** |
| --- | --- | --- |
| COUNT() | CUME\_DIST() | RANK() |
| COUNT\_BIG() | FIRST\_VALUE() | DENSE\_RANK() |
| SUM() | LAST\_VALUE() | NTILE() |
| MAX() | LAG() | ROW\_NUMBER() |
| MIN() | LEAD() |  |
| AVG() | PERCENT\_RANK() |  |
| GROUPING() | PERCENTILE\_CONT() |  |
| STRING\_AGG() | PERCENTILE\_DISC() |  |
| STDEV() | CUME\_DIST() |  |
| STDEVP() |  |  |
| VAR() |  |  |
| VARP() |  |  |

***Table 1.1 – Aggregate, Analytical and Ranking functions***

Each of the subsequent chapters will create and discuss queries for these categories for four industry specific databases that are in scope for this book. Please refer to appendix 1 for syntax and descriptions of what each of the above functions do if you are unfamiliar with them or need a refresher on how to use them in a query.

**The OVER() Clause**

The over clause appears right after the window function in the SELECT clause. Our first example uses the SUM() function to aggregate sales amounts by year and month. Please refer to listing 1.1 below:

***Listing 1.1 – SUM() function with the OVER() clause.***

SELECT OrderYear,OrderMonth,SalesAmount,

SUM(SalesAmount) OVER(

PARTITION BY OrderYear

ORDER BY OrderMonth ASC

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS AmountTotal

FROM OverExample

ORDER BY OrderYear,OrderMonth

GO

Between a set of parentheses after the OVER() keyword, three other clauses can be included such as PARTITION BY, ORDER BY, and ROWS or RANGE clauses (to define the window frame that presents the rows to the function for processing.)

Even if you have an ORDER BY clause in the OVER() clause you can also include the usual ORDER BY clause at the end of the query to sort the final processed result set in any order you feel is appropriate to the business requirements the query solves.

**Syntax**

Below are three basic syntax templates that can be used with the window functions. Reading these syntax templates is easy. Just keep in mind key words between square brackets mean they are optional. Below is the first syntax template available for the OVER() clause:

**Syntax 1**

**<Window Function> OVER (**

**[ <PARTITION BY expression> ]**

**[ <ORDER BY expression ASC | DESC > ]**

**[ <ROW or RANGE expression > ]**

**)**

Most of the window functions use this first syntax, and it is composed of three main clauses, the PARTITION BY clause, the ORDER BY clause and a ROW or RANGE specification. You can include one or more of these clauses or none. These combinations will affect how the partition is defined. For example, if you do not include a PARTITION BY clause the entire data set is considered one large partition. The expression is usually one or more columns but in the case of the PARTITION BY AND ORDER BY clauses it could also be a subquery (refer to appendix 1).

The **Window Function** is one of the functions identified in table 1.

This first syntax is pretty much the same for all functions except for the PERCENTILE\_DISC() and PERCENTILE\_CONT() functions that uses a slight variation:

**PERCENTILE\_DISC (numeric literal ) WITHIN GROUP (**

**ORDER BY expression [ ASC | DESC ]**

**) OVER ( [ <PARTITION BY expression> ] )**

These functions are used to calculate the percentile discrete and percentile continuous values in a data set column. The numerical literal can be a value like .25, .50 or .75 that is used to specify the percentile you wish to calculate. Notice that the ORDER BY clause is inserted between the parenthesis of the WITHIN GROUP command and the OVER() clause just includes the PARTITION BY clause.

Don’t worry about what this does for now, examples will be given that make the behavior of this code clear. For now, just understand that there are 3 basic syntax templates to be aware of.

In our chapter examples the expression will usually be a column or columns separated by commas although you can use other data objects like queries. Please refer to the Microsoft SQL Server documentation to check out the detailed syntax specification or appendix 1.

Lastly, our third syntax template applies to SQL Server 2022 (release 16.x). The window capability has been enhanced that allows you to specify window options in a named window that appears at the end of the query:

**WINDOW <window name> AS (**

**[ <PARTITION BY clause> ]**

**[ <ORDER BY clause> ]**

**[ <ROW or RANGE clause> ]**

**)**

As of this writing, SQL Server 2022 is available for evaluation only. Here is an example TSQL query that uses this new feature. Please refer to listing 1.2 below:

***Listing 1.2 – SQL Server 2022 Named Window feature***

SELECT OrderYear,OrderMonth,SalesAmount,

SUM(SalesAmount) OVER **SalesWindow** AS SQPRangeUPCR

FROM OverExample

WINDOW **SalesWindow** AS (

PARTITION BY OrderYear

ORDER BY OrderMonth

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

);

GO

The name of the window is **SalesWindow** and it is used right after the OVER operator instead of the partition by, order by and range specification clauses as used in the first syntax template we discussed.

Probably a good feature in case you have multiple window functions in your SELECT clause that need to use this partition and window frame configuration. This would avoid repeating the partition code in each column of the SELECT clause.

The partition and order by, range clauses are declared at the end of the query between parenthesis after the WINDOW keyword instead of right after the OVER() clause.

If you want to play around with this download and install the 2022 evaluation license and try it out on the example code available with the book or on your own queries. The setup and download are fast and simple. Make sure you get the latest version of SSMS. These are available on Microsoft’s download web site.

