# Installation

**SQL server Express 2019**

**SQL Server Management Studio (SSMS)**

* Graphical tool for interacting with a database.

**Sample Database (Adventure Work)**

* Download .bak file from Microsoft website.
* ***.bak*** file is backup file which can be used to restore a database.
* A backup is essential of a database to a database server.

# **Window Function**

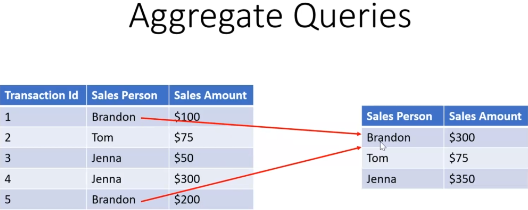
## **Introduction**

**As we worked with Aggregate queries in SQL and know how useful they can be in performing calculations like count, Max etc.**

* While optionally grouping the results of these calculations by other values in the data.

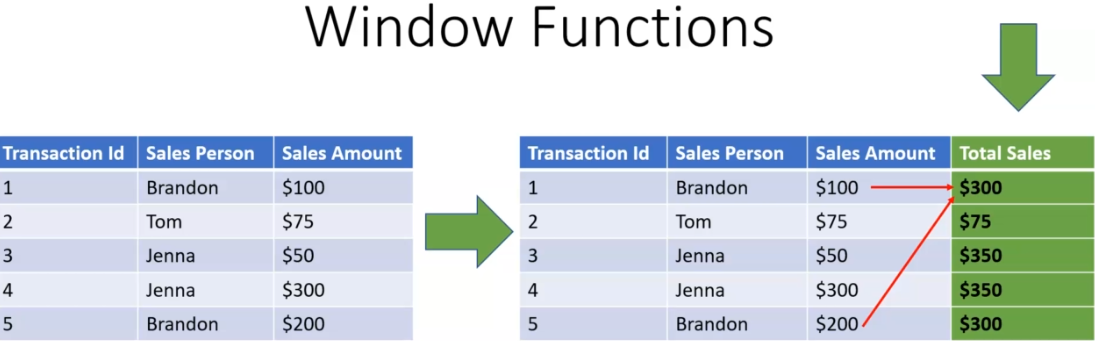
**However, these functions also force us to compromise the level of detail available to us and our query output.**

* For example, if we want to see account of all the rows in your table not broken up into groups, we won't actually be able to see these rows.
* Instead, your output will simply consist of the count and nothing else.

**Whereas aggregate function collapse the rows of our data into groups based on the unique values and columns we include in our select query.**

**So you can see that the two records for Brandon in our detailed data example above.**

* Have collapsed down into one single record where those two sales amounts have been added up to one number.

**Window functions allow you to retain low level detail while performing the same types of calculations.**

**The aggregate calculation simply becomes another column in your query output.**

* Now, again, you see that we've got two records for Brandon here in our detailed data, one with a one hundred dollar sales amount, and one with a two hundred dollar sales amount.
* But in our window function version, we have this new Column total sales where both records for Brandon reflect that total sales amount for Brandon.
* So we get the same result that we got with the aggregate query in terms of calculating Brandon's total sales.
* But we don't have to collapse our data down based on including that aggregate calculation and can just tack it on as a new column in our query output.

## **Syntax**

**SUM(SalesYTD) Over()**

**The first piece is just a function, it could be some count max or any other aggregate function or one of a number of ranking functions.**

* The second piece of the syntax is what makes this a window function, the ***over*** keyword.
* Now, when no arguments are passed over, as in this case, it simply applies are some function here across the entire dataset returning the result for each record in the data set.

