SELECT - OVER Clause (Transact-SQL)

08/11/2017 • 15 minutes to read • +3

In this article

Syntax
Arguments
General Remarks
Limitations and Restrictions
Examples
Examples: Parallel Data Warehouse
See Also

APPLIES TO: ✓ SQL Server ✓ Azure SQL Database ✓ Azure Synapse Analytics (SQL DW) ✓ Parallel Data Warehouse

Determines the partitioning and ordering of a rowset before the associated window function is applied. That is, the OVER clause defines a window or user-specified set of rows within a query result set. A window function then computes a value for each row in the window. You can use the OVER clause with functions to compute aggregated values such as moving averages, cumulative aggregates, running totals, or a top N per group results.

- Ranking functions
- Aggregate functions
- Analytic functions
- NEXT VALUE FOR function

Transact-SQL Syntax Conventions

Syntax

Copy

```
-- Syntax for SQL Server, Azure SQL Database, and Azure SQL Data
Warehouse

OVER (
       [ <PARTITION BY clause> ]
       [ <ORDER BY clause> ]
       [ <ROW or RANGE clause> ]
      )

<PARTITION BY clause> ::=
PARTITION BY value_expression , ... [ n ]

<ORDER BY clause> ::=
ORDER BY order_by_expression
    [ COLLATE collation_name ]
    [ ASC | DESC ]
    [ ,...n ]

<ROW or RANGE clause> ::=
{ ROWS | RANGE } <window frame extent>

<window frame extent> ::=
{   <window frame preceding>
  | <window frame between>
}
<window frame between> ::=
  BETWEEN <window frame bound> AND <window frame bound>

<window frame bound> ::=
{   <window frame preceding>
  | <window frame following>
}

<window frame preceding> ::=
{
    UNBOUNDED PRECEDING
  | <unsigned_value_specification> PRECEDING
  | CURRENT ROW
}

<window frame following> ::=
{
    UNBOUNDED FOLLOWING
  | <unsigned_value_specification> FOLLOWING
  | CURRENT ROW
}

<unsigned value specification> ::=
{  <unsigned integer literal> }
```

Copy

```
-- Syntax for Parallel Data Warehouse

OVER ( [ PARTITION BY value_expression ] [ order_by_clause ] )
```

Arguments

PARTITION BY
Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.

value_expression
Specifies the column by which the rowset is partitioned. *value_expression* can only refer to columns made available by the FROM clause. *value_expression* cannot refer to expressions or aliases in the select list. *value_expression* can be a column expression, scalar subquery, scalar function, or user-defined variable.

<ORDER BY clause>
Defines the logical order of the rows within each partition of the result set. That is, it specifies the logical order in which the window functioncalculation is performed.

order_by_expression
Specifies a column or expression on which to sort. *order_by_expression* can only refer to columns made available by the FROM clause. An integer cannot be specified to represent a column name or alias.

COLLATE *collation_name*
Specifies that the ORDER BY operation should be performed according to the collation specified in *collation_name*. *collation_name* can be either a Windows collation name or a SQL collation name. For more information, see Collation and Unicode Support. COLLATE is applicable only for columns of type **char**, **varchar**,

nchar, and **nvarchar**.

ASC | DESC
Specifies that the values in the specified column should be sorted in ascending or descending order. ASC is the default sort order. Null values are treated as the lowest possible values.

ROWS | RANGE
Applies to: SQL Server 2012 (11.x) and later.

Further limits the rows within the partition by specifying start and end points within the partition. This is done by specifying a range of rows with respect to the current row either by logical association or physical association. Physical association is achieved by using the ROWS clause.

The ROWS clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row. Alternatively, the RANGE clause logically limits the rows within a partition by specifying a range of values with respect to the value in the current row. Preceding and following rows are defined based on the ordering in the ORDER BY clause. The window frame "RANGE ... CURRENT ROW ..." includes all rows that have the same values in the ORDER BY expression as the current row. For example, ROWS BETWEEN 2 PRECEDING AND CURRENT ROW means that the window of rows that the function operates on is three rows in size, starting with 2 rows preceding until and including the current row.