**Partitions and Frames**

Finally, we get to discuss what a partition looks like and what frames or window frames are.

Basically, a query will generate a data result set which you can carve up into sections called partitions by including the PARTITION BY clause inside the OVER() clause. For each partition you can further define smaller windows to fine tune how the window functions will be applied.

A picture is worth a thousand words so let’s look at one now. Please refer to Figure 1.1 below:

Diagram

Description automatically generated

***Figure 1.1 – A simple data set with 3 partitions and example frame***

Here we have a simple data set composed of 8 rows. There are 3 example partitions in this data set. One can include all 8 rows of the data set, the other two include rows identified by the TYPE column. There are only two type values, type A and type B so each of these partitions will have 4 rows each. By the way, you cannot include multiple PARTITION BY clauses in an OVER() clause.

You can define only 1 partition per OVER() clause although you can have more than one column in the SELECT clause of the query that uses a partition. You can specify different column combinations to define the partitions.

Where the power of this architecture comes in is that we can create smaller window frames against the partition by using the ROW or RANGE operators. These will allow you to specify how many rows before and/or after the current row being processed will be used by the window function.

In our example snapshot above, the current row is row three and the window frame is defined so it includes only the prior row, the current row and next row relative to the current row. If we apply the SUM() function to this window frame and add all the values we get the result 60 (15 + 20 + 25). (Remember this is within the first partition which contains only four rows).

If processing continues on the next row, row four, only rows three and four are available to the SUM() function and the result is 45 (20 + 25). I neglected to mention that if we start at row one, then only rows one and two are available to the SUM()function because there is no prior row. The function returns the value 25 (10 + 15).

How do we control this type of processing? All we need to do is add a ROWS or RANGE specification to the query if required. We could also include an ORDER BY clause to specify how to order the rows within the partition so that the window function is applied as needed. For example, generate rolling monthly totals by month. Starting of course at month 1 (January) and ending at month 12 (December).

Sounds easy but we need to be aware of a few scenarios around default processing when we leave the ORDER BY clause and/or the PARTITION clause out. We will discuss these shortly.

**ROWS Frame Definition**

The ROWS clause operates on the physical set of rows belonging to the partition. Because the ROWS clause operates on the rows in the partition it is considered a physical operation.

The ORDER BY clause allows you to specify the logical order of the rows in the partition so the window function can evaluate them. For example, if you have sales data by year, and month, you would set the logical order of the rows in the partition by month so that the SUM() function can generate rolling year to date totals for each month. You can specify optional ascending and descending sort orders by using the keywords ASC and DESC.

The ROWS clause allows you to define the window frame into the partition. There are several variations of the ROWS clause that we need to be aware of. Below are two variations that generate the same window frame.

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

ROWS UNBOUNDED PRECEDING

This clause tells the function to operate on the current row and all rows preceding the current row if there are any in the partition. A simple diagram makes it all clear.

Please refer to Figure 1.2 below:

Diagram

Description automatically generated

***Figure 1.2 – Include the current row and all preceding rows***

If we start at row 1, since there are no prior rows in the partition before this row the SUM() function returns the value 10.

Moving on to row 2, the SUM() function will include the only available prior row (row 1) so the result is 25 (10 + 15).

Next, (shown in the figure above) the current row to be processed is row 3. The SUM() function will evaluate row 3 plus rows 1 and 2 in order to generate the total. The result is 45 (10 + 15 + 20).

Lastly, moving to row 4 the function will include the current row and all prior rows in its calculation and return 70 (10 + 15+ 20 +25). Processing now concludes for this partition.

Moving to partition B the processing repeats itself and only rows from partition B are included. All rows in partition A are ignored.

Our next ROWS clause takes us in the opposite direction.

ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

ROWS UNBOUNDED FOLLOWING (will not work, not supported)

The first clause is legal but the second is not. It is not supported. One would think that if there is a ROWS UNBOUNDED PRECEDING there should be a ROWS UNBOUNDED FOLLOWING but it is not supported at this time. Go figure!

As stated earlier, this clause takes us in the opposite direction than the prior scenario we just discussed. It will allow the aggregate or other window functions to include the current row and all succeeding rows until all rows in the partition are exhausted. Our next diagram shows us how this works.

Please refer to Figure 1.3 below:

Graphical user interface, diagram

Description automatically generated

***Figure 1.3 – Process the current row and all succeeding rows***

Remember rows are processed one by one in the partition.

If processing starts at row one, then all four values are included to calculate a sum of 70 (10 + 15 + 20 +25).

Next, if the current row being processed is row 2 as in the example above, then the SUM() function will include rows 2 thru 4 to generate the total value. It will generate a result of 60 (15 + 20 + 25).

Moving on to row 3 it will only include rows 3 and 4 and generate a total value of 45 (20 + 25).

Once processing gets to row 4, only the current row is used as there are no more rows available in the partition. The SUM() function calculates a total of 25.

When processing resumes at the next partition the entire scenario is repeated.

What if we do not want to include all prior or following rows but only a few before or after? The next window frame clause will accomplish the trick for limiting the number of following rows.

