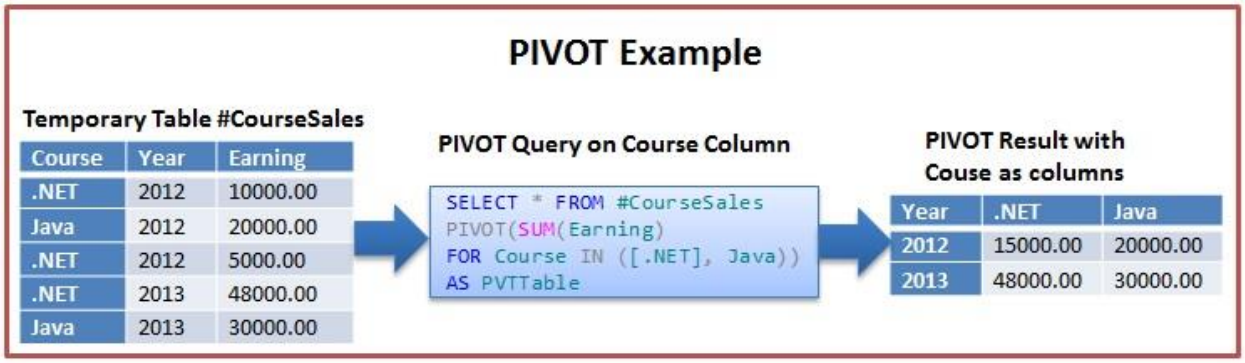
PIVOT and UNPIVOT

You can use the PIVOT and UNPIVOT relational operators to change a table-valued expression into another table.
PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output, and performs aggregations where they are required on any remaining column values that are wanted in the final output.

UNPIVOT performs the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.



PIVOT Example

Temporary Table #CourseSales

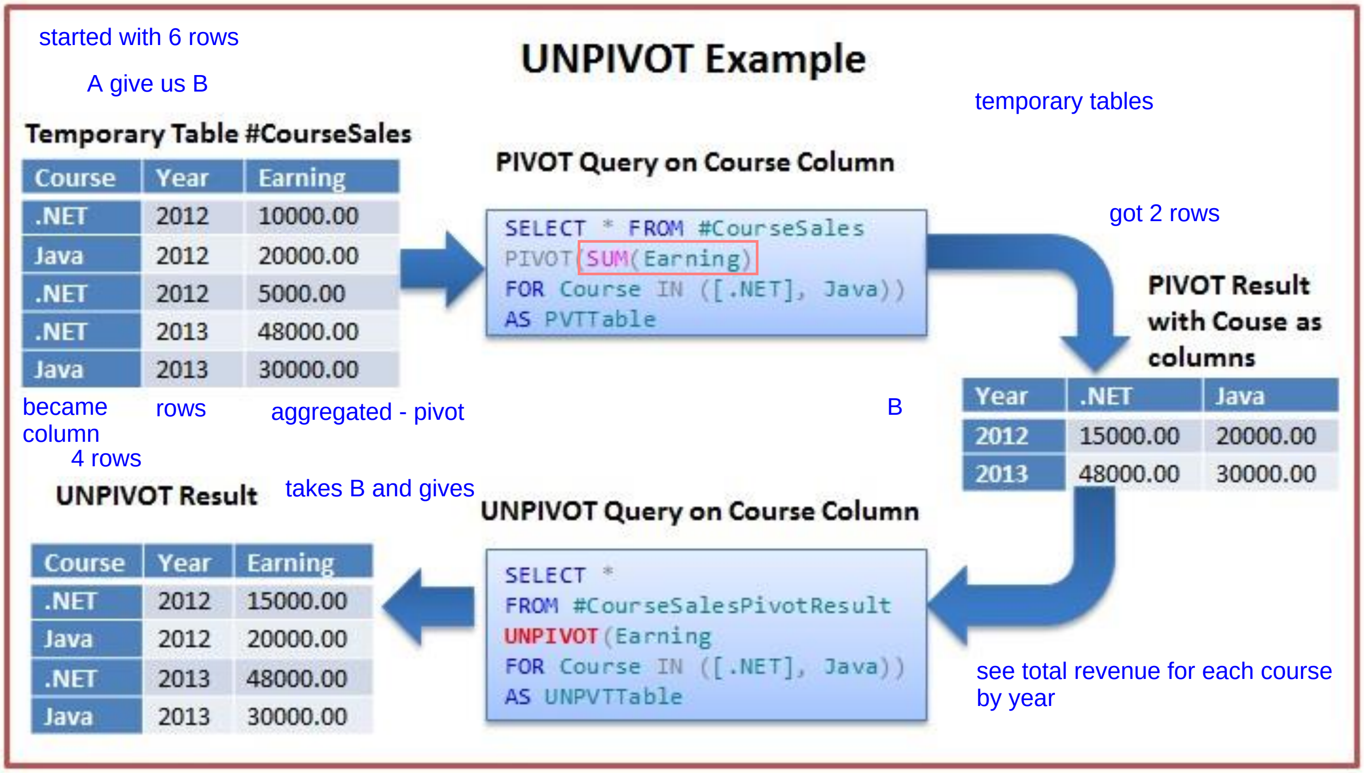
Course	Year	Earning
.NET	2012	10000.00
Java	2012	20000.00
.NET	2012	5000.00
.NET	2013	48000.00
Java	2013	30000.00

PIVOT Query on Course Column

```
SELECT * FROM #CourseSales
PIVOT(SUM(Earning)
FOR Course IN ([.NET], Java))
AS PVTTable
```

PIVOT Result with Couse as columns

Year	.NET	Java
2012	15000.00	20000.00
2013	48000.00	30000.00



started with 6 rows

A give us B

UNPIVOT Example

temporary tables

Temporary Table #CourseSales

Course	Year	Earning
.NET	2012	10000.00
Java	2012	20000.00
.NET	2012	5000.00
.NET	2013	48000.00
Java	2013	30000.00

PIVOT Query on Course Column

```
SELECT * FROM #CourseSales
PIVOT(SUM(Earning)
FOR Course IN ([.NET], Java))
AS PVTTable
```

got 2 rows

PIVOT Result with Couse as columns

Year	.NET	Java
2012	15000.00	20000.00
2013	48000.00	30000.00

became column rows aggregated - pivot B

4 rows

UNPIVOT Result

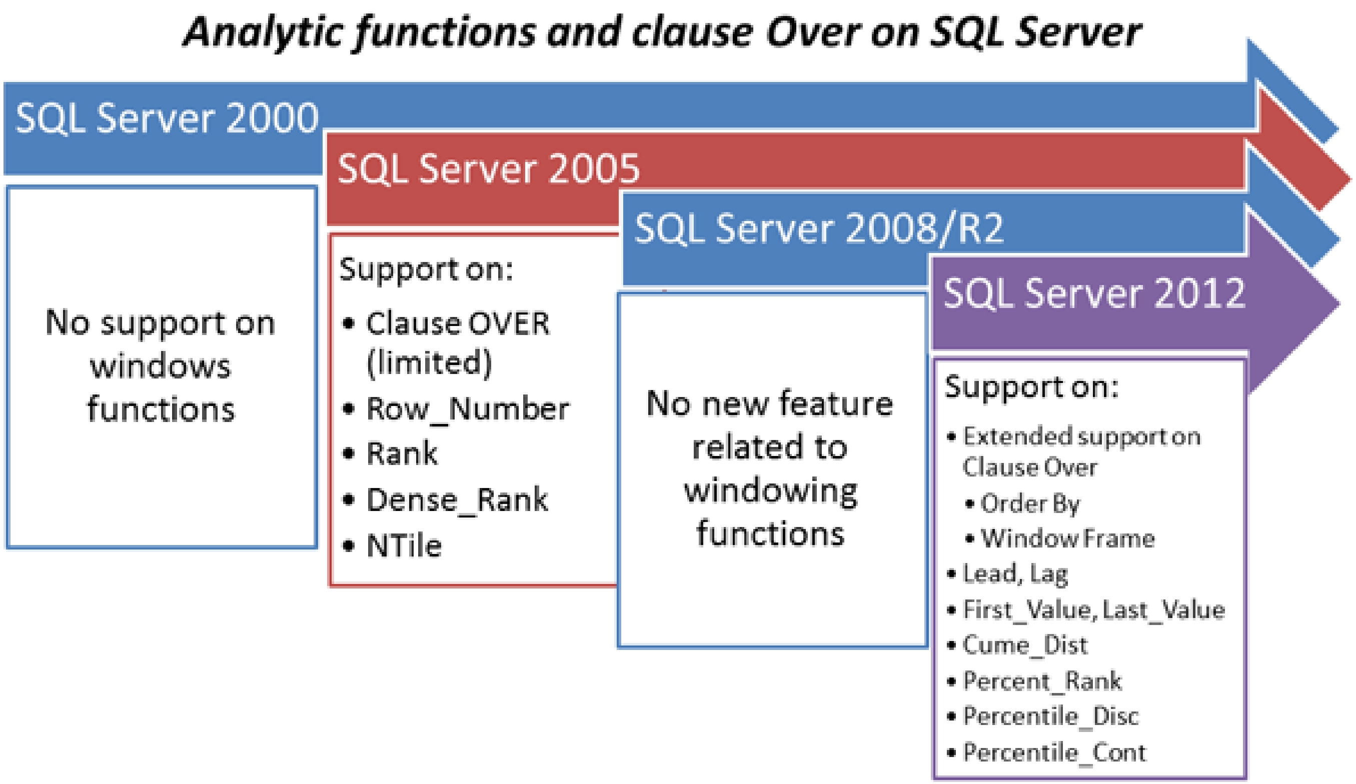
takes B and gives

Course	Year	Earning
.NET	2012	15000.00
Java	2012	20000.00
.NET	2013	48000.00
Java	2013	30000.00

UNPIVOT Query on Course Column

```
SELECT *
FROM #CourseSalesPivotResult
UNPIVOT(Earning
FOR Course IN ([.NET], Java))
AS UNPVTTable
```

see total revenue for each course by year



Analytic functions and clause Over on SQL Server

SQL Server 2000

No support on windows functions

SQL Server 2005

Support on:
- Clause OVER (limited)
- Row_Number
- Rank
- Dense_Rank
- NTile

SQL Server 2008/R2

No new feature related to windowing functions

SQL Server 2012

Support on:
- Extended support on Clause Over
  - Order By
  - Window Frame
- Lead, Lag
- First_Value, Last_Value
- Cume_Dist
- Percent_Rank
- Percentile_Disc
- Percentile_Cont

SQL Server 2005 introduced window ranking functions:

1. ROW_NUMBER
2. RANK
3. DENSE_RANK
4. NTILE

To test the functions, we'll use a table called Tab1. The code to create the table is the following:

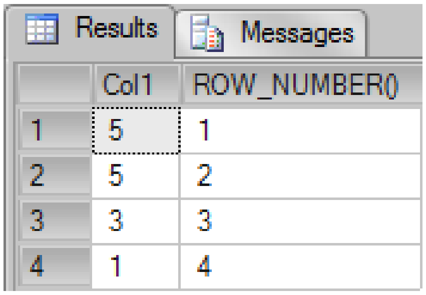
```
USE TempDB
GO
IF OBJECT_ID('Tab1') IS NOT NULL
  DROP TABLE Tab1
GO
CREATE TABLE Tab1 (Col1 INT)
GO

INSERT INTO Tab1 VALUES(5), (5), (3) , (1)
GO
```

Row_Number()

5. The ROW_NUMBER function is used to generate a sequence of numbers based in a set in a specific order, in easy words, it returns the sequence number of each row inside a set in the order that you specify.
6. For instance:

```
-- RowNumber
```

```
SELECT Col1,
      ROW_NUMBER() OVER(ORDER BY Col1 DESC) AS "ROW_NUMBER()"
  FROM Tab1
```

7. 

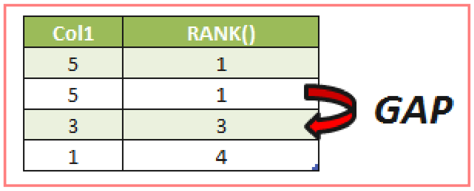
	Col1	ROW_NUMBER()
1	5	1
2	5	2
3	3	3
4	1	4

8. The column called "ROW_NUMBER()" is one of a series of numbers created in the order of Col1 descending. The clause OVER(ORDER BY Col1 DESC) is used to specify the order of the sequence for which the number should be created. It is necessary because rows in a relational table have no 'natural' order.

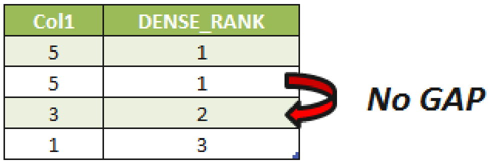
Rank() & Dense_Rank()

9. Return the position in a ranking for each row inside a partition. The ranking is calculated by 1 plus the number of previews rows.
10. It's important to mention that the function RANK returns the result with a GAP after a tie, whereas the function DENSE_RANK doesn't. To understand this better, let's see some samples.

```
-- Rank
SELECT Col1,
      RANK() OVER(ORDER BY Col1 DESC) AS "RANK()"  FROM Tab1
GO

-- Dense_Rank
SELECT Col1,
      DENSE_RANK() OVER(ORDER BY Col1 DESC) AS "DENSE_RANK"  FROM Tab1
```



Col1	RANK()
5	1
5	1
3	3
1	4

11. 

Col1	DENSE_RANK
5	1
5	1
3	2
1	3

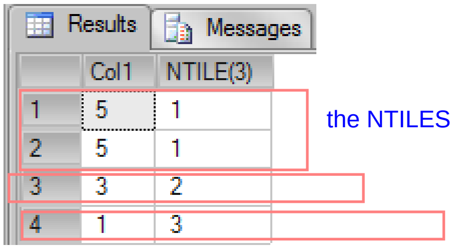
12. Notice that in the RANK result, we have the values 1,1,3 and 4. The value Col1 = "5" is duplicated so any ordering will produce a 'tie' for position between them. They have the same position in the rank, but, when the

ordinal position for the value 3 is calculated, this position isn't 2 because the position 2 was already used for the value 5, in this case the an GAP is generated and the function returns the next value for the tank, in this case the value 3.