Note

ROWS or RANGE requires that the ORDER BY clause be specified. If ORDER BY contains multiple order expressions, CURRENT ROW FOR RANGE considers all columns in the ORDER BY list when determining the current row.

UNBOUNDED PRECEDING
Applies to: SQL Server 2012 (11.x) and later.

Specifies that the window starts at the first row of the partition. UNBOUNDED PRECEDING can only be specified as window starting point.

<unsigned value specification> PRECEDING

Specified with <unsigned value specification>to indicate the number of rows or values to precede the current row. This specification is not allowed for RANGE.

CURRENT ROW
Applies to: SQL Server 2012 (11.x) and later.

Specifies that the window starts or ends at the current row when used with ROWS or the current value when used with RANGE. CURRENT ROW can be specified as both a starting and ending point.

BETWEEN <window frame bound > AND <window frame bound >
Applies to: SQL Server 2012 (11.x) and later.

Used with either ROWS or RANGE to specify the lower (starting) and upper (ending) boundary points of the window. <window frame bound> defines the boundary starting point and <window frame bound> defines the boundary end point. The upper bound cannot be smaller than the lower bound.

UNBOUNDED FOLLOWING
Applies to: SQL Server 2012 (11.x) and later.

Specifies that the window ends at the last row of the partition. UNBOUNDED FOLLOWING can only be specified as a window end point. For example RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING defines a window that starts with the current row and ends with the last row of the partition.

<unsigned value specification> FOLLOWING
Specified with <unsigned value specification> to indicate the number of rows or values to follow the current row. When <unsigned value specification> FOLLOWING is specified as the window starting point, the ending point must be <unsigned value specification>FOLLOWING. For example, ROWS BETWEEN 2 FOLLOWING AND 10 FOLLOWING defines a window that starts with the second row that follows the current row and ends with the tenth row that follows the current row. This specification is not allowed for RANGE.

unsigned integer literal
Applies to: SQL Server 2012 (11.x) and later.

Is a positive integer literal (including 0) that specifies the number of rows or values to precede or follow the current row or value. This specification is valid only for ROWS.

General Remarks

More than one window function can be used in a single query with a single FROM clause. The OVER clause for each function can differ in partitioning and ordering.

If PARTITION BY is not specified, the function treats all rows of the query result set as a single group.

Important!

If ROWS/RANGE is specified and <window frame preceding> is used for <window frame extent> (short syntax) then this specification is used for the window frame boundary starting point and CURRENT ROW is used for the boundary ending point. For example "ROWS 5 PRECEDING" is equal to "ROWS BETWEEN 5 PRECEDING AND CURRENT ROW".

Note

If ORDER BY is not specified entire partition is used for a window frame. This applies only to functions that do not require ORDER BY clause. If ROWS/RANGE is not specified but ORDER BY is specified, RANGE UNBOUNDED PRECEDING AND CURRENT ROW is used as default for window frame. This applies only to functions that have can accept optional ROWS/RANGE specification. For example, ranking functions cannot accept ROWS/RANGE, therefore this window frame is not applied even though ORDER BY is present and ROWS/RANGE is not.

Limitations and Restrictions

The OVER clause cannot be used with the CHECKSUM aggregate function.

RANGE cannot be used with <unsigned value specification> PRECEDING or <unsigned value specification> FOLLOWING.

Depending on the ranking, aggregate, or analytic function used with the OVER clause, <ORDER BY clause> and/or the <ROWS and RANGE clause> may not be supported.

Examples

A. Using the OVER clause with the ROW_NUMBER function

The following example shows using the OVER clause with ROW_NUMBER function to display a row number for each row within a partition. The ORDER BY clause specified in the OVER clause orders the rows in each partition by the column `SalesYTD`. The ORDER BY clause in the SELECT statement determines the order in which the entire query result set is returned.