ROWS BETWEEN CURRENT ROW AND n FOLLOWING

Including this clause in the OVER() clause will allow us to control the number of rows to include relative to the current row. That is, how many rows after the current row are going to be used from the available partition rows. Let’s examine another simple example where we want to include only 1 row following the current row in the calculation.

Please refer to Figure 1.4 below:

Diagram

Description automatically generated

***Figure 1.4 – Current row and one row following***

Here is the window frame clause to include in the OVER() clause:

ROWS BETWEEN CURRENT AND 1 FOLLOWING

Processing starts at row 1. I think by now you understand that only rows 1 and 2 are used and the result is 25 (10 + 15).

Next, in the example above, row 2 is the current row. If only the next row is included then the SUM() function will return a total of 35 (15 + 20).

Moving to row 3 (the next current row) the SUM() function will return 45 as the sum (20 plus 25).

Finally when processing gets to the last row in the partition then only row 4 is used and the SUM() function returns 25.

When processing continues to the next partition the sequence repeats itself ignoring the values from the prior partition.

We can now see how this process creates windows into the partition that change as the rows are processed one by one by the window function. Remember, the order in which the rows are processed is controlled by the ORDER BY clause used in the OVER() clause.

Next example takes us in the opposite direction. We want to include the current row and the prior two rows (if there are any) in the calculation.

The window frame clause is:

ROWS BETWEEN n PRECEDING AND CURRENT ROW

The letter n represents an unsigned integer value specifying the number of rows. The example below uses 2 as the number of rows.

Please refer to Figure 1.5 below:

Diagram

Description automatically generated

***Figure 1.5 – Include two prior rows and the current row.***

The window frame for this scenario is:

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

Starting at row 1 (the current row) the SUM() function will only use row 1 as there are no prior rows. The value returned by the window function is 10.

Moving to row 2 there is only one prior row so the SUM() function returns 25 (10 + 15)

Next, the current row is row 3 so the SUM() function uses the two prior rows and returns a value of 45 (10 + 15 +20).

In the figure above the current row being processed is row 4 and the preceding two rows are included, the SUM() function when applied will return a total value of 60 (15 + 20 + 25).

Since all rows of the first partition have been processed, we move to partition B and this time only row 5 is used as there are no prior rows. We are at the beginning of the partition. The SUM() function returns a total value of 30).

Processing continues at row 6 so the sum function processes rows 5 and 6 (the current and prior rows). The total value calculated is 65 (30 + 35)

Next, the current row is 7 so the window function will include rows 5, 6 and 7. The SUM() function returns 105 (30 + 35 + 40).

Finally processing gets and ends at row 8 of the partition. The rows used by the SUM() function are rows 6, 7, and 8. The SUM() function() returns 120 (35 + 40 + 45). There are no more partitions so the processing ends.

What if we want to specify several rows before and after the current row? The next clause does the trick. This was the first example in the chapter but let’s review it again and make one change.

ROWS BETWEEN n PRECEDING AND n FOLLOWING

In this case we want to include the current row, the prior row and two following rows. We could well specify two rows preceding and 3 rows following or any combination within the number of rows in the partition if we had more rows in the partition.

Please refer to Figure 1.6 below:

Diagram

Description automatically generated

***Figure 1.6 – Rows between n = 1 preceding and n = 2 following***

Here is the ROWS clause for this scenario:

ROWS BETWEEN 1 PRECEDING AND 2 FOLLOWING

Starting at row 1 of the partition the SUM() function can only use rows 1, 2 and 3 as there are no rows in the partition prior to row 1. The result calculated by the SUM() function is 45 (10 + 15 + 20).

Moving to the next row, row 2 of the partition the SUM() function can use rows 1, 2 (the current row), 3 and 4. The result calculated by the SUM() function is 70 (10 + 15 + 20 + 25).

Next the current row is row 3. This window frame uses row 2, 3 and only 4 so the SUM() function will calculate a total sum of 60 (15 + 20 + 25).

Finally, moving to row 4 (last row of the partition) the SUM() function can only use rows 3 and 4 as there are no more available rows following in the partition. The result is 45 (20 + 25).

Processing at the next partition continues in the same fashion, ignoring all rows from the prior partition.

We have pretty much examined most of the configurations for the ROWS clause. Next, we look at window frames defined by the RANGE clause which works at a logical level. It considers values instead of physical row position. Duplicate ORDER BY values will yield a strange behavior.

**RANGE Frame Definition**

We discussed how the ROWS clause works at the physical level by using values relative to the row positions they belong to. The RANGE clause operates at the logical level. It considers the values of the column instead of physical row position within the partition. It displays a strange (in my opinion) behavior. If it encounters duplicate values in the row sort column, it will add them all up and display the total in all rows with the duplicate ORDER BY column. Any values like moving totals will go out the window! (Pardon the pun.)

An example illustrates this scenario.

Suppose you have a small table with 5 columns: Rows, Year, Month, Amount and Running Total. The table is loaded with 12 rows for each year, let’s say 2010 and 2011 plus a duplicate month row giving us a total of 13 rows. Each row represents a month in the year but for the month of March two rows are inserted for the first year (that’s why we have 13 rows and not 12 rows). In other words, an extra amount value exists for the same month resulting in duplicate month numbers.

The RANGE clause will use all the rows with duplicate current month values to apply the window function to (in addition to any other rows specified in the RANGE clause).