**It Allow us to include aggregate calculation in out queries, WITHOUT changing the output in any way.**

* **The aggregate calculation is simple tacked on the query as an additional column.**

**It is possible to group these calculation, just like we can with aggregate queries.**

## **Practical**

**-- Window function**

**SELECT BusinessEntityID**

**,TerritoryID**

**,SalesQuota**

**,Bonus**

**,CommissionPct**

**,SalesYTD**

**,SalesLastYear**

**,[Total YID Sales] = Sum(SalesYTD) over ()**

**,[MAX YID Sales] = MAX(SalesYTD) over ()**

**,[% of Best Performer] = [SalesYTD]/MAX(SalesYTD) over ()**

**FROM Sales.SalesPerson**

**-- Aggregate function**

**SELECT SUM(SalesYTD) as [Total YID Sales], MAX(SalesYTD) as [MAX YID Sales]**

**--,([SalesYTD]/MAX(SalesYTD)) as [% of Best Performer]**

**FROM Sales.SalesPerson**

## **Partition By**

### **Introduction**

The *PARTITION BY* clause divides a query’s result set into partitions.

* The window function is operated on each partition separately and recalculate for each partition.

**A PARTITION BY clause is used to partition rows of table into groups.**

**It is useful when we have to perform a calculation on individual rows of a group using other rows of that group.**

* It is always used inside OVER() clause.
* The partition formed by partition clause are also known as Window.
* This clause works on windows functions ***only***. Like- RANK(), LEAD(), LAG() etc.
* *If this clause is omitted in OVER() clause, then whole table is considered as a single partition.*

**We've seen how to perform aggregate calculations across our entire data set without losing row level visibility into our data using *over*.**

**Now we wanted to roll those aggregate totals up into groups the same way we do with group by using aggregate functions, by using *partition by*.**

**It allows us to compute aggregate totals for groups within our data, while still retaining row-level details.**

* When we combine our ***over*** functioning with ***partition by***, we now have the power to compute aggregate totals for groups within our data while still retaining that row level detail that distinguishes window functions.

**It assigns each row of our queries output to a group, without collapsing our data into fewer rows as with *Group BY*.**

**The *first step* partition by takes is to assign each row of your query output to a group, but it does this behind the scenes without collapsing your data into fewer rows, as with grouped by.**

* Instead of groups being assigned based on the distinct values of all the non-segregated columns of our data, we have to specify the columns these groups will be based on.

### **Syntax**

**window\_function ( expression ) OVER (**

***PARTITION BY* expression1, expression2, ...**

**order\_clause**

**frame\_clause**

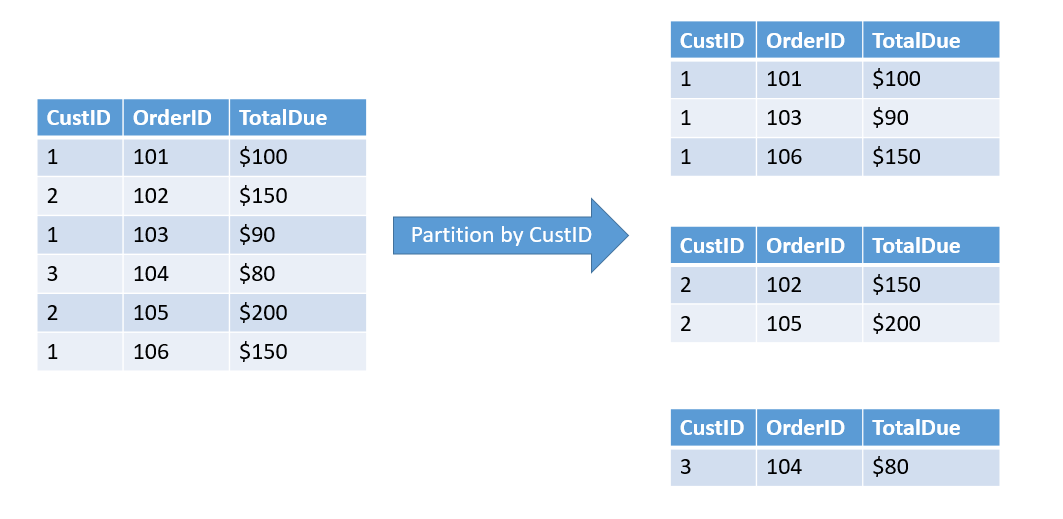
**)**

**You can specify one or more columns or expressions to partition the result set.**

* The expression1, expression1, etc., can only refer to the columns derived by the FROM clause. They cannot refer to expressions or aliases in the select list.
* The expressions of the PARTITION BY clause can be column expressions, scalar subquery, or scalar function.