NTILE()

The NTILE function is used for calculating summary statistics. It distributes the rows within an ordered partition into a specified number of "buckets" or groups. The groups are numbered, starting at one. For each row, NTILE returns the number of the group to which the row belongs. It makes it easy to calculate n-tile distributions such as percentiles.

```
-- NTILE
SELECT Col1,
    NTILE(3) OVER(ORDER BY Col1 DESC) AS "NTILE(3)"
 FROM Tab1
```

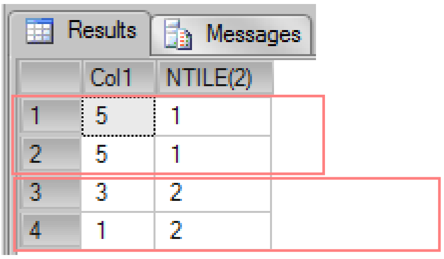


	Col1	NTILE(3)
1	5	1
2	5	1
3	3	2
4	1	3

the NTILES 1, 2 and 3...

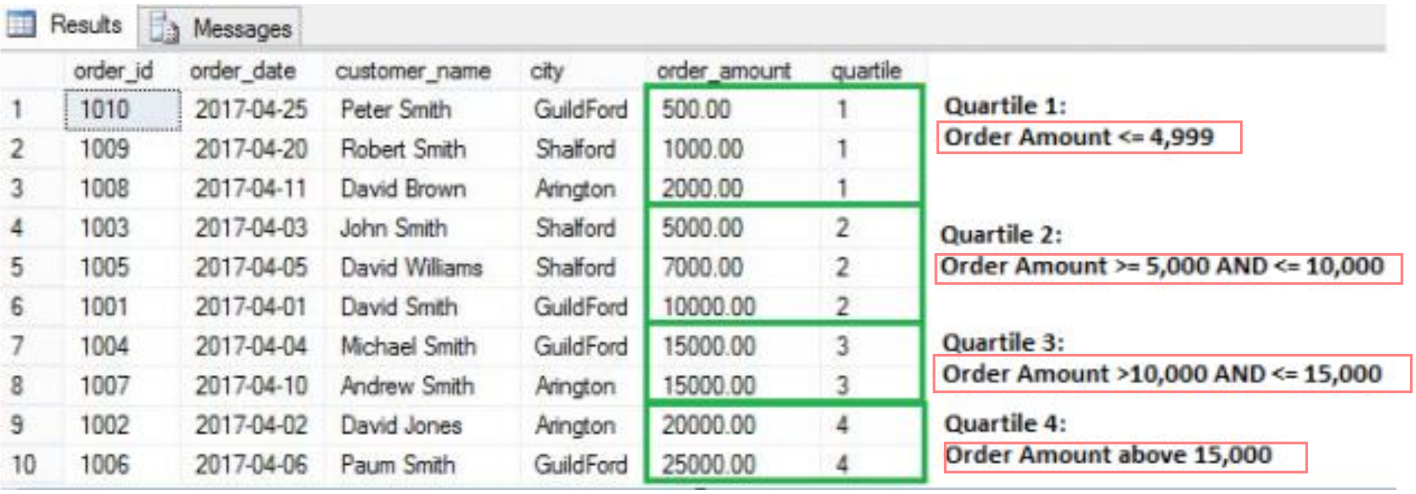
In the result above we can see that 4 rows were divided by 3 it's 1, the remaining row is added in the initial group. Let's see another sample without remained rows.

```
-- NTILE
SELECT Col1,
    NTILE(2) OVER(ORDER BY Col1 DESC) AS "NTILE(2)"
 FROM Tab1
```



	Col1	NTILE(2)
1	5	1
2	5	1
3	3	2
4	1	2

It is good practice to order on a unique key, ensure that there are more buckets than rows, and to have an equal number of rows in each bucket



	order_id	order_date	customer_name	city	order_amount	quartile
1	1010	2017-04-25	Peter Smith	GuildFord	500.00	1
2	1009	2017-04-20	Robert Smith	Shalford	1000.00	1
3	1008	2017-04-11	David Brown	Arington	2000.00	1
4	1003	2017-04-03	John Smith	Shalford	5000.00	2
5	1005	2017-04-05	David Williams	Shalford	7000.00	2
6	1001	2017-04-01	David Smith	GuildFord	10000.00	2
7	1004	2017-04-04	Michael Smith	GuildFord	15000.00	3
8	1007	2017-04-10	Andrew Smith	Arington	15000.00	3
9	1002	2017-04-02	David Jones	Arington	20000.00	4
10	1006	2017-04-06	Paum Smith	GuildFord	25000.00	4

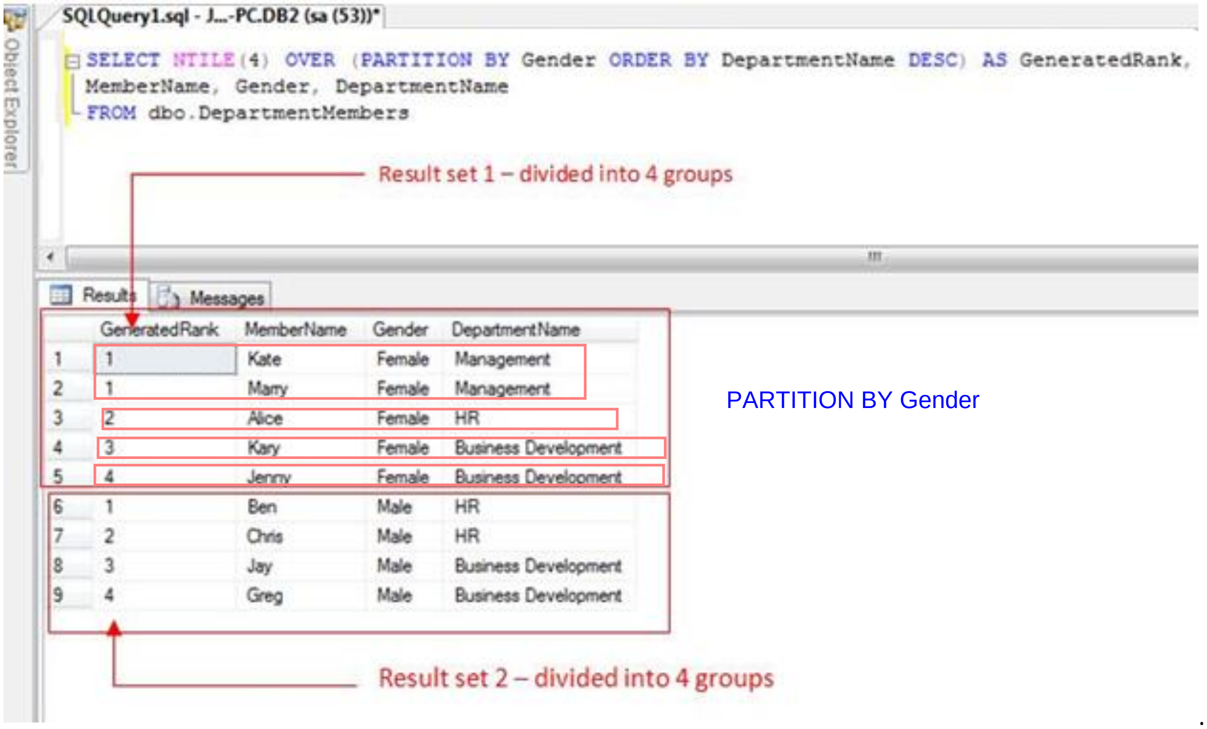
Quartile 1:
Order Amount <= 4,999

Quartile 2:
Order Amount >= 5,000 AND <= 10,000

Quartile 3:
Order Amount >10,000 AND <= 15,000

Quartile 4:
Order Amount above 15,000

space



```
SELECT NTILE(4) OVER (PARTITION BY Gender ORDER BY DepartmentName DESC) AS GeneratedRank,
MemberName, Gender, DepartmentName
FROM dbo.DepartmentMembers
```

Result set 1 – divided into 4 groups

	GeneratedRank	MemberName	Gender	DepartmentName
1	1	Kate	Female	Management
2	1	Mary	Female	Management
3	2	Alice	Female	HR
4	3	Kary	Female	Business Development
5	4	Jenny	Female	Business Development
6	1	Ben	Male	HR
7	2	Chris	Male	HR
8	3	Jay	Male	Business Development
9	4	Greg	Male	Business Development

Result set 2 – divided into 4 groups

PARTITION BY Gender

.

SQL Server 2012 enhances:

1. Window Aggregate Functions by introducing window order and frame clauses.
   - The "window concept" is in how you define the set of rows for the function to apply to, and where you can use the function in the language.
   - Avoid the limitations that apply to grouped queries and subqueries.
     - Grouped queries that define groups, or sets, of rows to which aggregate functions can be applied.
     - Each group is represented by one result row.
     - After grouping the data; apply all computations in the context of the groups.
2. Analytic Functions that calculate an aggregate value based on a group of rows.
   - Unlike aggregate functions, analytic functions can return multiple rows for each group.
   - Use analytic functions to compute moving averages, running totals, percentages or top-N results within a group.

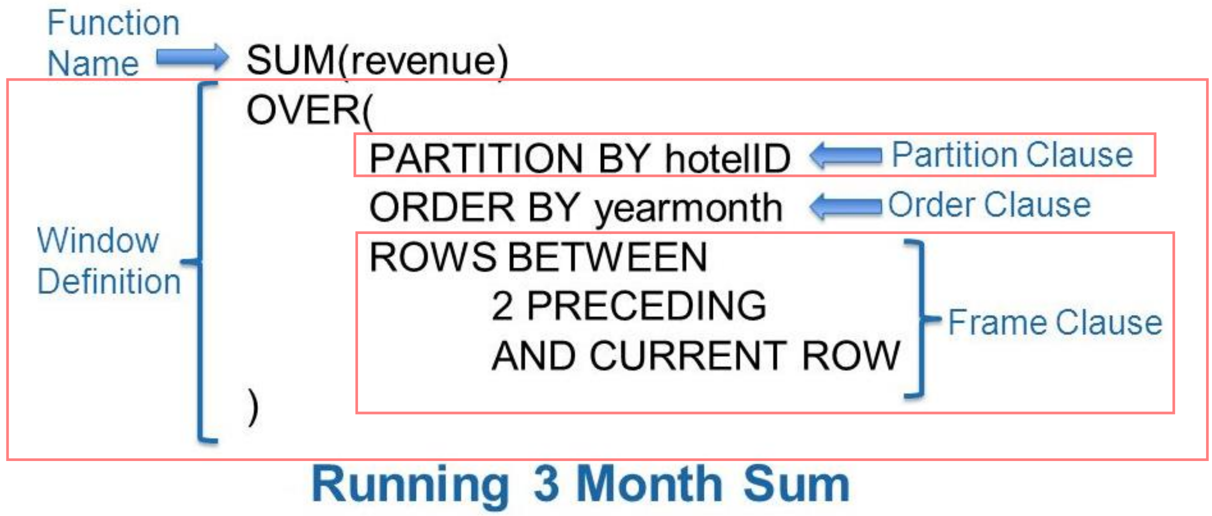
Offset (Analytic) Functions

1. LAG
    - Analytic function in a SELECT statement to compare values in the current row with values in a previous row.
2. LEAD
    - Analytic function in a SELECT statement to compare values in the current row with values in a following row
3. FIRST_VALUE
    - Returns the first value in an ordered set of values.
4. LAST_VALUE
    - Returns the last value in an ordered set of values.

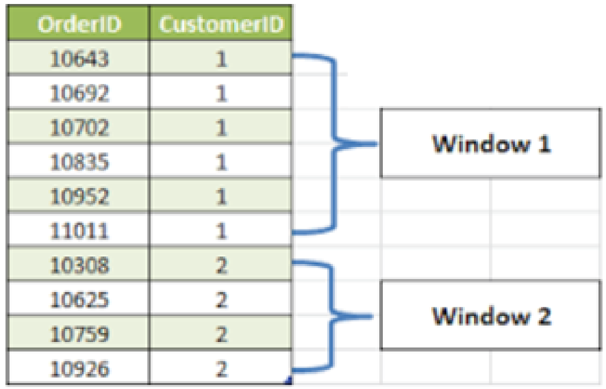
Window Distribution (Analytic) Functions

1. PERCENT_RANK
    - Calculates the relative rank of a row within a group of rows.
2. CUME_DIST
    - Calculates the cumulative distribution of a value within a group of values.
3. PERCENTILE_DISC
    - Returns an actual value from the set.
4. PERCENTILE_CONT
    - Calculates a percentile based on a continuous distribution of the column value in SQL Server.
    - Example: Use PERCENTILE_CONT and PERCENTILE_DISC to find the median employee salary in each department.
    - Note: These functions may not return the same value:
        - PERCENTILE_CONT interpolates the appropriate value, whether or not it exists in the data set. (interpolation is a method of constructing new data points within the range of a discrete set of known data points)
        - PERCENTILE_DISC always returns an actual value from the set.

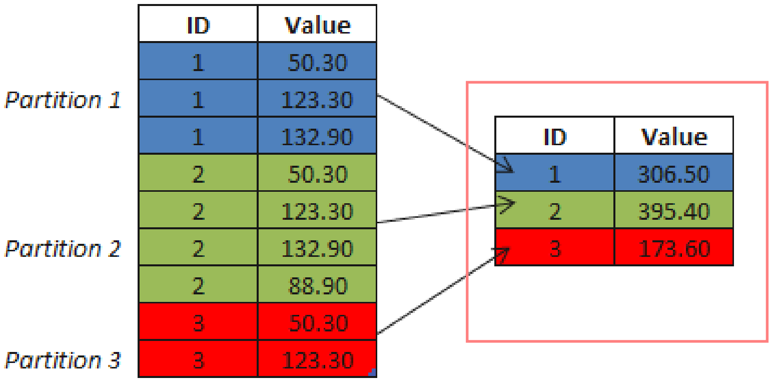
Windows, Partitions, Frames: More Containers….



```
SUM(revenue)
OVER(
        PARTITION BY hotelID
        ORDER BY yearmonth
        ROWS BETWEEN
                2 PRECEDING
                AND CURRENT ROW
)
```



OrderID	CustomerID
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1
10308	2
10625	2
10759	2
10926	2



ID	Value
1	50.30
1	123.30
1	132.90
2	50.30
2	123.30
2	132.90
2	88.90
3	50.30
3	123.30

ID	Value
1	306.50
2	395.40
3	173.60

In the blue we have partition 1, green as partition 2 and red as partition 3. Because we applied the aggregation function in the column value, grouping the results by ID, we the lose the details of the data. In this case the details are the values of the columns of the rows in the partitions 1, 2 and 3.