Check out the partial table, table 1.2 below:

| **Row** | **Year** | **Month** | **Amount** | **Running Total** |
| --- | --- | --- | --- | --- |
| 1 | 2010 | 1 | 100.00 | 100.00 |
| 2 | 2010 | 2 | 100.00 | 200.00 |
| **3** | **2010** | **3** | **200.00** | **600.00** |
| **4** | **2010** | **3** | **200.00** | **600.00** |

***Table 1.2 – partial RANGE running totals***

When we get to row 3, the first of two duplicate months the window function (SUM() in this case) will include all prior rows plus add the values in rows 3 and the next row 4 to generate a total value of 600.00. This is displayed for both rows 3 and 4. Weird! One would expect a running total value of 400.00 for row 3.

Let’s try another one. If we apply the following range clause to calculate running totals:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

or

RANGE UNBOUNDED PRECEDING

We get interesting results. Check out the partial results in Figure 1.7 below:

Graphical user interface, table

Description automatically generated

***Figure 1.7 – Sales by Year and Month***

Everything works as expected until we get to the duplicate rows with a value of 5 (May) for the month. The window frame for this partition includes rows 1 thru 4 and the current row 5 but we also have a duplicate month value in row 6 which calculates the rolling total value 60. This value is displayed in both row 5 and 6. Aggregations continue for the rest of the months (not shown).

Let’s process rows in the opposite direction, Here is our next RANGE clause.

RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

RANGE UNBOUNDED FOLLOWING (will not work)

By the way, the second RANGE clause is not supported but I included it so you are aware that it will not work and generate an error. Please refer to figure 1.8 below:

Table

Description automatically generated

***Figure 1.8 – Range between current row and unbounded following***

Rows 5 and 6 represent two sales figures for the month of May. Since these are considered duplicates, row 5 displays a rolling value of 90 and row 6 also shows 90 instead of 80 as one would expect if they did not understand the behavior of this window frame declaration.

Notice by the way, how the rolling sum values decrease as the row processing advances from row 1 onwards. We started with 130 and ended with 10. Moving totals in reverse.

Putting it all together, here is a conceptual view that illustrates the generation of window frames as rows are processed when the following ROWS clause is used:

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

This is the first example in the chapter. Partition for type A rows is considered only. Please refer to figure 1.9 below:

Table

Description automatically generated

***Figure 1.9 – Window Frame and processed rows***

Both partitions are shown so you can see how the frame resets when it gets to the next partition.

The generated window frames, as processing goes row by row, shows which values are included for the window function to process within the partition. The first window and last window only have two rows to process but the second and third both have 3 rows to process, the prior, current, and next rows. The same pattern repeats itself for the second partition.

Let’s put our knowledge to the test by looking at a simple query based on the figures we just examined.

**Example 1**

We will start by creating a small test temporary table for our analysis.

The following listing shows the test table CREATE TABLE DDL (Data Declaration Language) statement created as a temporary table and one INSERT statement to load all the rows we wish to process.

Please refer to the partial listing 1.3 below:

***Listing 1.3 – Creating the test table***

CREATE TABLE #TestTable (

Row SMALLINT,

[Year]SMALLINT,

[Month]SMALLINT,

Amount DECIMAL(10,2)

);

INSERT INTO #TestTable VALUES

-- 2010

(1,2010,1,10),

(2,2010,2,10),

(3,2010,3,10),

(4,2010,4,10),

**(5,2010,5,10),**

**(6,2010,5,10),**

(7,2010,6,10),

(8,2010,7,10),

(9,2010,8,10),

(10,2010,9,10),

(11,2010,10,10),

(12,2010,11,10),

(13,2010,12,10),

-- 2011

(14,2011,1,10),

(15,2011,2,10),

(16,2011,3,10),

(17,2011,4,10),

**(18,2011,5,10),**

**(19,2011,5,10),**

(20,2011,6,10),

**(21,2011,7,10),**

**(22,2011,7,10),**

**(23,2011,7,10),**

(24,2011,8,10),

(25,2011,9,10),

(26,2011,10,10),

(27,2011,11,10),

(28,2011,12,10);

The CREATE statement is used to create a simple table with 4 columns, a Row column to identify the row numbers, a Year column and Month column for the calendar information and the Amount column to store a numerical value.

The column names Year and Month are not good names as they are reserved words in SQL Server but since this is a simple test table, I did not get too rigorous with naming standards. Also, I used them in the last two figures, so we are consistent. (At least for our simple examples.)

We wish to load two years’ worth of data with some duplicate months so we can see the difference between how the ROWS clause and the RANGE clause treats these situations.

Here is our first query. Let’s calculate some amount totals by year and month. We go forward with a RANGE clause that considers the current row and all following rows.

Please refer to the partial listing 1.4 below:

***Listing 1.4 – Range, current row, and unbounded following***

SELECT Row,[Year],[Month],Amount,

SUM(Amount) OVER (

PARTITION BY [Year]

ORDER BY [Month]

RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

) AS RollingSalesTotal

FROM #TestTable

ORDER BY [Year],[Month] ASC

GO

Notice the square brackets around the Year and Month column names. Using these will eliminate the colors SSMS uses to display reserved keywords. Leaving them out won’t hurt but the colors will be used in the query pane which makes you aware you are using reserved keywords. Don’t forget to drop the temporary table.