**Note that a scalar subquery and scalar function always returns a single value.**

**If you omit the PARTITION BY clause, the whole result set is treated as a *single partition*.**

**Example:**

### **Practical**

/\*

Fetch Sum of lineTotal

Grouping our salesorder table, detailed data by product\_Id

and Order quantity using traditional aggregate function

\*/

select

ProductID,

OrderQty,

[LineTotal] = SUM(LineTotal)

from Sales.SalesOrderDetail

Group by ProductID,OrderQty

/\*

This operation works by creating a group for each unique combination of values

from all the fields listed in our group by product ID in order quantity.

the aggregate calculation or calculations of our choosing, some count, etc.

will then be performed over each of these groups. (ProductID,OrderQty)

\*/

-- Sum of linesTotal via Over function

select

ProductID,

SalesOrderID,

SalesOrderDetailID,

OrderQty,

UnitPrice,

UnitPriceDiscount,

[LineTotal] = SUM(LineTotal) over()

from Sales.SalesOrderDetail

order by ProductID,OrderQty

-- Sum of linesTotal via Over function

select

ProductID,

SalesOrderID,

SalesOrderDetailID,

OrderQty,

UnitPrice,

UnitPriceDiscount,

LineTotal,

[ProductIDLineTotal] = SUM(LineTotal) over(Partition by ProductID,OrderQty)

from Sales.SalesOrderDetail

order by ProductID,OrderQty

/\*

every row in our query output with this combination of values

is simply considered a member of the same group.

the window function is basically returning the sum of line total for each of these rows,

as our aggregate function did for the entire group.

we are able to perform aggregate calculations on groups within our data.

without losing any road level detail or the flexibility to include any columns we want, and if we want

to add more layers to our groups, it's as easy as tacking on an extra field onto our partition by clause.

\*/

## **Row\_Number**

### How To Generate Row Number Or Sequence Number Using Hana Graphical ...**Introduction**

**It is a window function that assigns a sequential integer number to each row in the query’s result set.**

### **Syntax**

**ROW\_NUMBER() OVER (**

**[PARTITION BY partition\_expression, ... ]**

**ORDER BY sort\_expression [ASC | DESC], ...**

**)**

**The PARTITION BY clause divides the result set into partitions (another term for groups of rows).**

* The ***ROW\_NUMBER() function*** is applied to each partition separately and reinitialized the row number for each partition.
* The PARTITION BY clause is optional.
* If you skip it, the ROW\_NUMBER() function will treat the whole result set as a single partition.

**The ORDER BY clause defines the logical order of the rows within each partition of the result set.**

* The ORDER BY clause is mandatory because the ROW\_NUMBER() function is order sensitive.

### **Practical**

Use AdventureWorks2019

-- Ranking all records within each group of salesOrder IDs.

select

SalesOrderID,

SalesOrderDetailID,

UnitPriceDiscount,

LineTotal,

[ProductIDLineTotal] = SUM(LineTotal) over(Partition by ProductID,OrderQty),

[Ranking] = ROW\_NUMBER() over(Partition by SalesOrderID order by LineTotal desc)

from Sales.SalesOrderDetail

order by SalesOrderID

/\*

Row\_number() will assign a unique ranking,

beginning with one to every record within a group,

Our groups are defined by unique combinations of values

from all the fields listed in the partitioned by clause

\*/

-- Row\_Number without partition

select

SalesOrderID,

SalesOrderDetailID,

UnitPriceDiscount,

LineTotal,

[ProductIDLineTotal] = SUM(LineTotal) over(Partition by ProductID,OrderQty),

[Ranking] = ROW\_NUMBER() over(order by LineTotal desc)

from Sales.SalesOrderDetail

order by Ranking

/\*

ROW\_NUMBER() now treats our entire query output as one giant group

in which the record with the highest line total.

we look carefully at the very end of our query output,

There are a number of records which is the exact same 'LineTotal' Value.

So it turns out that Rowe no uses brute force to deal with these kinds of ties,

ROW\_NUMBER() always returns a sequential series of rankings.

It will randomly assign a rank to records whose values

in that order by field be specified.

That's where the rank and dence\_rank functions come in.

\*/