SQL

Copy

```sql
USE AdventureWorks2012;
GO
SELECT ROW_NUMBER() OVER(PARTITION BY PostalCode ORDER BY SalesYTD
DESC) AS "Row Number",
    p.LastName, s.SalesYTD, a.PostalCode
FROM Sales.SalesPerson AS s
    INNER JOIN Person.Person AS p
        ON s.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.Address AS a
        ON a.AddressID = p.BusinessEntityID
WHERE TerritoryID IS NOT NULL
    AND SalesYTD <> 0
ORDER BY PostalCode;
GO
```

Here is the result set.

Copy

```
Row Number      LastName                 SalesYTD              
PostalCode
--------------- ------------------------ --------------------- -----
-----
1               Mitchell                 4251368.5497          98027
2               Blythe                   3763178.1787          98027
```

```
3               Carson                  3189418.3662          98027
4               Reiter                  2315185.611           98027
5               Vargas                  1453719.4653          98027
6               Ansman-Wolfe            1352577.1325          98027
1               Pak                     4116871.2277          98055
2               Varkey Chudukatil       3121616.3202          98055
3               Saraiva                 2604540.7172          98055
4               Ito                     2458535.6169          98055
5               Valdez                  1827066.7118          98055
6               Mensa-Annan             1576562.1966          98055
7               Campbell                1573012.9383          98055
8               Tsoflias                1421810.9242          98055
```

B. Using the OVER clause with aggregate functions

The following example uses the `OVER` clause with aggregate functions over all rows returned by the query. In this example, using the `OVER` clause is more efficient than using subqueries to derive the aggregate values.

SQL Copy

```sql
USE AdventureWorks2012;
GO
SELECT SalesOrderID, ProductID, OrderQty
    ,SUM(OrderQty) OVER(PARTITION BY SalesOrderID) AS Total
    ,AVG(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Avg"
    ,COUNT(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Count"
    ,MIN(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Min"
    ,MAX(OrderQty) OVER(PARTITION BY SalesOrderID) AS "Max"
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN(43659,43664);
GO
```

Here is the result set.

Copy

```
SalesOrderID ProductID   OrderQty Total       Avg         Count
Min    Max
------------ ----------- -------- ----------- ----------- ---------
-- ------ ------
43659        776         1        26          2           12
1      6
```

```
43659        777          3          26           2            12
1      6
43659        778          1          26           2            12
1      6
43659        771          1          26           2            12
1      6
43659        772          1          26           2            12
1      6
43659        773          2          26           2            12
1      6
43659        774          1          26           2            12
1      6
43659        714          3          26           2            12
1      6
43659        716          1          26           2            12
1      6
43659        709          6          26           2            12
1      6
43659        712          2          26           2            12
1      6
43659        711          4          26           2            12
1      6
43664        772          1          14           1            8
1      4
43664        775          4          14           1            8
1      4
43664        714          1          14           1            8
1      4
43664        716          1          14           1            8
1      4
43664        777          2          14           1            8
1      4
43664        771          3          14           1            8
1      4
43664        773          1          14           1            8
1      4
43664        778          1          14           1            8
1      4
```

The following example shows using the `OVER` clause with an aggregate function in a calculated value.

SQL Copy

```sql
USE AdventureWorks2012;
GO
```

```
SELECT SalesOrderID, ProductID, OrderQty
    ,SUM(OrderQty) OVER(PARTITION BY SalesOrderID) AS Total
    ,CAST(1. * OrderQty / SUM(OrderQty) OVER(PARTITION BY SalesOr-
derID)
        *100 AS DECIMAL(5,2))AS "Percent by ProductID"
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN(43659,43664);
GO
```

Here is the result set. Notice that the aggregates are calculated by `SalesOrderID` and the `Percent by ProductID` is calculated for each line of each `SalesOrderID`.