Here are the partial results so you can see the entire range of values for one year:

Please refer to figure 1.10 below:

Graphical user interface, table

Description automatically generated

***Figure 1.10 – Range between current row and unbounded following***

Looks like the totals are decreasing which makes sense as we are going forward from the current row being processed by the window function. Look at the familiar duplicate behavior in row 5 and 6 when both months are 5 (May). Both rows have the same value of 90.00.

When we get to December, in row 13 we have no more rows to process in the partition, so we only return the value 10.00 for the sum. This is the value for the current and last row in the partition.

In the second year partition we have two clusters of duplicate rows, one for the month of May and one for the month of July. Again, we see the behavior that is characteristic of the RANGE clause.

By the way, the following window frame clauses are not supported:

RANGE BETWEEN CURRENT ROW AND n FOLLOWING (will not work)

RANGE BETWEEN n PRECEEDING AND CURRENT ROW (will not work)

RANGE BETWEEN n PRECEDING AND n FOLLOWING (will not work)

These do not work, as the RANGE clause is a logical operation so specifying a number for the rows to include in the window frame is not allowed. If you try to use these, you will get this wonderful error message:

RANGE is only supported with UNBOUNDED and CURRENT ROW window frame delimiters

Pretty clear, don’t you think?

Last item we need to discuss in this chapter is the default behavior if we do not include window frame clauses in our OVER() clause. The behavior changes depending on how you include or not include the ORDER BY and PARTITION BY clause.

**ROWS & RANGE default behavior**

By now we understand the syntax of the OVER() clause and we know that it could include the following clauses:

* PARTITION BY clause
* ORDER BY clause
* ROW or RANGE clause
* None of the above (empty)

The first three clauses are optional. You can leave them out or include one or more as required. There are two scenarios to consider when applying (or not) these clauses.

**Scenario 1**

The default behavior of the window frames is dependent on whether the ORDER BY clause is included or not. There are two configurations to consider:

If the ORDER BY clause and the PARTITION BY clause are omitted and we do not include the window frame clause.

If the ORDER BY clause is omitted but the PARTITION BY clause is included and we do not include the window frame clause.

The default window frame behavior for both these cases is:

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

**Scenario 2**

On the other hand, if we include an ORDER BY clause the following two conditions also have a default window frame behavior:

If the ORDER BY clause is included but the PARTITION BY clause is omitted and we do not include the window frame clause.

If the ORDER BY clause is included and the PARTITION BY clause is included and we do not include the window frame clause.

The default window frame behavior for both these cases is:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Make sure you keep these default behaviors in mind as you start to develop queries that use the window functions. This is very important otherwise you will get some unexpected and possibly erroneous results. Your users will not be happy (is your resume up to date?).

The following table might help you remember these default behaviors. Please refer to table 1.3 below:

| **ORDER BY** | **PARTITION BY** | **DEFAULT FRAME** |
| --- | --- | --- |
| No | No | ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING |
| No | Yes | ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING |
| Yes | No | RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW |
| Yes | Yes | RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW |

***Table 1.3 – Window frame default behaviors***

With this new knowledge under our belts let’s check out the ROW and RANGE clauses with some more simple examples. Remember you can override default behavior by including the ROW or RANGE clause you need.

**ROWS Window Frame Examples**

Let’s start off with the ROWS clause. This clause you will use most of the time. The RANGE clause is a bit dodgy in my opinion and should be avoided.

**Data Set**

Prior to creating some queries, we need to create a small table and load it so we can practice. The table we create is called OverExample (clever name I might add). It is made up of three columns, OrderYear, OrderMonth and SalesAmount. Now we created this table in the beginning of the chapter but I include it here for easy reference.

Please refer to the partial listing 1.5 below:

***Listing 1.5 – Practice Table OverExample***

CREATE TABLE OverExample(

OrderYear SMALLINT,

OrderMonth SMALLINT,

SalesAmount DECIMAL(10,2)

);

INSERT INTO OverExample VALUES

(2010,1,10000.00),

(2010,2,10000.00),

(2010,2,10000.00),

--missing rows

(2010,8,10000.00),

(2010,8,10000.00),

(2010,9,10000.00),

(2010,10,10000.00),

(2010,11,10000.00),

(2010,12,10000.00),

-- 2011

(2011,1,10000.00),

(2011,2,10000.00),

(2011,2,10000.00),

--missing rows

(2011,10,10000.00),

(2011,11,10000.00),

(2011,12,10000.00);

GO

The INSERT statements load two years’ worth of data so we have enough rows to test out queries with the window functions and the OVER() clause. The SUM() aggregate function will be used to generate some rolling total calculations.

Please refer to Listing 1.6 below:

***Listing 1.6 - Scenario 1***

SELECT OrderYear,OrderMonth,SalesAmount,

SUM(SalesAmount) OVER (

) AS NPBNOB,

SUM(SalesAmount) OVER (

PARTITION BY OrderYear

) AS PBNOB,

SUM(SalesAmount) OVER (

PARTITION BY OrderYear

ORDER BY OrderMonth

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) AS PBOBUPUF

FROM OverExample;

GO

The SUM() function is used three times with different ORDER BY/PARTITION BY configurations. I also used column names that are meant to indicate the structure of the OVER() clause in terms of whether the ORDER BY, PARTITION BY and ROW/RANGE clauses are included. I did this so when you examine the results you can quickly remember what clauses were used. I know, weird names but you can easily remember the frame clauses we are using.