OVER Clause

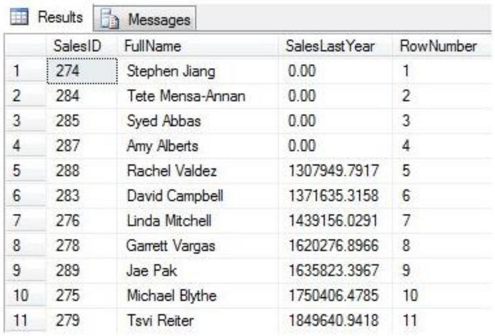
- The ROWS clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row.
- Alternatively, the RANGE clause logically limits the rows within a partition by specifying a range of values with respect to the value in the current row.
- Preceding and 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.
- 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.

The OVER clause

Use the OVER clause to define the "window" or specific set of rows to apply the windowing function

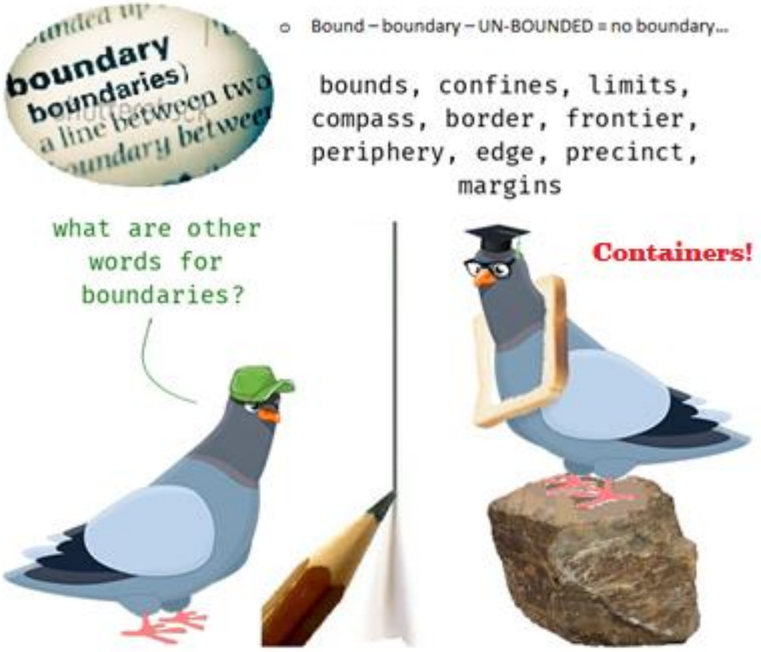
- OVER must be combined with an ORDER BY clause (which makes sense)

```
SELECT  BusinessEntityID AS SalesID,
 FirstName + ' ' + LastName AS FullName,
 SalesLastYear,
 ROW_NUMBER() OVER(ORDER BY SalesLastYear ASC)
AS RowNumber
FROM Sales.vSalesPerson;
```



	SalesID	FullName	SalesLastYear	RowNumber
1	274	Stephen Jiang	0.00	1
2	284	Tete Mensa-Annan	0.00	2
3	285	Syed Abbas	0.00	3
4	287	Amy Alberts	0.00	4
5	288	Rachel Valdez	1307949.7917	5
6	283	David Campbell	1371635.3158	6
7	276	Linda Mitchell	1439156.0291	7
8	278	Garrett Vargas	1620276.8966	8
9	289	Jae Pak	1635823.3967	9
10	275	Michael Blythe	1750406.4785	10
11	279	Tsvi Reiter	1849640.9418	11

1. OVER
    - Defines the set of rows for the function to work with – called the "Window".
    - Without a PARTITION BY the entire result set is the "Window".
2. PARTITION BY
    - Divides the query result set into partitions.
    - The window function is applied to each partition separately and computation restarts for each partition – thus creating many "Windows".
    - If not specified the function treats all rows of the query result set as a single group.
3. BETWEEN <window frame bound > AND <window frame bound >
    - Bound – boundary – UN-BOUNDED = no boundary…

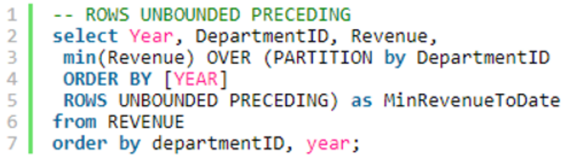

boundary
boundaries
a line between two
o Bound – boundary – UN-BOUNDED = no boundary…
bounds, confines, limits, compass, border, frontier, periphery, edge, precinct, margins
what are other words for boundaries?
Containers!

4. CURRENT ROW
    - Specifies that the window starts or ends at the current row when used with ROWS or the current value when used with RANGE.
5. BETWEEN <window frame bound > AND <window frame bound >

- Specify the lower (starting) and upper (ending) boundary points of the window.
- <window frame bound> defines the boundary starting point
- <window frame bound> defines the boundary end point.
- The upper bound cannot be smaller than the lower bound.

Preceding and following rows are defined based on the ordering in the ORDER BY clause

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



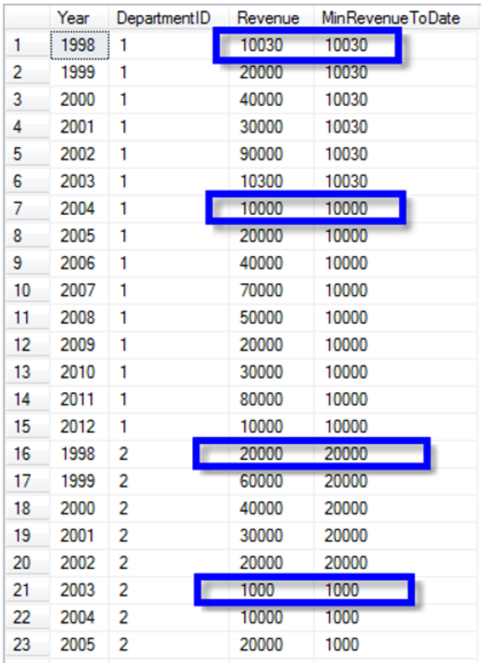
```sql
-- ROWS UNBOUNDED PRECEDING
select Year, DepartmentID, Revenue,
 min(Revenue) OVER (PARTITION by DepartmentID
 ORDER BY [YEAR]
 ROWS UNBOUNDED PRECEDING) as MinRevenueToDate
from REVENUE
order by departmentID, year;
```

In this example, the MinRevenueToDate lists the lowest revenue for this row and all earlier rows ordered by date in the current department id.

Row 1 sets the MinRevenueToDate to 10030, and doesn't change until row 7 with a lower revenue year.

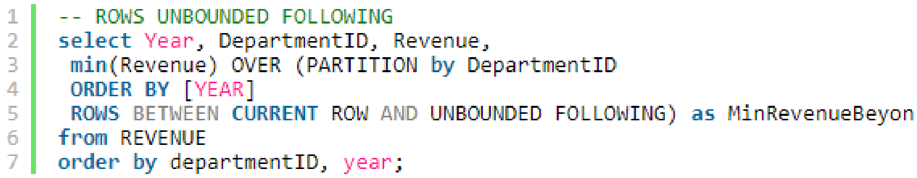
Row 16 starts over for a new department with the lowest running revenue to date.

Then row 21 resets to a lower revenue number.

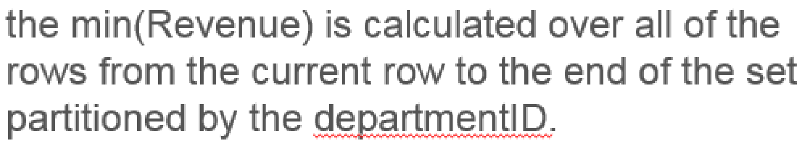


	Year	DepartmentID	Revenue	MinRevenueToDate
1	1998	1	10030	10030
2	1999	1	20000	10030
3	2000	1	40000	10030
4	2001	1	30000	10030
5	2002	1	90000	10030
6	2003	1	10300	10030
7	2004	1	10000	10000
8	2005	1	20000	10000
9	2006	1	40000	10000
10	2007	1	70000	10000
11	2008	1	50000	10000
12	2009	1	20000	10000
13	2010	1	30000	10000
14	2011	1	80000	10000
15	2012	1	10000	10000
16	1998	2	20000	20000
17	1999	2	60000	20000
18	2000	2	40000	20000
19	2001	2	30000	20000
20	2002	2	20000	20000
21	2003	2	1000	1000
22	2004	2	10000	1000
23	2005	2	20000	1000

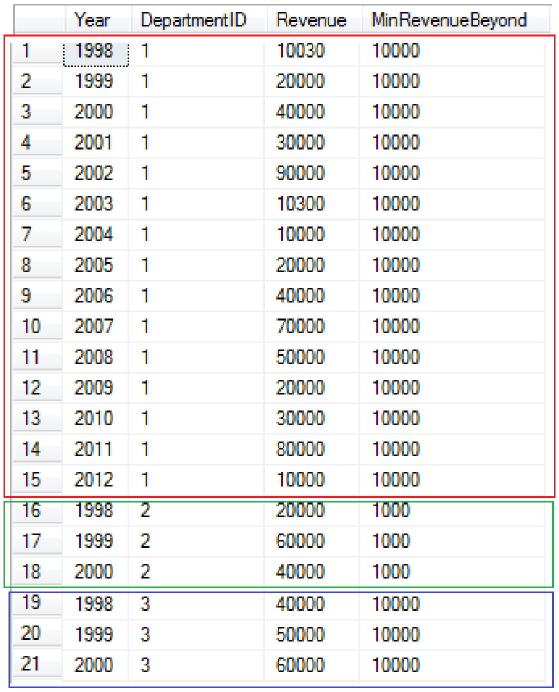
7. <unsigned value specification> PRECEDING
   - Indicates the number of rows or values to precede the current row.
   - Not allowed for RANGE.
8. UNBOUNDED FOLLOWING (opposite "direction" of PRECEDING)
   - Specifies that the window ends at the last row of the partition.
   - Example: RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING defines a window that starts with the current row and ends with the last row of the partition.



```
-- ROWS UNBOUNDED FOLLOWING
select Year, DepartmentID, Revenue,
 min(Revenue) OVER (PARTITION by DepartmentID
 ORDER BY [YEAR]
 ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) as MinRevenueBeyon
from REVENUE
order by departmentID, year;
```



the min(Revenue) is calculated over all of the rows from the current row to the end of the set partitioned by the departmentID.

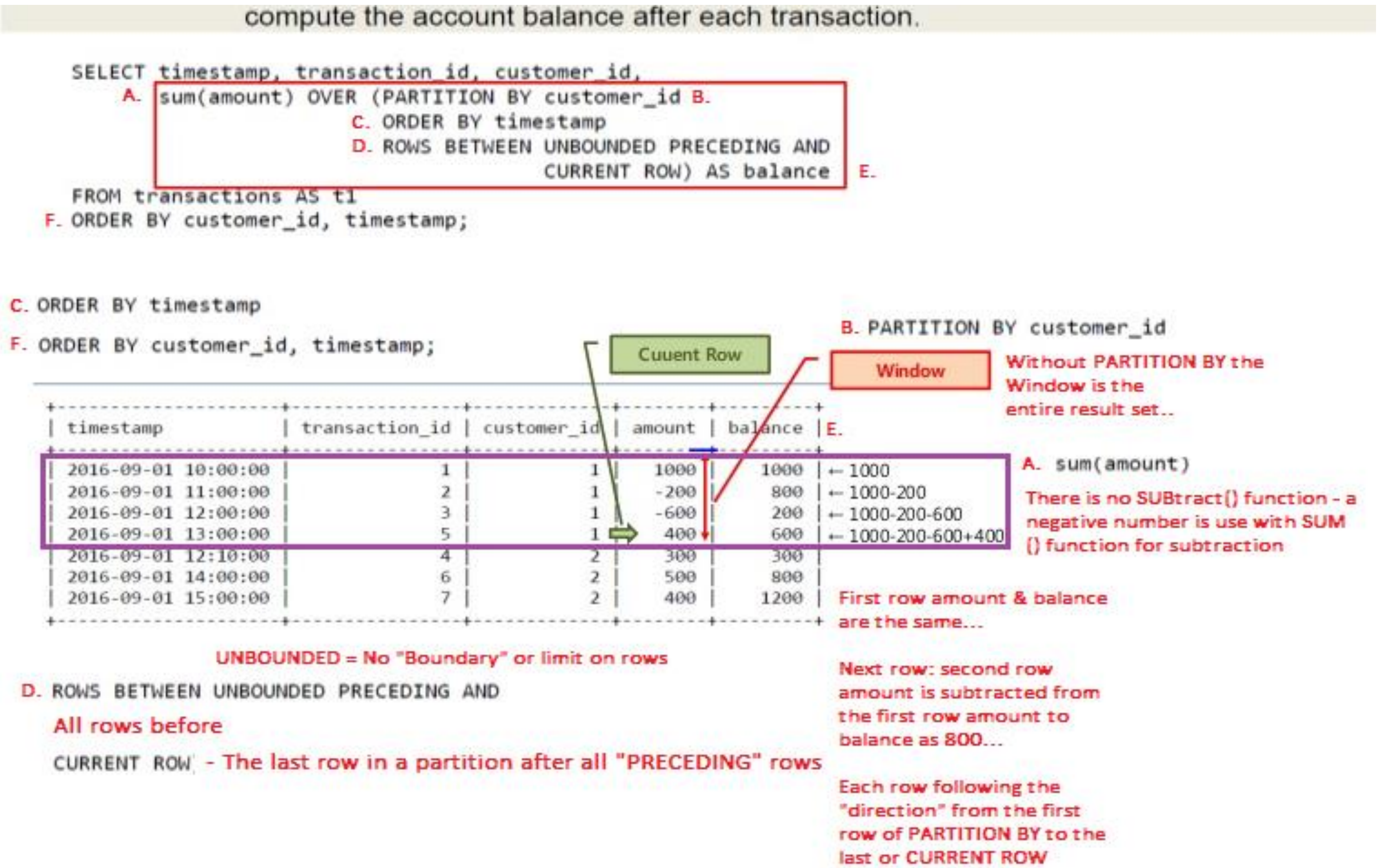


	Year	DepartmentID	Revenue	MinRevenueBeyond
1	1998	1	10030	10000
2	1999	1	20000	10000
3	2000	1	40000	10000
4	2001	1	30000	10000
5	2002	1	90000	10000
6	2003	1	10300	10000
7	2004	1	10000	10000
8	2005	1	20000	10000
9	2006	1	40000	10000
10	2007	1	70000	10000
11	2008	1	50000	10000
12	2009	1	20000	10000
13	2010	1	30000	10000
14	2011	1	80000	10000
15	2012	1	10000	10000
16	1998	2	20000	1000
17	1999	2	60000	1000
18	2000	2	40000	1000
19	1998	3	40000	10000
20	1999	3	50000	10000
21	2000	3	60000	10000

9. <unsigned value specification> FOLLOWING
   - Indicates the number of rows or values to follow the current row.
   - The ending point must be <unsigned value specification>FOLLOWING.
   - 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.
   - Not allowed for RANGE.