Copy

```
SalesOrderID ProductID   OrderQty Total       Percent by ProductID
------------ ----------- -------- ----------- ---------------------
------------------
43659        776         1        26          3.85
43659        777         3        26          11.54
43659        778         1        26          3.85
43659        771         1        26          3.85
43659        772         1        26          3.85
43659        773         2        26          7.69
43659        774         1        26          3.85
43659        714         3        26          11.54
43659        716         1        26          3.85
43659        709         6        26          23.08
43659        712         2        26          7.69
43659        711         4        26          15.38
43664        772         1        14          7.14
43664        775         4        14          28.57
43664        714         1        14          7.14
43664        716         1        14          7.14
43664        777         2        14          14.29
43664        771         3        14          21.4
43664        773         1        14          7.14
43664        778         1        14          7.14

 (20 row(s) affected)
```

C. Producing a moving average and cumulative total

The following example uses the AVG and SUM functions with the OVER clause to provide a moving average and cumulative total of yearly sales for each territory in the `Sales.SalesPerson` table. The data is partitioned by `TerritoryID` and logically ordered by `SalesYTD`. This means that the AVG function is computed for each territory based on the sales year. Notice that for `TerritoryID` 1, there are two rows for sales year 2005 representing the two sales people with sales that year. The average sales for these two rows is computed and then the third row representing sales for the year 2006 is included in the computation.

SQL Copy

```sql
USE AdventureWorks2012;
GO
SELECT BusinessEntityID, TerritoryID
   ,DATEPART(yy,ModifiedDate) AS SalesYear
   ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
   ,CONVERT(varchar(20),AVG(SalesYTD) OVER (PARTITION BY Territo-
ryID
                                            ORDER BY
DATEPART(yy,ModifiedDate)
                                           ),1) AS MovingAvg
   ,CONVERT(varchar(20),SUM(SalesYTD) OVER (PARTITION BY Territo-
ryID
                                            ORDER BY
DATEPART(yy,ModifiedDate)
                                            ),1) AS CumulativeTotal
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5
ORDER BY TerritoryID,SalesYear;
```

Here is the result set.

Copy

```
BusinessEntityID TerritoryID SalesYear   SalesYTD
MovingAvg            CumulativeTotal
---------------- ----------- ----------- -------------------- -----
--------------- --------------------
274              NULL        2005        559,697.56
559,697.56           559,697.56
287              NULL        2006        519,905.93
539,801.75           1,079,603.50
285              NULL        2007        172,524.45
417,375.98           1,252,127.95
283              1           2005        1,573,012.94
1,462,795.04         2,925,590.07
```

```
280                1            2005        1,352,577.13
1,462,795.04          2,925,590.07
284                1            2006        1,576,562.20
1,500,717.42          4,502,152.27
275                2            2005        3,763,178.18
3,763,178.18          3,763,178.18
277                3            2005        3,189,418.37
3,189,418.37          3,189,418.37
276                4            2005        4,251,368.55
3,354,952.08          6,709,904.17
281                4            2005        2,458,535.62
3,354,952.08          6,709,904.17

(10 row(s) affected)
```

In this example, the OVER clause does not include PARTITION BY. This means that the function will be applied to all rows returned by the query. The ORDER BY clause specified in the OVER clause determines the logical order to which the AVG function is applied. The query returns a moving average of sales by year for all sales territories specified in the WHERE clause. The ORDER BY clause specified in the SELECT statement determines the order in which the rows of the query are displayed.

SQL Copy

```sql
SELECT BusinessEntityID, TerritoryID
   ,DATEPART(yy,ModifiedDate) AS SalesYear
   ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
   ,CONVERT(varchar(20),AVG(SalesYTD) OVER (ORDER BY DATEPART(yy,ModifiedDate)
                                            ),1) AS MovingAvg
   ,CONVERT(varchar(20),SUM(SalesYTD) OVER (ORDER BY DATEPART(yy,ModifiedDate)
                                            ),1) AS CumulativeTotal
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5
ORDER BY SalesYear;
```

Here is the result set.