The first SUM() function has an empty OVER() clause (the name NPBNOB = No PARTITION BY, No ORDER BY). Recall that the default window frame behavior for this combination is:

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

The second SUM() function has an OVER() clause that includes a PARTITION BY but no ORDER BY so the default window frame behavior is also:

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

The column name is PBNOB (PARTITION BY, No ORDER BY).

This time two partitions are created, one for each year in the data set.

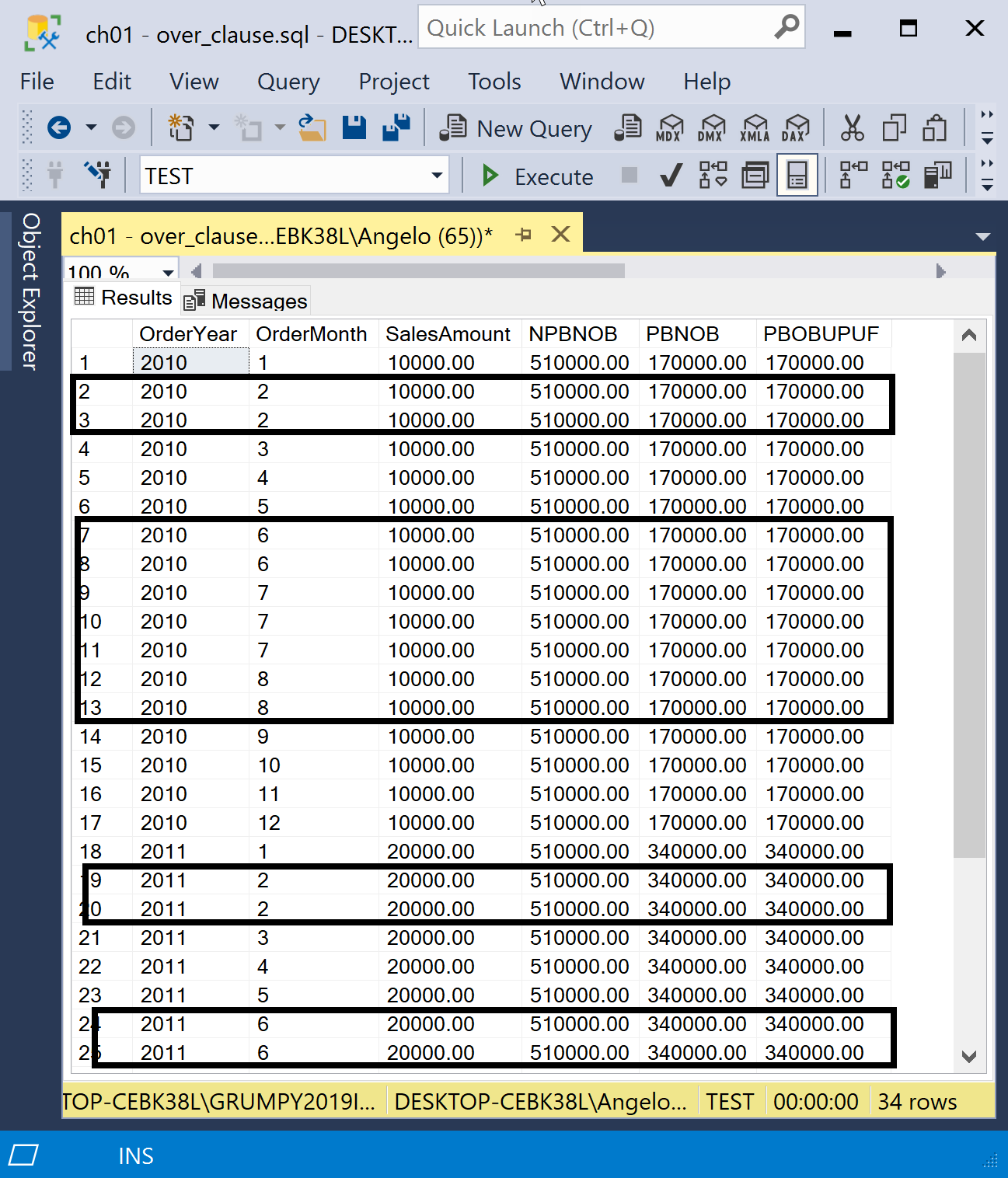
Last but not least the third SUM() function uses an OVER() clause that has both a PARTITION BY and ORDER BY clause. It also used a window frame ROWS clause. The column name is PBOBUPUF (PARTITION BY, ORDER BY, UNBOUNDED PRECEDING UNBOUNDED FOLLOWING).

The default frame behavior for this last column configuration is:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Looks like we can override this behavior by replacing the RANGE clause and including a ROWS clause. It can be done!

Let’s see what the results look like when we execute this query. Please refer to figure 1.11 below.



***Figure 1.11 – Scenario 1 results***

Interesting results. The first SUM()/OVER() clause combination simply displays the grand total for all the rows in the table for each row. Just as the default window frame behavior enforces.