ROWS or RANGE requires that the ORDER BY clause be specified

10. ROWS
    - Limits the rows within a partition by specifying a fixed number of rows preceding or following the current row.
    - 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.
11. RANGE
    - Logically limits the rows within a partition by specifying a range of values with respect to the value in the current row.



compute the account balance after each transaction.

```
SELECT timestamp, transaction_id, customer_id,
 A. sum(amount) OVER (PARTITION BY customer_id B.
                      C. ORDER BY timestamp
                      D. ROWS BETWEEN UNBOUNDED PRECEDING AND
                                 CURRENT ROW) AS balance   E.
FROM transactions AS t1
F. ORDER BY customer_id, timestamp;
```

C. ORDER BY timestamp

F. ORDER BY customer_id, timestamp;

Cuuent Row

B. PARTITION BY customer_id

Window

Without PARTITION BY the Window is the entire result set..

```
+---------------------+----------------+-------------+--------+---------+
| timestamp           | transaction_id | customer_id | amount | balance | E.
+---------------------+----------------+-------------+--------+---------+
| 2016-09-01 10:00:00 |              1 |           1 |   1000 |    1000 | ← 1000
| 2016-09-01 11:00:00 |              2 |           1 |   -200 |     800 | ← 1000-200
| 2016-09-01 12:00:00 |              3 |           1 |   -600 |     200 | ← 1000-200-600
| 2016-09-01 13:00:00 |              5 |           1 |    400 |     600 | ← 1000-200-600+400
| 2016-09-01 12:10:00 |              4 |           2 |    300 |     300 |
| 2016-09-01 14:00:00 |              6 |           2 |    500 |     800 |
| 2016-09-01 15:00:00 |              7 |           2 |    400 |    1200 |
+---------------------+----------------+-------------+--------+---------+
```

A. sum(amount)

There is no SUBtract() function - a negative number is use with SUM () function for subtraction

First row amount & balance are the same...

UNBOUNDED = No "Boundary" or limit on rows

D. ROWS BETWEEN UNBOUNDED PRECEDING AND

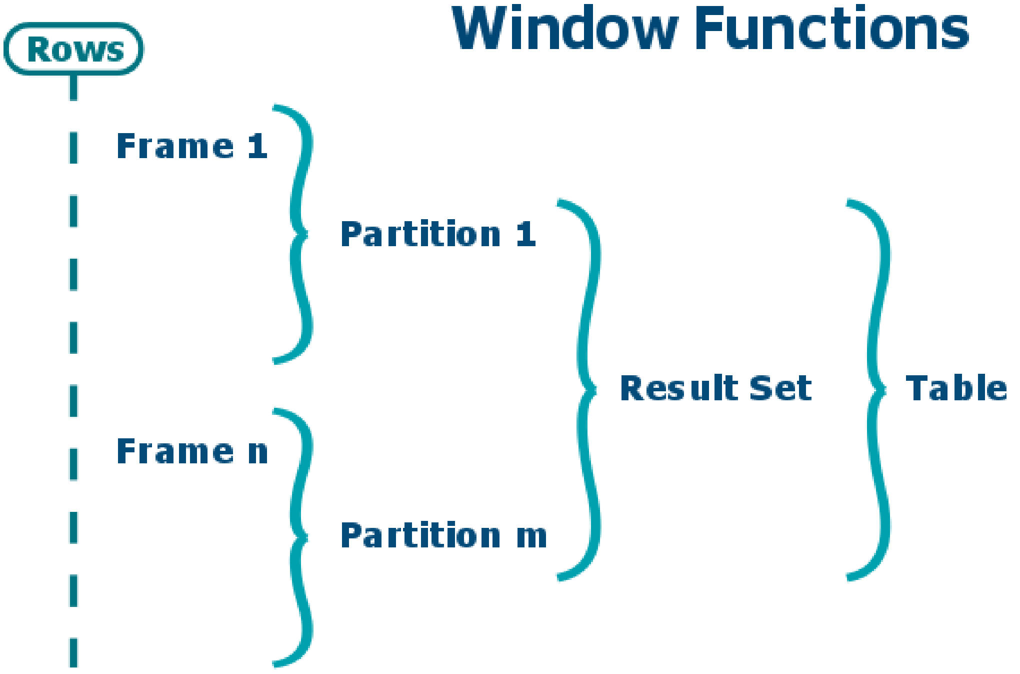
All rows before

CURRENT ROW - The last row in a partition after all "PRECEDING" rows

Next row: second row amount is subtracted from the first row amount to balance as 800...

Each row following the "direction" from the first row of PARTITION BY to the last or CURRENT ROW

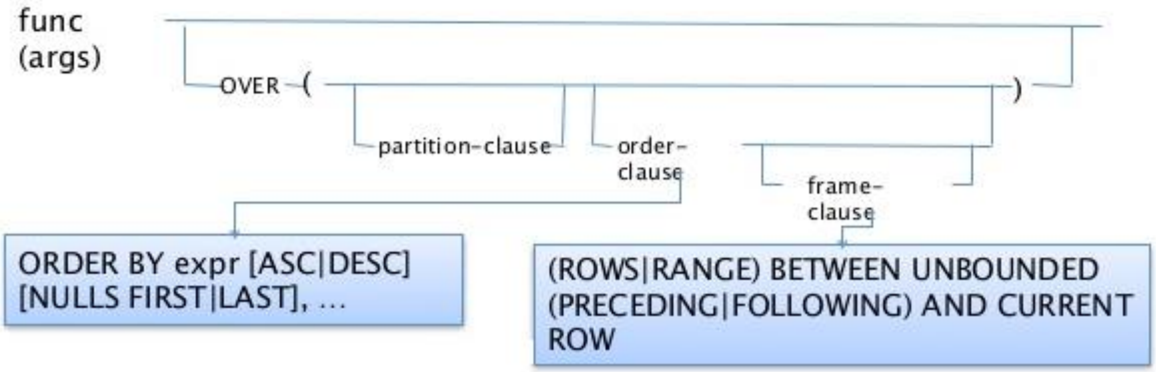
Window Functions



Frame

A frame

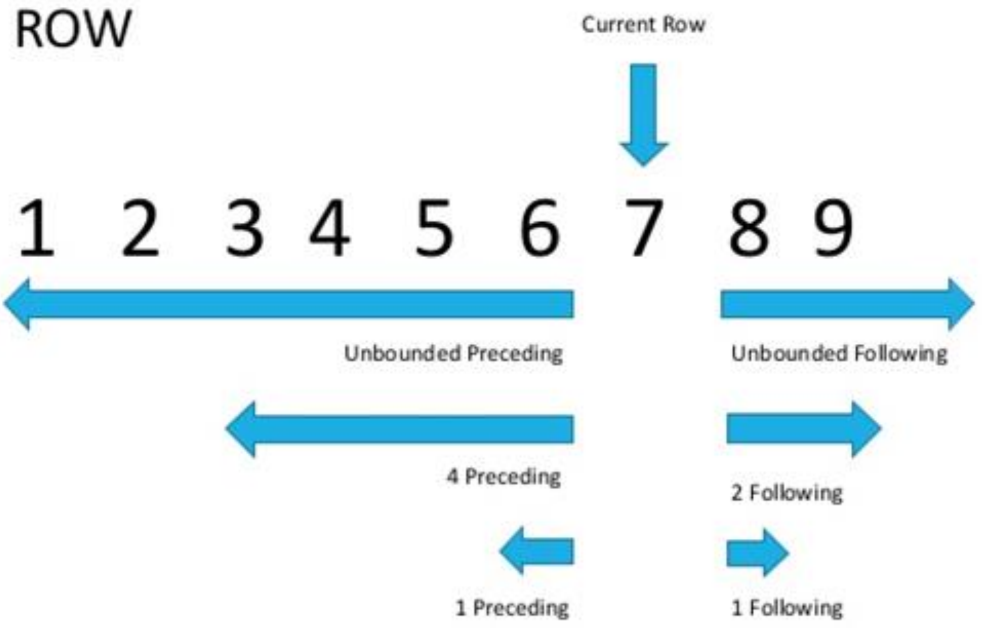
- Specified by ORDER BY clause and frame clause in OVER()
- Allows to tell how far the set is applied
- Also defines ordering of the set
- Without order and frame clauses, the whole of partition is a single frame

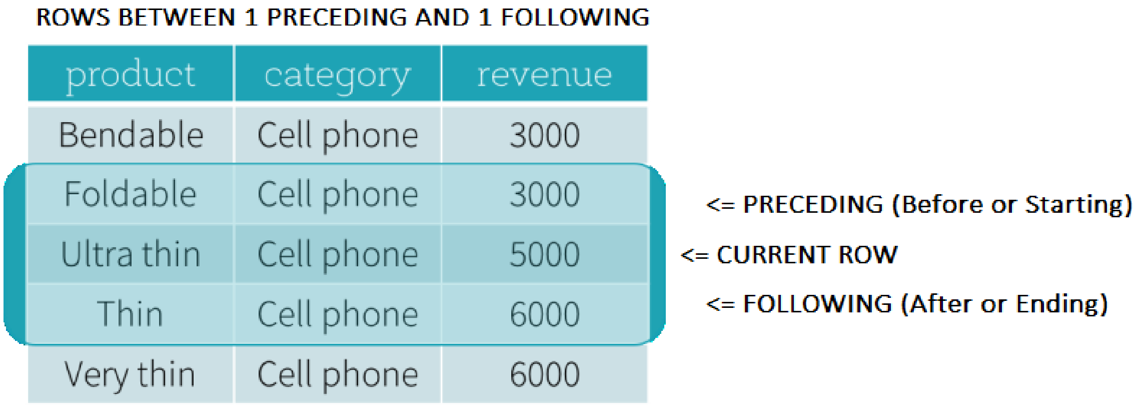


ORDER BY expr [ASC|DESC] [NULLS FIRST|LAST], …

(ROWS|RANGE) BETWEEN UNBOUNDED (PRECEDING|FOLLOWING) AND CURRENT ROW

Space





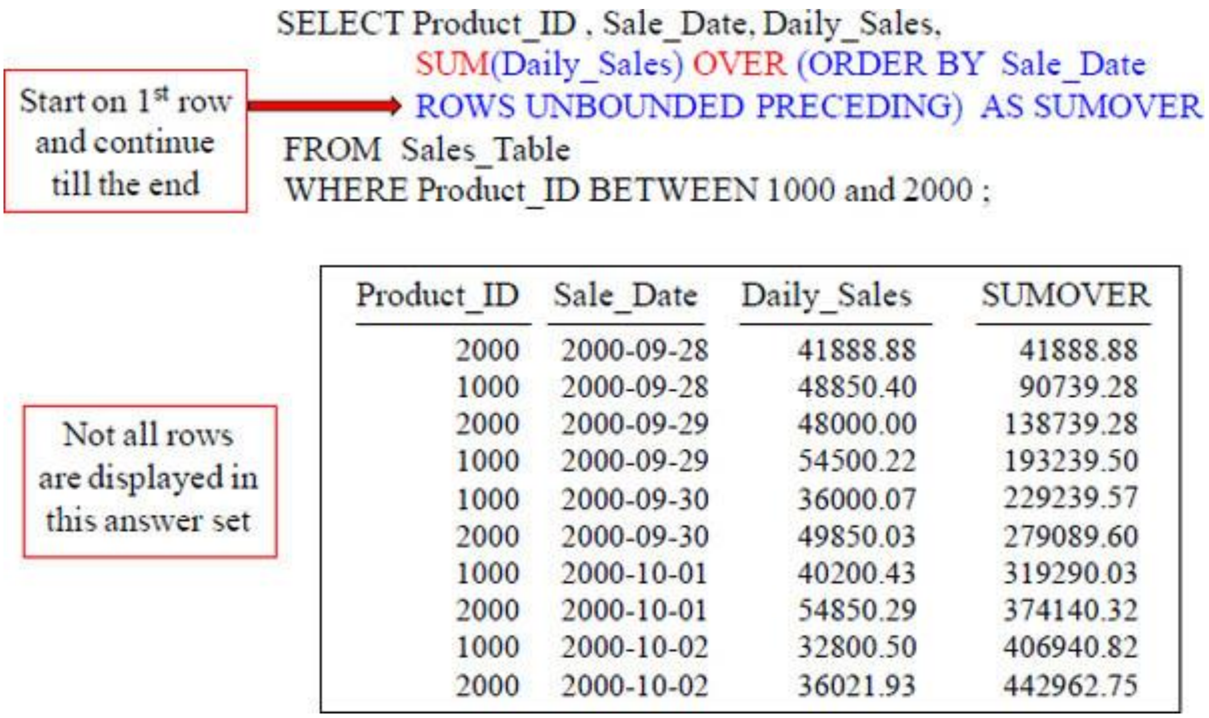
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

product	category	revenue
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Ultra thin	Cell phone	5000
Thin	Cell phone	6000
Very thin	Cell phone	6000