Copy

```
BusinessEntityID TerritoryID SalesYear   SalesYTD
```

```
MovingAvg          CumulativeTotal
---------------- ----------- ----------- -------------------- -----
--------------- --------------------
274              NULL        2005        559,697.56
2,449,684.05         17,147,788.35
275              2           2005        3,763,178.18
2,449,684.05         17,147,788.35
276              4           2005        4,251,368.55
2,449,684.05         17,147,788.35
277              3           2005        3,189,418.37
2,449,684.05         17,147,788.35
280              1           2005        1,352,577.13
2,449,684.05         17,147,788.35
281              4           2005        2,458,535.62
2,449,684.05         17,147,788.35
283              1           2005        1,573,012.94
2,449,684.05         17,147,788.35
284              1           2006        1,576,562.20
2,138,250.72         19,244,256.47
287              NULL        2006        519,905.93
2,138,250.72         19,244,256.47
285              NULL        2007        172,524.45
1,941,678.09         19,416,780.93
(10 row(s) affected)
```

D. Specifying the ROWS clause

Applies to: SQL Server 2012 (11.x) and later.

The following example uses the ROWS clause to define a window over which the rows are computed as the current row and the *N* number of rows that follow (1 row in this example).

SQL

Copy

```sql
SELECT BusinessEntityID, TerritoryID
    ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
    ,DATEPART(yy,ModifiedDate) AS SalesYear
    ,CONVERT(varchar(20),SUM(SalesYTD) OVER (PARTITION BY Territo-
ryID
                                             ORDER BY
DATEPART(yy,ModifiedDate)
                                             ROWS BETWEEN CURRENT
ROW AND 1 FOLLOWING ),1) AS CumulativeTotal
FROM Sales.SalesPerson
```

```sql
WHERE TerritoryID IS NULL OR TerritoryID < 5;
```

Here is the result set.

Copy

```
BusinessEntityID TerritoryID SalesYTD             SalesYear   Cumu-
lativeTotal
---------------- ----------- -------------------- ----------- -----
---------------
274              NULL        559,697.56           2005
1,079,603.50
287              NULL        519,905.93           2006
692,430.38
285              NULL        172,524.45           2007
172,524.45
283              1           1,573,012.94         2005
2,925,590.07
280              1           1,352,577.13         2005
2,929,139.33
284              1           1,576,562.20         2006
1,576,562.20
275              2           3,763,178.18         2005
3,763,178.18
277              3           3,189,418.37         2005
3,189,418.37
276              4           4,251,368.55         2005
6,709,904.17
281              4           2,458,535.62         2005
2,458,535.62
```

In the following example, the ROWS clause is specified with UNBOUNDED PRECEDING. The result is that the window starts at the first row of the partition.

SQL Copy

```sql
SELECT BusinessEntityID, TerritoryID
    ,CONVERT(varchar(20),SalesYTD,1) AS  SalesYTD
    ,DATEPART(yy,ModifiedDate) AS SalesYear
    ,CONVERT(varchar(20),SUM(SalesYTD) OVER (PARTITION BY Territo-
ryID
                                             ORDER BY
DATEPART(yy,ModifiedDate)
                                             ROWS UNBOUNDED PRECED-
ING),1) AS CumulativeTotal
```

```
FROM Sales.SalesPerson
WHERE TerritoryID IS NULL OR TerritoryID < 5;
```

Here is the result set.

Copy

```
BusinessEntityID TerritoryID SalesYTD             SalesYear   Cumu-
lativeTotal
---------------- ----------- -------------------- ----------- -----
---------------
274              NULL        559,697.56           2005
559,697.56
287              NULL        519,905.93           2006
1,079,603.50
285              NULL        172,524.45           2007
1,252,127.95
283              1           1,573,012.94         2005
1,573,012.94
280              1           1,352,577.13         2005
2,925,590.07
284              1           1,576,562.20         2006
4,502,152.27
275              2           3,763,178.18         2005
3,763,178.18
277              3           3,189,418.37         2005
3,189,418.37
276              4           4,251,368.55         2005
4,251,368.55
281              4           2,458,535.62         2005
6,709,904.17
```

Examples: Parallel Data Warehouse

E. Using the OVER clause with the ROW_NUMBER function

The following example returns the ROW_NUMBER for sales representatives based on their assigned sales quota.