The second and third SUM()/OVER() clause combinations partition the data set by year so the grand totals by year are displayed in each row. They appear to be identical. The window frame behavior is the same for both combinations:

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

That’s because for the second combination the default behavior kicked in and for the third combination, we overrode the RANGE behavior and included the ROWS clause. This behavior is enforced within each year partition. Let’s try another example.

**Example 2**

This time we check out some combinations of the SUM()/OVER() clause that include default window frame behaviors versus declared RANGE behaviors. (Keep in mind the default window frame behaviors we discussed earlier.)

Let’s see the results these combinations deliver. Please refer to Listing 1.7 below:

***Listing 1.7 – Scenario 2 - various default versus window frame clauses***

SELECT OrderYear,OrderMonth,SalesAmount,

SUM(SalesAmount) OVER (

ORDER BY OrderYear,OrderMonth

) AS NPBOB,

-- sames as PBOBRangeUPCR

SUM(SalesAmount) OVER (

PARTITION BY OrderYear

ORDER BY OrderMonth

) AS PBOB,

SUM(SalesAmount) OVER (

ORDER BY OrderYear,OrderMonth

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS NPBOBRangeUPCR,

-- same as PBOB

SUM(SalesAmount) OVER (

PARTITION BY OrderYear

ORDER BY OrderMonth

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS PBOBRangeUPCR

FROM OverExample;

GO

This time we added an ORDER BY clause to the first SUM()/OVER() combination, a PARTITION BY and ORDER BY clause in the second SUM()/OVER() combination and the third SUM()/OVER() combination which included an ORDER BY clause, no PARTITION BY clause but we included a RANGE window frame clause. A fourth SUM()/OVER() combination included both a PARTITION BY and ORDER BY clause plus a RANGE window frame clause. Results should be interesting.

Please refer to Figure 1.12 below

Table

Description automatically generated

***Figure 1.12 – More default versus range behavior***

At first glance, all the results seem to be the same! This is true for the first year’s partition, but once we get to the second year’s partition, values for the first and third columns keep incrementing while values for the second and fourth column are reset due to the new year partition.

Values for duplicate month entries have been framed with boxes. Column NPBOB has no PARTITION BY clause, but it has an ORDER BY clause, so the default window frame behavior defined by RANGE BETWEEN UNBOUNDED PRECEDING and CURRENT ROW kicks in. Once again notice how the rows with the duplicate month numbers result in the totals being the same. Month 6 (June) for 2010 has totals of 8000.00. Same for months 7 (July) and 8 (August).

The second SUM()/OVER() combination does have a PARTITION BY and ORDER BY clause so the default window frame behavior is:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

The PARTITION BY clause defines the partitions for each year and the default frame behavior causes the window function to calculate values by considering the current row and prior row values until duplicate month entries are encountered. If this is a running total, we would expect 7000.00 and 8000.00 but both entries for these rows have the higher value: 8000.00.

The third SUM()/OVER() combination in the query includes an ORDER BY clause but no PARTITION BY clause. A window frame specification is included:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

The results are the same as the prior combination but once we get to the partition for year 2011 the rolling sum just keeps going as a PARTITION BY clause was not included. Once again, notice the behavior when rows with the same month are encountered.

Last but not least the fourth SUM()/OVER() combination has a PARTITION BY, an ORDER BY and a RANGE window frame clause. This combination behaves just like the combination for the PBOB column.

We can now see how the default behaviors kick in for the various combinations of the ORDER BY and PARTITION BY clauses. When a window frame specification is included, it behaves just like the default behavior in the other columns. Keep these in mind when you start creating window function based queries.

**Example 3**

The next example compares ROWS window frames specifications versus the RANGE window frame specification.

Please refer to Listing 1.8 below:

***Listing 1.8 – ROWS versus RANGE comparison***

SELECT OrderYear,OrderMonth,SalesAmount,

SUM(SalesAmount) OVER (

PARTITION BY OrderYear

ORDER BY OrderMonth

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS POBRowsUPCR,

SUM(SalesAmount) OVER (

PARTITION BY OrderYear

ORDER BY OrderMonth

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS PBOBRangeUPCR

FROM OverExample;

GO

The SUM() function in both cases have identical PARTITION BY and ORDER BY clauses. The first has a ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW clause while the second combination has RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW clause.

What do you think the results will be?

Please refer to Figure 1.13 below

Table

Description automatically generated

***Figure 1.13 – ROWS versus RANGE, UNBOUNDED PRECEDING AND CURRENT ROW***

The results are almost the same but the SUM()/OVER() combination that uses the RANGE window frame specification will generate the same values for any rows having duplicate ORDER BY values. As we discussed earlier, this clause works at a logical level (versus physical for ROWS) it will add up all the duplicate values and post the same results in each duplicate row.

**Example 4**

In this next example we will look at two simple queries. The only difference is that the first query includes an ORDER BY clause that calls out column names, while the second query uses a simple SELECT query instead of the column names in the ORDER BY clause.

Why do we do this? Because using a simple query will eliminate a possibly expensive sort step in the estimated query plan. This technique is used when you have only one large partition. The query you need to use is a scalar query, which means it will only return one value so you cannot process multiple partitions.