<= PRECEDING (Before or Starting)

<= CURRENT ROW

<= FOLLOWING (After or Ending)



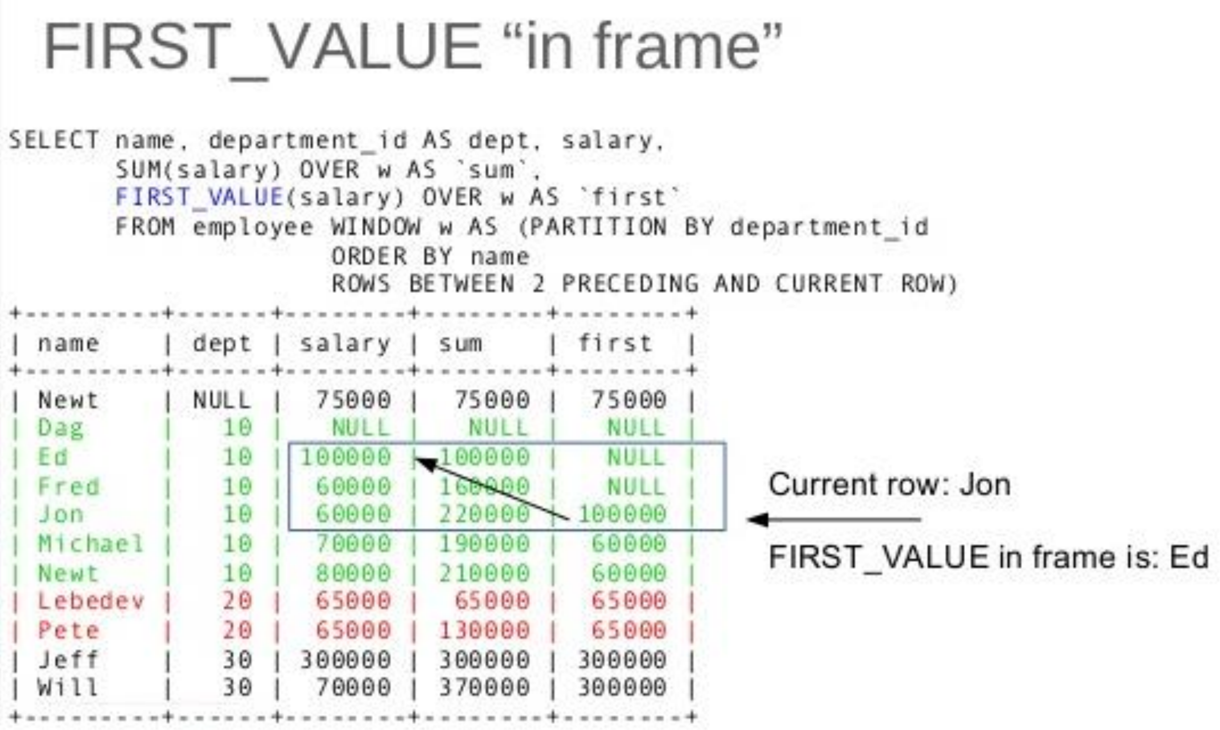
Start on 1st row and continue till the end

```
SELECT Product_ID , Sale_Date, Daily_Sales,
          SUM(Daily_Sales) OVER (ORDER BY  Sale_Date
          ROWS UNBOUNDED PRECEDING)  AS SUMOVER
FROM  Sales_Table
WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Not all rows are displayed in this answer set

```
Product_ID   Sale_Date    Daily_Sales     SUMOVER
      2000  2000-09-28       41888.88    41888.88
      1000  2000-09-28       48850.40    90739.28
      2000  2000-09-29       48000.00   138739.28
      1000  2000-09-29       54500.22   193239.50
      1000  2000-09-30       36000.07   229239.57
      2000  2000-09-30       49850.03   279089.60
      1000  2000-10-01       40200.43   319290.03
      2000  2000-10-01       54850.29   374140.32
      1000  2000-10-02       32800.50   406940.82
      2000  2000-10-02       36021.93   442962.75
```

Space



FIRST_VALUE "in frame"

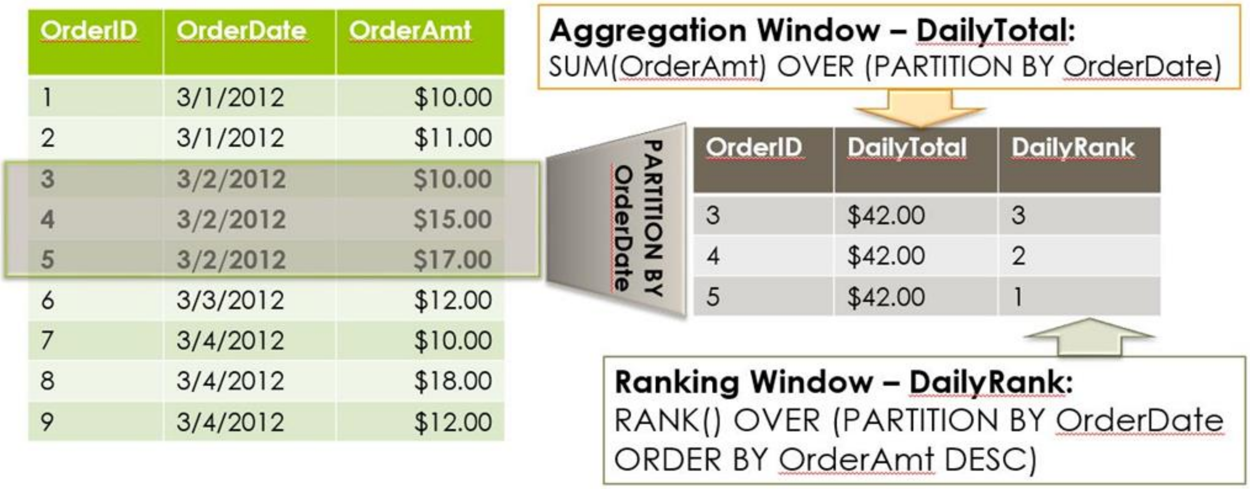
```
SELECT name, department_id AS dept, salary,
       SUM(salary) OVER w AS `sum`,
       FIRST_VALUE(salary) OVER w AS `first`
       FROM employee WINDOW w AS (PARTITION BY department_id
                            ORDER BY name
                            ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
+---------+------+--------+--------+--------+
| name    | dept | salary | sum    | first  |
+---------+------+--------+--------+--------+
| Newt    | NULL |  75000 |  75000 |  75000 |
| Dag     |   10 |   NULL |   NULL |   NULL |
| Ed      |   10 | 100000 | 100000 |   NULL |
| Fred    |   10 |  60000 | 160000 |   NULL |
| Jon     |   10 |  60000 | 220000 | 100000 |
| Michael |   10 |  70000 | 190000 |  60000 |
| Newt    |   10 |  80000 | 210000 |  60000 |
| Lebedev |   20 |  65000 |  65000 |  65000 |
| Pete    |   20 |  65000 | 130000 |  65000 |
| Jeff    |   30 | 300000 | 300000 | 300000 |
| Will    |   30 |  70000 | 370000 | 300000 |
+---------+------+--------+--------+--------+
```

Current row: Jon

FIRST_VALUE in frame is: Ed

SPACE

Basic PARTITION BY

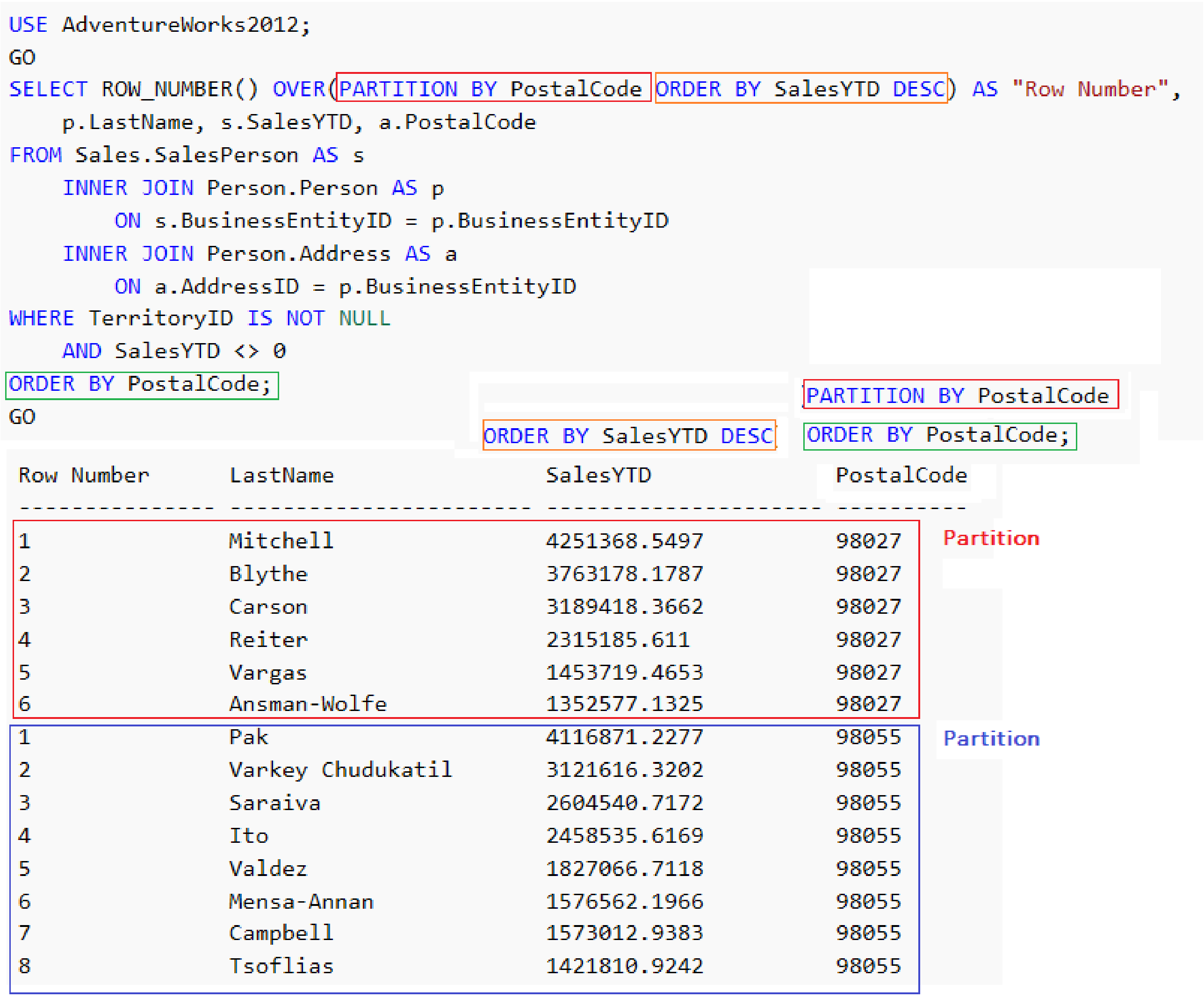


OrderID	OrderDate	OrderAmt
1	3/1/2012	$10.00
2	3/1/2012	$11.00
3	3/2/2012	$10.00
4	3/2/2012	$15.00
5	3/2/2012	$17.00
6	3/3/2012	$12.00
7	3/4/2012	$10.00
8	3/4/2012	$18.00
9	3/4/2012	$12.00

Aggregation Window – DailyTotal:
SUM(OrderAmt) OVER (PARTITION BY OrderDate)

OrderID	DailyTotal	DailyRank
3	$42.00	3
4	$42.00	2
5	$42.00	1

Ranking Window – DailyRank:
RANK() OVER (PARTITION BY OrderDate ORDER BY OrderAmt DESC)