SQL Copy

```
-- Uses AdventureWorks

SELECT ROW_NUMBER() OVER(ORDER BY SUM(SalesAmountQuota) DESC) AS
RowNumber,
    FirstName, LastName,
CONVERT(varchar(13), SUM(SalesAmountQuota),1) AS SalesQuota
FROM dbo.DimEmployee AS e
INNER JOIN dbo.FactSalesQuota AS sq
    ON e.EmployeeKey = sq.EmployeeKey
WHERE e.SalesPersonFlag = 1
GROUP BY LastName, FirstName;
```

Here is a partial result set.

Copy

```
RowNumber  FirstName  LastName            SalesQuota
---------  ---------  ------------------  -------------
1          Jillian    Carson              12,198,000.00
2          Linda      Mitchell            11,786,000.00
3          Michael    Blythe              11,162,000.00
4          Jae        Pak                 10,514,000.00
```

F. Using the OVER clause with aggregate functions

The following examples show using the OVER clause with aggregate functions. In this example, using the OVER clause is more efficient than using subqueries.

SQL Copy

```
-- Uses AdventureWorks

SELECT SalesOrderNumber AS OrderNumber, ProductKey,
       OrderQuantity AS Qty,
       SUM(OrderQuantity) OVER(PARTITION BY SalesOrderNumber) AS
Total,
       AVG(OrderQuantity) OVER(PARTITION BY SalesOrderNumber) AS
Avg,
       COUNT(OrderQuantity) OVER(PARTITION BY SalesOrderNumber) AS
Count,
       MIN(OrderQuantity) OVER(PARTITION BY SalesOrderNumber) AS
Min,
       MAX(OrderQuantity) OVER(PARTITION BY SalesOrderNumber) AS
```

```
Max
FROM dbo.FactResellerSales
WHERE SalesOrderNumber IN(N'SO43659',N'SO43664') AND
      ProductKey LIKE '2%'
ORDER BY SalesOrderNumber,ProductKey;
```

Here is the result set.

Copy

```
OrderNumber  Product  Qty  Total  Avg  Count  Min  Max
-----------  -------  ---  -----  ---  -----  ---  ---
SO43659      218      6    16     3    5      1    6
SO43659      220      4    16     3    5      1    6
SO43659      223      2    16     3    5      1    6
SO43659      229      3    16     3    5      1    6
SO43659      235      1    16     3    5      1    6
SO43664      229      1     2     1    2      1    1
SO43664      235      1     2     1    2      1    1
```

The following example shows using the OVER clause with an aggregate function in a calculated value. Notice that the aggregates are calculated by `SalesOrderNumber` and the percentage of the total sales order is calculated for each line of each `SalesOrderNumber`.

SQL Copy

```
-- Uses AdventureWorks

SELECT SalesOrderNumber AS OrderNumber, ProductKey AS Product,
       OrderQuantity AS Qty,
       SUM(OrderQuantity) OVER(PARTITION BY SalesOrderNumber) AS
Total,
       CAST(1. * OrderQuantity / SUM(OrderQuantity)
        OVER(PARTITION BY SalesOrderNumber)
            *100 AS DECIMAL(5,2)) AS PctByProduct
FROM dbo.FactResellerSales
WHERE SalesOrderNumber IN(N'SO43659',N'SO43664') AND
      ProductKey LIKE '2%'
ORDER BY SalesOrderNumber,ProductKey;
```

The first start of this result set is:

Copy

```
OrderNumber  Product  Qty  Total  PctByProduct
-----------  -------  ---  -----  ------------
SO43659      218      6    16     37.50
SO43659      220      4    16     25.00
SO43659      223      2    16     12.50
SO43659      229      2    16     18.75
```

See Also

Aggregate Functions (Transact-SQL)
Analytic Functions (Transact-SQL)
Excellent blog post about window functions and OVER, on sqlmag.com, by Itzik Ben-Gan

Is this page helpful?

Yes No