Let’s check out the query with the column names first.

Please refer to listing 1.9a below:

***Listing 1.9a – ORDER BY without subquery***

WITH YearQtrSales (

SalesYear,

SalesQtr,

SalesMonth,

SalesTotal

)

AS

(

SELECT

SalesYear,

SalesQtr,

SalesMonth,

SalesTotal

FROM dbo.TestSales

)

SELECT

SalesYear,

SalesQtr,

SalesMonth,

SalesTotal,

SUM(SalesTotal) OVER(

ORDER BY SalesYear,SalesQtr,SalesMonth

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS RollMonthlySales1

FROM YearQtrSales

ORDER BY

SalesYear,

SalesQtr,

SalesMonth

GO

As can be seen we have no PARTITION BY clause and an ORDER BY clause that sorts by three columns. This configuration will give us rolling totals by month for both years. In other words, when we get to the second year in the results set the totals do not reset but keep on incrementing by month till the end of the data set.

The second query replaces the column names in the ORDER BY clause with a simple SELECT statement: SELECT (1).

Please refer to listing 1.9b below:

***Listing 1.9b – ORDER BY with subquery***

WITH YearQtrSales (

SalesYear,

SalesQtr,

SalesMonth,

SalesTotal

)

AS

(

SELECT

SalesYear,

SalesQtr,

SalesMonth,

SalesTotal

FROM dbo.TestSales

)

SELECT

SalesYear,

SalesQtr,

SalesMonth,

SalesTotal,

SUM(SalesTotal) OVER(

**ORDER BY (SELECT (1))**

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS RollMonthlySales2

FROM YearQtrSales

ORDER BY

SalesYear,

SalesQtr,

SalesMonth

GO

Same query as the previous one. Let’s compare execution plans.

Please refer to figure 1.14 below:

Graphical user interface, application

Description automatically generated

***Figure 1.14 – Comparing execution plans***

As can be seen in the left-hand query plan we have a pretty expensive sort task with a cost of 77%. The query plan on the right does not have this task so it should be faster in terms of performance. Let’s check out the results for both queries, side by side.

Please refer to figure 1.15 below:

Graphical user interface

Description automatically generated

***Figure 1.15 – Comparing Results.***

Results match. Notice how the rolling totals keep incrementing even when we start a new year. This type of configuration where we use a subquery is usually good when you only deal with one single partition with a large volume of rows. Eliminating a sort step on a query results set of a couple of hundred of thousand or more rows can increase performance.

**Example 5**

Our last example is a simple one that illustrates a new feature available with SQL Server 2022. As of this writing it is available as an evaluation release. It allows you to specify the PARTITION BY, ORDER BY and RANGE/ROW clauses within a set of parentheses and give it a name. It’s called the named window clause. Here is a simple example.

Please refer to Listing 1.10 below:

***Listing 1.10 – Named Windows***

SELECT OrderYear

,OrderMonth

,SUM(SalesAmount) OVER **SalesWindow** AS TotalSales

FROM dbo.OverExample

WINDOW **SalesWindow** AS (

PARTITION BY OrderYear

ORDER BY OrderMonth

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

)

GO

Notice how the named window code appears at the end of the query. The keyword WINDOW is followed by the name you wish to give this window and then you can add the PARTITION BY, ORDER BY and RANGE/ROW clauses between the parentheses as we just discussed in this chapter.

The OVER() clause still appears after the window function but it simply refers to the name. Is this value added? I do not know but I would have preferred something like:

SUM(SalesAmount) OVER WINDOW (

PARTITION BY OrderYear

ORDER BY OrderMonth

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS …

At least it is clear that we are creating a window into the partition. But that’s me.

On the other hand, if the OVER() clause is to be used in multiple columns then the SQL Server 2022 version just discussed makes more sense. Yes, I think this is better.

Let’s look at the results for this last query.

Please refer to Figure 1.16 below:

Table

Description automatically generated

***Figure 1.16 – Named WINDOW SQL Server 2022 enhancement***

No surprises here. Works just like the prior examples. Same behavior when duplicate ORDER BY column values exist. Go to the Microsoft SQL Server web site to download the latest version. Make sure to check out what’s new in this version.

**Summary**

We’ve covered the window functions we will use in this book. More importantly, we discussed the OVER() clause and how to set up partitions and window frames to control how the window functions are applied to the data sets, partitions and window frames.

We discussed how window frames have default behavior depending on the combination of PARTITION BY and ORDER BY clause within the OVER() clauses.

We also looked at some basic diagrams to illustrate how window frames affect the window function processing flow and under what conditions default window frames kick in.

Next, we briefly discussed how you can use subqueries with the PARTITION BY and OVER clauses. This is of value as using subqueries eliminates a sort step and can make the overall query perform faster. This was demonstrated by looking at query plans for queries that used subqueries and queries that did not. Some tricks were tested to see how you can retrieve multiple values but package them as a single value with the STRING\_AGG() function.

Lastly, we looked at some simple examples to put our knowledge to work and briefly discussed a new SQL Server 2022 feature called named windows.

If you are unfamiliar with how window functions such as RANK() or other functions work, now is a good time to check out appendix 1 which gives you a brief description on what these functions do. These will be used many times in the remaining chapters in the book.