```
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
```

PARTITION BY PostalCode

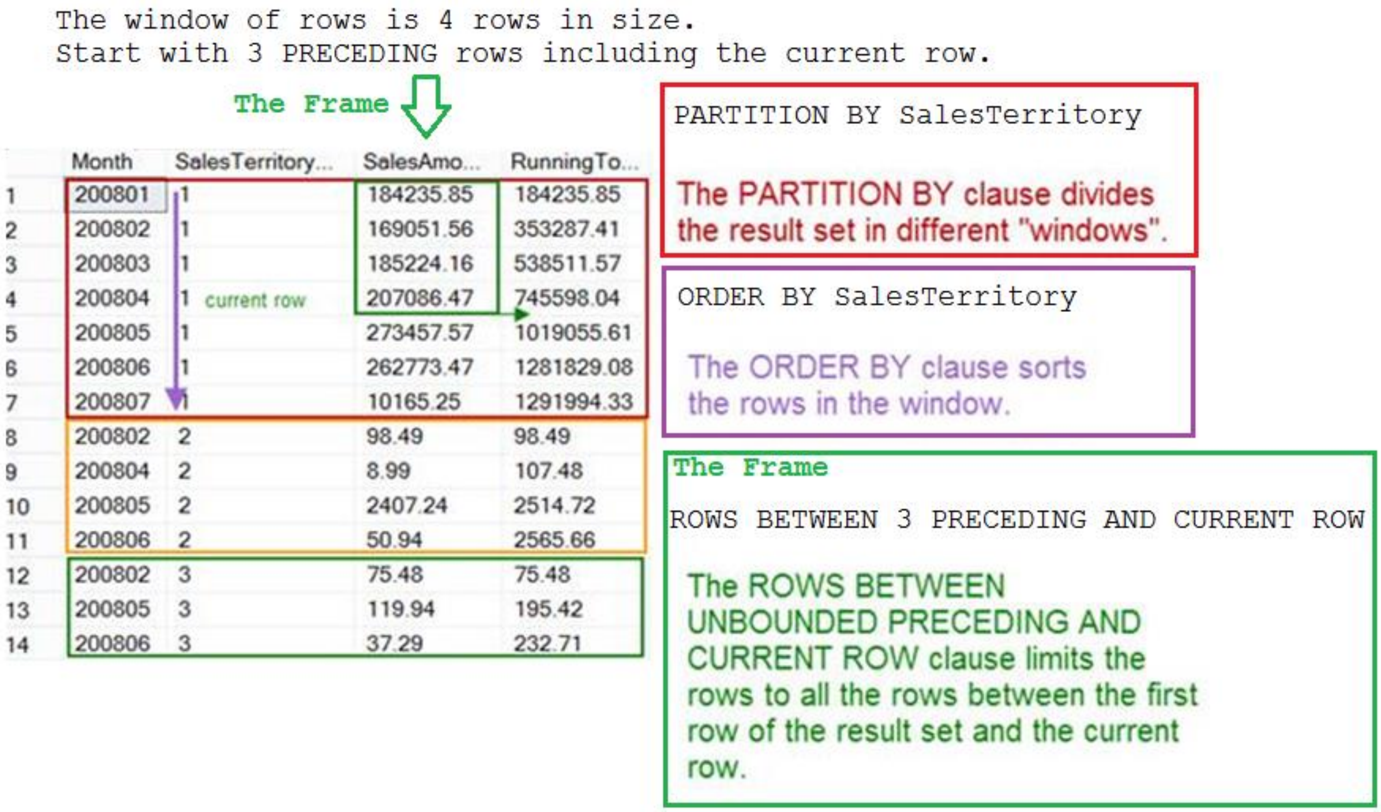
ORDER BY SalesYTD DESC

ORDER BY PostalCode;

```
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
```

Partition

Partition



The window of rows is 4 rows in size.
Start with 3 PRECEDING rows including the current row.

	Month	SalesTerritory...	SalesAmo...	RunningTo...
1	200801	1	184235.85	184235.85
2	200802	1	169051.56	353287.41
3	200803	1	185224.16	538511.57
4	200804	1	207086.47	745598.04
5	200805	1	273457.57	1019055.61
6	200806	1	262773.47	1281829.08
7	200807	1	10165.25	1291994.33
8	200802	2	98.49	98.49
9	200804	2	8.99	107.48
10	200805	2	2407.24	2514.72
11	200806	2	50.94	2565.66
12	200802	3	75.48	75.48
13	200805	3	119.94	195.42
14	200806	3	37.29	232.71

PARTITION BY SalesTerritory

The PARTITION BY clause divides the result set in different "windows".

ORDER BY SalesTerritory

The ORDER BY clause sorts the rows in the window.

The Frame

ROWS BETWEEN 3 PRECEDING AND CURRENT ROW

The ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW clause limits the rows to all the rows between the first row of the result set and the current row.

Window Frame: Another Discussion… might be repetive…

The concept of a window frame - examine the syntax of a window function and the window frame:

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

The window frame is a very important concept when used in windowing and aggregation functions, and it can also be very confusing. One reason for the confusion is that it is also known by the synonymous terms *window frame, window size* or *sliding window*. I'm calling this a window frame because this is the term that Microsoft chose to call it in books online.

In the window frame, you can specify the subset of rows in which the windowing function will work. You can specify the top and bottom boundary condition of the sliding window using the window specification clause. The syntax for the window specification clause is:

[ROWS | RANGE] BETWEEN <Start expr> AND <End expr>

Where:

<Start expr> is one of:

- *UNBOUNDED PRECEDING*: The window starts in the first row of the partition
- *CURRENT ROW*: The window starts in the current row
- *<unsigned integer literal> PRECEDING* or *FOLLOWING*

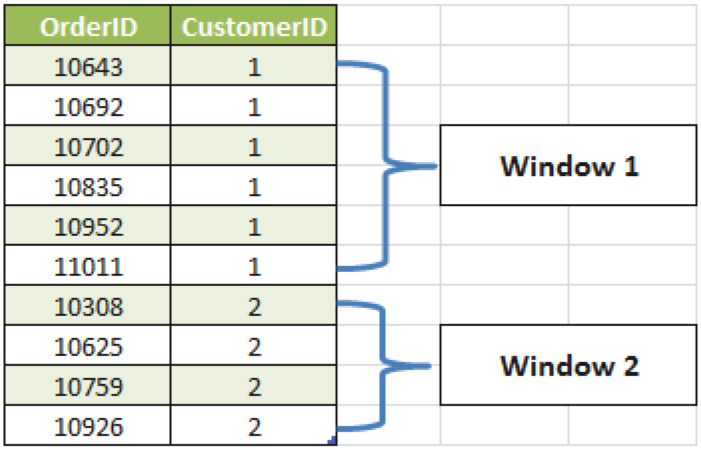
<End expr> is one of:

- *UNBOUNDED FOLLOWING*: The window ends in the last row of the partition
- *CURRENT ROW*: The window ends in the current row
- *<unsigned integer literal> PRECEDING* or *FOLLOWING*

Where it is not explicitly specified, the default window frame is "*range between unbounded preceding and current row*", in other words, the top row in the window is the first row in the current partition, and the bottom row in the window is the current row.

To see all this theory in practice, let's consider the following query on the table Orders from the Northwind database:

```
USE NorthWind
GO
SELECT OrderID, CustomerID
FROM Orders
WHERE CustomerID IN (1,2)
```

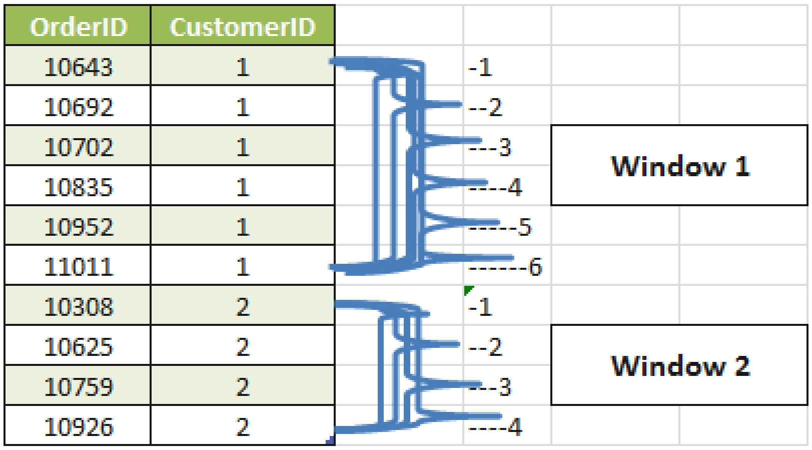


OrderID	CustomerID
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1
10308	2
10625	2
10759	2
10926	2

The result of the query returns two customers and their respective orders. In the picture we can see two windows partitioned by CustomerID, where CustomerID = 1 and CustomerID = 2.

Even then, this is not a 100% correct representation of a window, but it's easier to understand it when we look at the window as it is in the picture; where the windows correspond to each distinct CustomerID.

A more correct picture of a window might be the following:

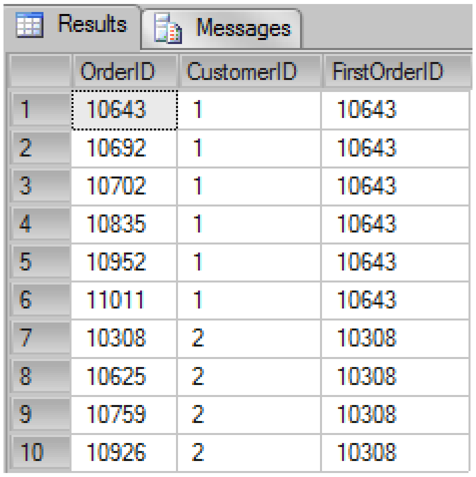


OrderID	CustomerID
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1
10308	2
10625	2
10759	2
10926	2

Considering that the first window has 6 rows, we have 6 windows that coexist. Because they coexist, it's easy to implement the frame in a window. I could tell that a window goes from *unbounded preceding to 3 following rows*.

I can understand if this is not quite so straightforward to understand, so let's, instead, see some examples: Let's suppose that I want to return the first OrderID of each window above. I could write something like the following query:

```
SELECT OrderID, CustomerID,
    FIRST_VALUE(OrderID) OVER(PARTITION BY CustomerID ORDER BY OrderID) AS FirstOrderID
FROM Orders
WHERE CustomerID IN (1,2)
```

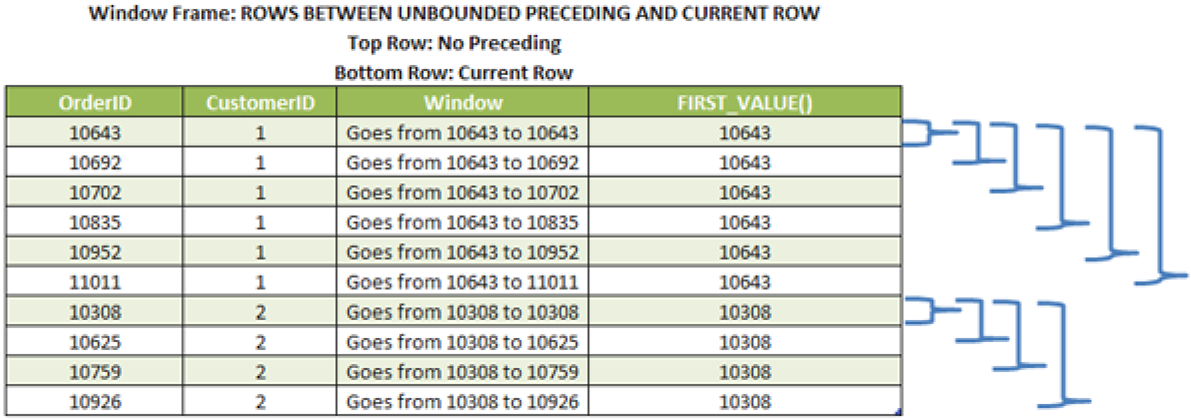


	OrderID	CustomerID	FirstOrderID
1	10643	1	10643
2	10692	1	10643
3	10702	1	10643
4	10835	1	10643
5	10952	1	10643
6	11011	1	10643
7	10308	2	10308
8	10625	2	10308
9	10759	2	10308
10	10926	2	10308

Remember, if I don’t specify the window frame clause, then the default is *“range between unbounded preceding and current row”*. In other words, the query above is equivalent to the following:

```
SELECT OrderID, CustomerID,
    FIRST_VALUE(OrderID) OVER(PARTITION BY CustomerID
                    ORDER BY OrderID
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND CURRENT ROW) AS FirstOrderID
FROM Orders
WHERE CustomerID IN (1,2)
```

To try to make things clearer, here’s an illustration of the way that the top and bottom boundary works in the query above (using the function FIRST_VALUE). This would work something like this:


Window Frame: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
Top Row: No Preceding
Bottom Row: Current Row

OrderID	CustomerID	Window	FIRST_VALUE()
10643	1	Goes from 10643 to 10643	10643
10692	1	Goes from 10643 to 10692	10643
10702	1	Goes from 10643 to 10702	10643
10835	1	Goes from 10643 to 10835	10643
10952	1	Goes from 10643 to 10952	10643
11011	1	Goes from 10643 to 11011	10643
10308	2	Goes from 10308 to 10308	10308
10625	2	Goes from 10308 to 10625	10308
10759	2	Goes from 10308 to 10759	10308
10926	2	Goes from 10308 to 10926	10308

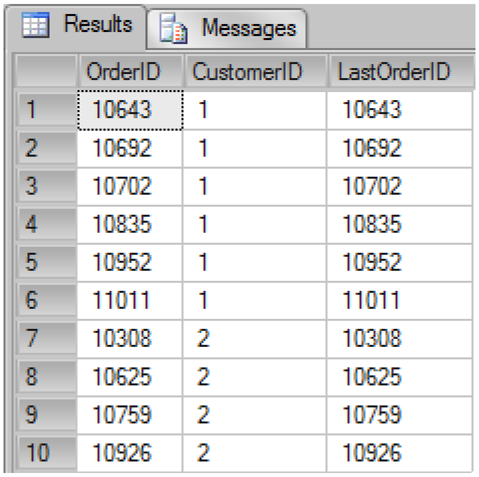
In the picture, we can see that the fist value of the first window is 10643 and the first value of the next window is also 10643, the top row specified in the frame (no preceding) says that it is unbounded preceding.

A good example of how the window frame works is the function LAST_VALUE, because we need to change the default frame in order to really return the last value of a partition.

This function can be a little confusing at first, but as soon we understood the window frame we can see that the action that the function performs by default is correct. It’s very common to test the last_value function and think that this is not working properly, and some people even demand a “fix”for the function because they reckon that it is not working correctly.

Let’s see what the function returns for the same sample in the Orders table:

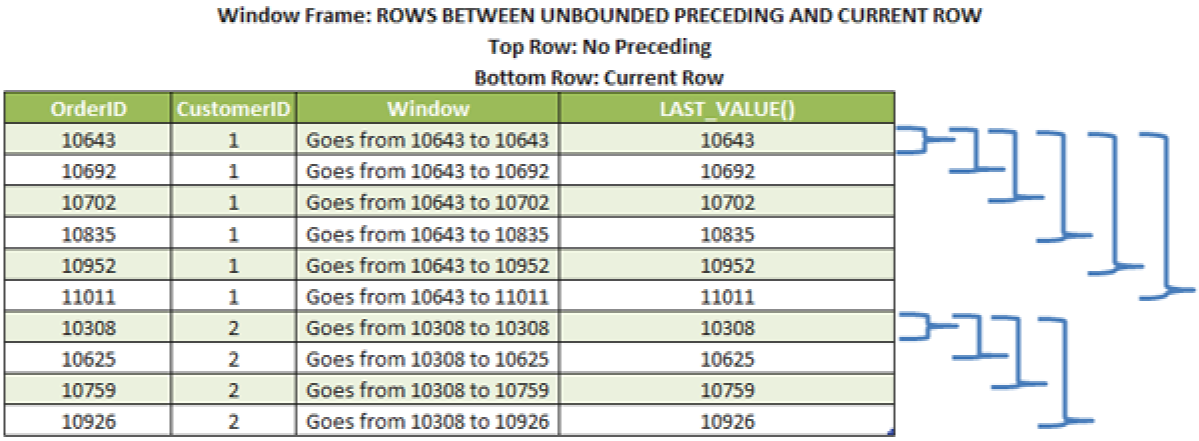
```
SELECT OrderID, CustomerID,
    LAST_VALUE(OrderID) OVER(PARTITION BY CustomerID
                    ORDER BY OrderID) AS FirstOrderID
FROM Orders
WHERE CustomerID IN (1,2)
```



	OrderID	CustomerID	LastOrderID
1	10643	1	10643
2	10692	1	10692
3	10702	1	10702
4	10835	1	10835
5	10952	1	10952
6	11011	1	11011
7	10308	2	10308
8	10625	2	10625
9	10759	2	10759
10	10926	2	10926

As we can see, the result was not what one might expect. I wanted to return the last OrderID of each customer, and SQL Server is returning the actual (current row ?) OrderID for each row.

Let's see the same illustration we saw earlier with the first_value function but now using the LAST_VALUE concept:



Window Frame: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
Top Row: No Preceding
Bottom Row: Current Row

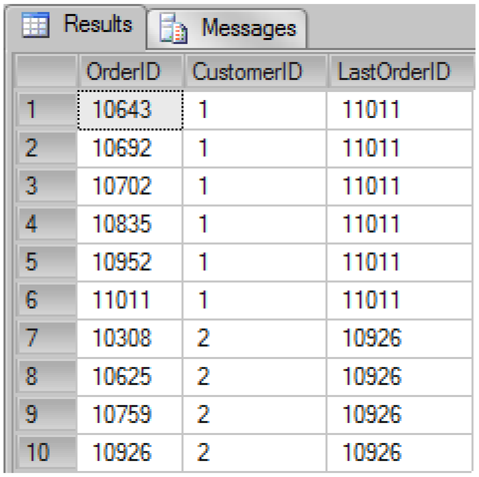
OrderID	CustomerID	Window	LAST_VALUE()
10643	1	Goes from 10643 to 10643	10643
10692	1	Goes from 10643 to 10692	10692
10702	1	Goes from 10643 to 10702	10702
10835	1	Goes from 10643 to 10835	10835
10952	1	Goes from 10643 to 10952	10952
11011	1	Goes from 10643 to 11011	11011
10308	2	Goes from 10308 to 10308	10308
10625	2	Goes from 10308 to 10625	10625
10759	2	Goes from 10308 to 10759	10759
10926	2	Goes from 10308 to 10926	10926

- Remember: By not specifying the window frame clause in the query:
    - It is using the default frame and it will use the bottom row as the current row
    - It will return the value of the actual row as the last value.
    - Look at the blue brackets, you'll see that the size of the sliding window is limited to the current row.

The difference when specifying the frame as unbounded:

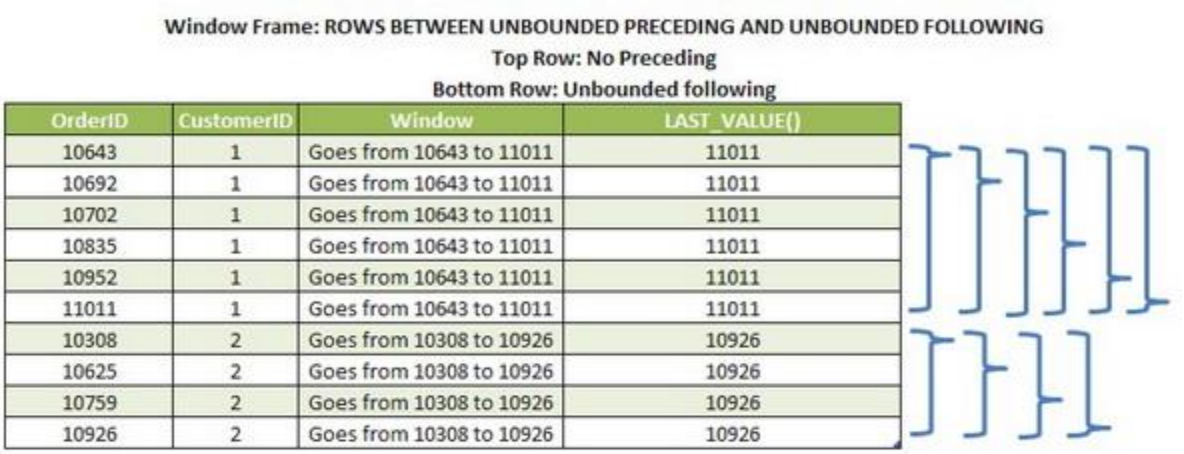
```
SELECT OrderID, CustomerID,
    LAST_VALUE(OrderID) OVER(PARTITION BY CustomerID
                    ORDER BY OrderID
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND UNBOUNDED FOLLOWING) AS FirstOrderID
FROM Orders
```

WHERE CustomerID IN (1,2)



	OrderID	CustomerID	LastOrderID
1	10643	1	11011
2	10692	1	11011
3	10702	1	11011
4	10835	1	11011
5	10952	1	11011
6	11011	1	11011
7	10308	2	10926
8	10625	2	10926
9	10759	2	10926
10	10926	2	10926

Now we've got the expected result. Let's see how the illustration would be for this scenario:



Window Frame: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
Top Row: No Preceding
Bottom Row: Unbounded following

OrderID	CustomerID	Window	LAST_VALUE()
10643	1	Goes from 10643 to 11011	11011
10692	1	Goes from 10643 to 11011	11011
10702	1	Goes from 10643 to 11011	11011
10835	1	Goes from 10643 to 11011	11011
10952	1	Goes from 10643 to 11011	11011
11011	1	Goes from 10643 to 11011	11011
10308	2	Goes from 10308 to 10926	10926
10625	2	Goes from 10308 to 10926	10926
10759	2	Goes from 10308 to 10926	10926
10926	2	Goes from 10308 to 10926	10926

The window frame goes from *unbounded preceding* to *unbounded following*. In other words, when SQL Server reads the last value of a window, it goes on until the *unbounded following* that is the last row in the partition.

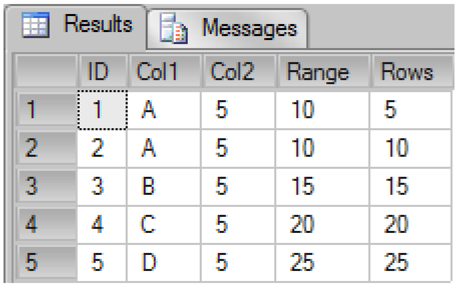
RANGE versus ROW

Another confusing thing about the window frame is the RANGE versus ROW. The question is, what is the difference between them?

- The ROWS clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row.
- 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 specified by 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.

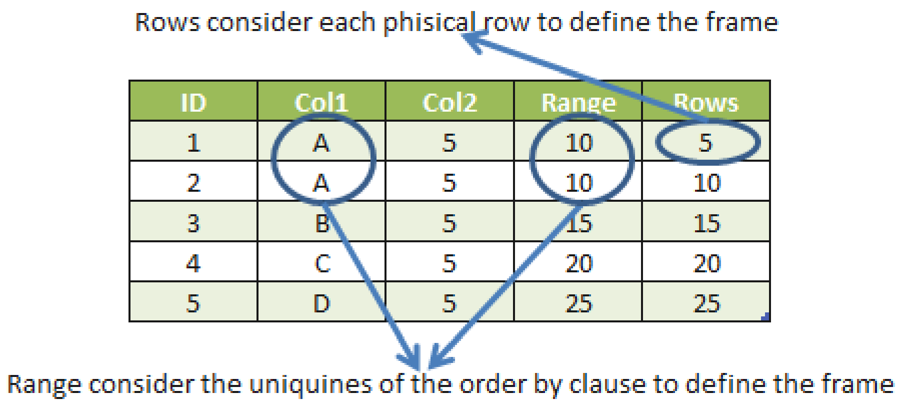
Example:

```
USE tempdb
GO
IF OBJECT_ID('tempdb.dbo.#TMP')  IS NOT NULL
  DROP TABLE #TMP
GO
CREATE TABLE #TMP (ID INT, Col1 CHAR(1), Col2 INT)
GO

INSERT INTO #TMP VALUES(1,'A', 5), (2, 'A', 5), (3, 'B', 5), (4, 'C', 5), (5, 'D', 5)
GO
--SELECT * FROM #TMP

SELECT *,
     SUM(Col2) OVER(ORDER BY Col1 RANGE UNBOUNDED PRECEDING) "Range"
     SUM(Col2) OVER(ORDER BY Col1 ROWS UNBOUNDED PRECEDING) "Rows"   FROM #TMP
```



	ID	Col1	Col2	Range	Rows
1	1	A	5	10	5
2	2	A	5	10	10
3	3	B	5	15	15
4	4	C	5	20	20
5	5	D	5	25	25

We have two running calculations in this query. One of these is using the RANGE frame and the other is using the ROWS frame. You can see that the result for the value “A” is different for each frame.



Rows consider each phisical row to define the frame

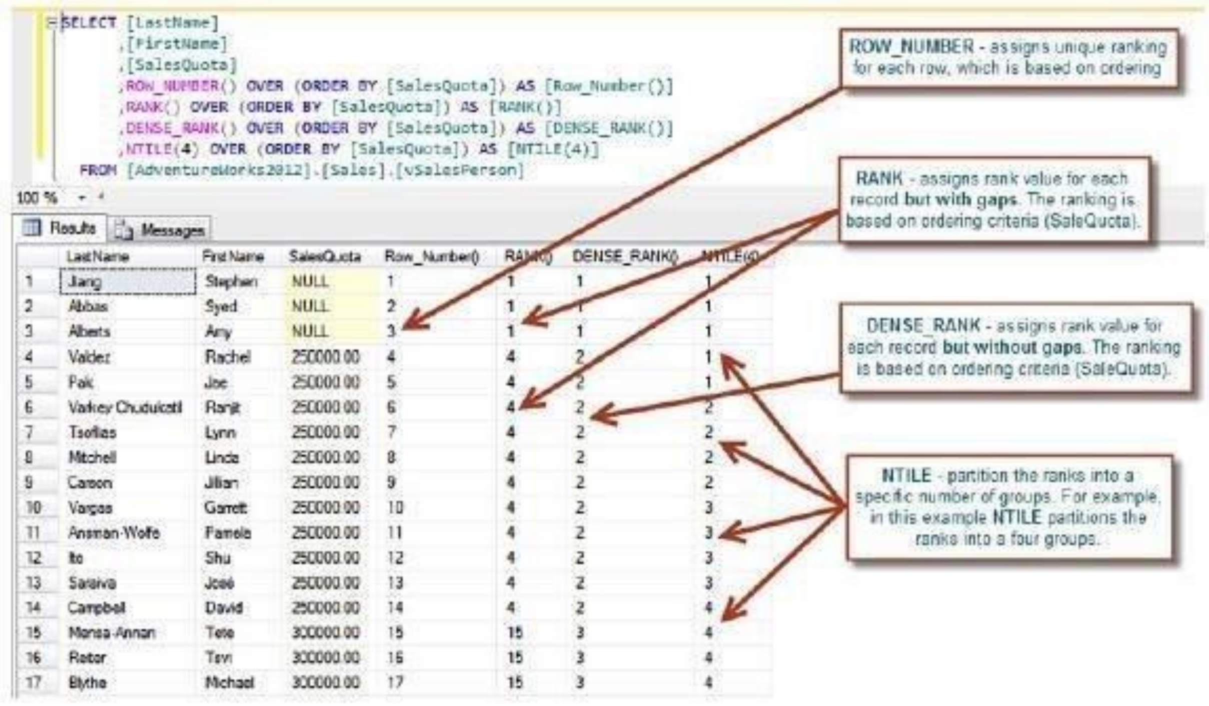
ID	Col1	Col2	Range	Rows
1	A	5	10	5
2	A	5	10	10
3	B	5	15	15
4	C	5	20	20
5	D	5	25	25

Range consider the uniquines of the order by clause to define the frame

We can see in the picture that the aggregation works differently depending on the frame.

There is another ***very important*** difference between the ROW and RANGE clause: the RANGE clause ***always*** uses an on-disk window to process the window spool operator.
The most important factor that affects the performance of window functions.

- A window spool operator has two alternative ways of storing the frame data, with the in–memory worktable or with a disk–based worktable. You'll have a huge difference on performance according to the way that the query processor is executing the operator.
- The in-memory worktable is used when you define the frame as ROWS and it is lower than 10000 rows. If the frame is greater than 10000 rows, then the window spool operator will work with the on-disk worktable.
- The on-disk based worktable is used with the default frame, that is, "range…"and a frame with more than 10000 rows.



```
SELECT [LastName]
      ,[FirstName]
      ,[SalesQuota]
      ,ROW_NUMBER() OVER (ORDER BY [SalesQuota]) AS [Row_Number()]
      ,RANK() OVER (ORDER BY [SalesQuota]) AS [RANK()]
      ,DENSE_RANK() OVER (ORDER BY [SalesQuota]) AS [DENSE_RANK()]
      ,NTILE(4) OVER (ORDER BY [SalesQuote]) AS [NTILE(4)]
  FROM [AdventureWorks2012].[Sales].[vSalesPerson]
```

	LastName	FirstName	SalesQuota	Row_Number()	RANK()	DENSE_RANK()	NTILE(4)
1	Jiang	Stephen	NULL	1	1	1	1
2	Abbas	Syed	NULL	2	1	1	1
3	Alberts	Amy	NULL	3	1	1	1
4	Valdez	Rachel	250000.00	4	4	2	1
5	Pak	Jae	250000.00	5	4	2	1
6	Varkey Chudukatil	Ranjit	250000.00	6	4	2	2
7	Tsoflias	Lynn	250000.00	7	4	2	2
8	Mitchell	Linda	250000.00	8	4	2	2
9	Carson	Jillian	250000.00	9	4	2	2
10	Vargas	Garrett	250000.00	10	4	2	3
11	Ansman-Wolfe	Pamela	250000.00	11	4	2	3
12	Ito	Shu	250000.00	12	4	2	3
13	Saraiva	José	250000.00	13	4	2	3
14	Campbell	David	250000.00	14	4	2	4
15	Mensa-Annan	Tete	300000.00	15	15	3	4
16	Reiter	Tsvi	300000.00	16	15	3	4
17	Blythe	Michael	300000.00	17	15	3	4

ROW_NUMBER - assigns unique ranking for each row, which is based on ordering

RANK - assigns rank value for each record **but with gaps**. The ranking is based on ordering criteria (SaleQuota).

DENSE_RANK - assigns rank value for each record **but without gaps**. The ranking is based on ordering criteria (SaleQuota).

NTILE - partition the ranks into a specific number of groups. For example, in this example NTILE partitions the ranks into a four groups.

Examples: https://www.red-gate.com/simple-talk/sql/learn-sql-server/window-functions-in-sql-server-part-2-the-frame/

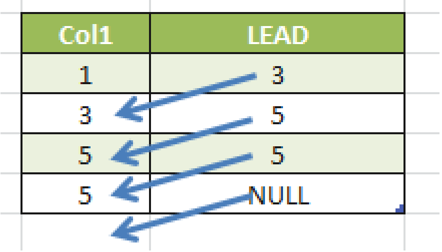
The script to create a table to test the functions:

```
USE TempDB
GO
IF OBJECT_ID('Tab1') IS NOT NULL DROP TABLE Tab1
GO
CREATE TABLE Tab1 (Col1 INT)
GO
INSERT INTO Tab1 VALUES(5), (5), (3) , (1)
GO
```

LEAD()

The LEAD function is used to read a value from the next row, or the row below the actual row. If the next row doesn't exist, then NULL is returned.

```
-- LEAD
SELECT Col1,
      LEAD(Col1) OVER(ORDER BY Col1) AS "LEAD()"   FROM Tab1
```

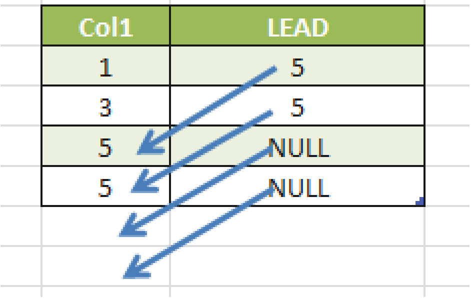


Col1	LEAD
1	3
3	5
5	5
5	NULL

As we can see, the LEAD column has the next row value, but NULL is returned in the last row.

By default, the next row is returned; but you can change this behavior by specifying a parameter to read N following rows, for instance:

```
-- LEAD
SELECT Col1,
      LEAD(Col1, 2) OVER(ORDER BY Col1) AS "LEAD()"   FROM Tab1
```



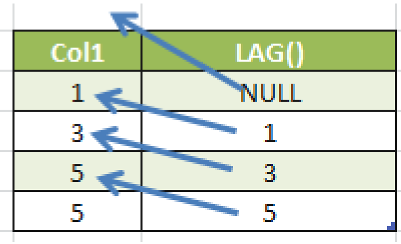
Col1	LEAD
1	5
3	5
5	NULL
5	NULL

In this last query, I used the parameter "2" in the function in order to specify that I want to read the second row after the current row.

LAG()

The LAG() function is similar to the LEAD() function, but it returns the row before the actual row rather than return the next row. For instance:

```
-- LAG
SELECT Col1,
    LAG(Col1, 2) OVER(ORDER BY Col1) AS "LAG()"   FROM Tab1
```



Col1	LAG()
1	NULL
3	1
5	3
5	5

As we can see, the function returns the value of the row before the actual row; but when the row before doesn't exist, then NULL is returned.

You may be wondering whether I could do the same thing by using the function LEAD() with a negative parameter (OffSet): In other words, instead of read 1 following value I could read -1 following value.

Let's see a sample:

```
SELECT Col1,
    LEAD(Col1, -1) OVER(ORDER BY Col1) AS "LEAD() as LAG()"   FROM Tab1
```

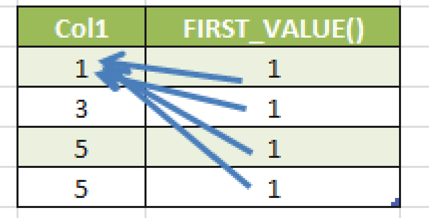
```
Msg 8730, Level 16, State 1, Line 1
Offset parameter for Lag and Lead functions cannot be a negative value.
```

As we can see, we cannot use negative values in this parameter to the function.

FIRST_VALUE()

As the name says, FIRST_VALUE() returns the first value in a partition window. For instance:

```
-- FIRST_VALUE
SELECT Col1,
    FIRST_VALUE(Col1) OVER(ORDER BY Col1) AS "FIRST_VALUE()"   FROM Tab1
```

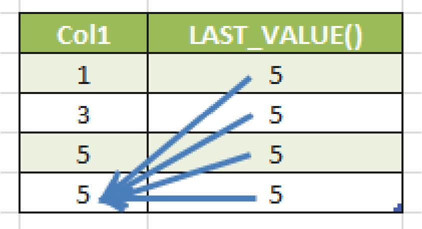


Col1	FIRST_VALUE()
1	1
3	1
5	1
5	1

LAST_VALUE()

Also, as the name suggests, it returns the last value in a partition window: For instance:

```
-- LAST_VALUE
SELECT Col1,
LAST_VALUE(Col1) OVER(ORDER BY Col1
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
AS "LAST_VALUE()"   FROM Tab1
```



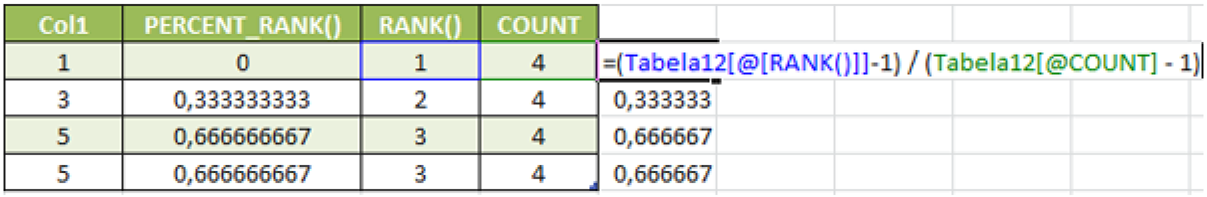
Col1	LAST_VALUE()
1	5
3	5
5	5
5	5

To get the last value in the partition window I’ve specified a different frame. This is what the words “ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING” mean.

PERCENT_RANK()

The PERCENT_RANK() function is very similar to the RANK() function, but the values that are returned range between 0 and 1.

```
SELECT Col1,
    PERCENT_RANK() OVER(ORDER BY Col1) AS "PERCENT_RANK()"
    RANK() OVER(ORDER BY Col1) AS "RANK()"
    (SELECT COUNT(*) FROM Tab1) "COUNT"   FROM Tab1
```



Col1	PERCENT_RANK()	RANK()	COUNT	
1	0	1	4	=(Tabela12[@[RANK()]]-1) / (Tabela12[@COUNT] - 1)
3	0,333333333	2	4	0,333333
5	0,666666667	3	4	0,666667
5	0,666666667	3	4	0,666667

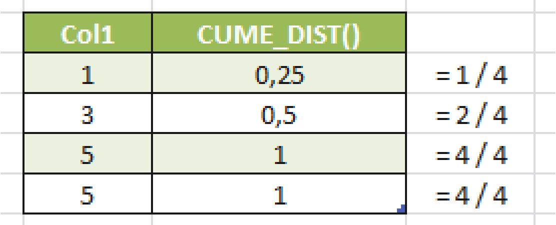
You can do the same thing by calculating the percent rank using the following formula:

- (RANK() – 1) / (NumberOfRows – 1)

CUME_DIST()

The function CUME_DIST() is also used to calculate a rank from 0 to 1 based on the position of the row in the rank, for instance:

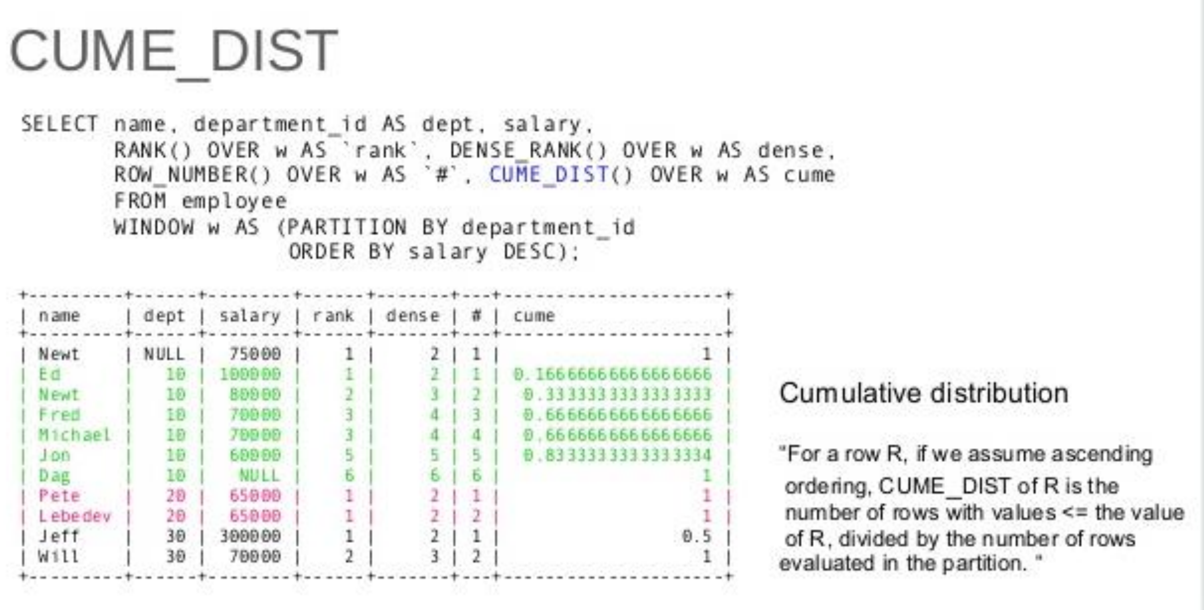
```
-- CUME_DIST()
SELECT Col1,
    CUME_DIST() OVER(ORDER BY Col1) AS "CUME_DIST()"   FROM Tab1
```



Col1	CUME_DIST()	
1	0,25	= 1 / 4
3	0,5	= 2 / 4
5	1	= 4 / 4
5	1	= 4 / 4

The same behavior could be achieved by using the following formula:

- COUNT(*) OVER (ORDER BY Col1) / COUNT(*) OVER ()



CUME_DIST

```
SELECT name, department_id AS dept, salary,
       RANK() OVER w AS `rank`, DENSE_RANK() OVER w AS dense,
       ROW_NUMBER() OVER w AS `#`, CUME_DIST() OVER w AS cume
       FROM employee
       WINDOW w AS (PARTITION BY department_id
                    ORDER BY salary DESC);
```

```
+---------+------+--------+------+-------+---+--------------------+
| name    | dept | salary | rank | dense | # | cume               |
+---------+------+--------+------+-------+---+--------------------+
| Newt    | NULL |  75000 |    1 |     2 | 1 |                  1 |
| Ed      |   10 | 100000 |    1 |     2 | 1 | 0.1666666666666666 |
| Newt    |   10 |  80000 |    2 |     3 | 2 |  0.333333333333333 |
| Fred    |   10 |  70000 |    3 |     4 | 3 |  0.666666666666666 |
| Michael |   10 |  70000 |    3 |     4 | 4 |  0.666666666666666 |
| Jon     |   10 |  60000 |    5 |     5 | 5 |  0.833333333333334 |
| Dag     |   10 |   NULL |    6 |     6 | 6 |                  1 |
| Pete    |   20 |  65000 |    1 |     2 | 1 |                  1 |
| Lebedev |   20 |  65000 |    1 |     2 | 2 |                  1 |
| Jeff    |   30 | 300000 |    1 |     2 | 1 |                0.5 |
| Will    |   30 |  70000 |    2 |     3 | 2 |                  1 |
+---------+------+--------+------+-------+---+--------------------+
```

Cumulative distribution

"For a row R, if we assume ascending ordering, CUME_DIST of R is the number of rows with values <= the value of R, divided by the number of rows evaluated in the partition. "

References

1. SELECT - OVER Clause (Transact-SQL): https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-2017
2. Aggregate Functions: https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql?view=sql-server-2017
3. Analytic Functions (Transact-SQL): https://docs.microsoft.com/en-us/sql/t-sql/functions/analytic-functions-transact-sql?view=sql-server-2017
4. How to Use Microsoft SQL Server 2012's Window Functions, Part 1: http://www.itprotoday.com/microsoft-sql-server/how-use-microsoft-sql-server-2012s-window-functions-part-1
5. Window Functions in SQL Server: Part 2-The Frame: https://www.red-gate.com/simple-talk/sql/learn-sql-server/window-functions-in-sql-server-part-2-the-frame/
6. Window Functions in SQL Server: Part 2-The Frame: https://www.red-gate.com/simple-talk/sql/learn-sql-server/window-functions-in-sql-server-part-2-the-frame/
7. Window Functions in SQL Server: Part 3: Questions of Performance: https://www.red-gate.com/simple-talk/sql/learn-sql-server/window-functions-in-sql-server-part-3-questions-of-performance/
8. Another sample database – TSQL2012 – zip file: http://tsql.solidq.com/books/source_code/TSQL2012.zip