Window Functions

# What is a Window Function?

## Window Function Basics

In our last class, we used subqueries to combine aggregated data with full row details. We did that because aggregate functions (like MAX, COUNT, MIN, et cetera) group rows together and obscures the details. This means it reduces both the rows and the columns that are available to us.

Window functions are functions that can perform these same aggregations without losing the details. The results of the aggregation are applied to each row in the set, and other columns details can be included in the SELECT statement because we do not need to use a GROUP BY clause.

There are three broad categories of window functions – rank functions, aggregate functions, and analytic (or value) functions. Each of these groups of functions are quite different in how they operate. What they have in common is that they work on a window (or partition) of the data set, which is just a subset of the full data set created in the FROM clause.

All window functions define their window using the OVER clause.

**Window functions** are functions that work on a defined set, or “window”, of data then apply the result of that function to all rows in that set.

## OVER

<https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql>

OVER (

[ <PARTITION BY clause> ]

[ <ORDER BY clause> ]

[ <ROW or RANGE clause> ]

)

We define the window for a window function using the OVER clause. The OVER statement can be used on its own, or used in conjunction with a PARTITION BY, ORDER BY, and/or ROW (or RANGE) clause. Each of these clauses help define the window.

Let’s see an example using the SUM() aggregate function, which we are already familiar with.

Working with the WideWorldImporters.Sales.OrderLines table, if we wanted to get the total value of all items ever sold, we could do this using:

USE WideWorldImporters;

GO

SELECT SUM(Quantity \* UnitPrice)

FROM Sales.OrderLines;

This query only returns one row and we can’t see any of the details from the OrderLines table. What if we wanted to display the total sales alongside the details for each row? This is where we would use a window function.

For our demonstration, we’ll add the OrderID, OrderLineID, Description, and a calculation of the total price for that line. Our window in this case is the entire set, so we do not need to add any extra clauses.

SELECT OrderID,

OrderLineID,

Description,

Quantity \* UnitPrice AS TotalLinePrice,

SUM(Quantity \* UnitPrice) OVER() AS TotalSales

FROM Sales.OrderLines

ORDER BY OrderLineID;

Notice that the same amount from our SUM() example is now applied to every row as a new column called “TotalSales”.

## PARTITION BY

Let’s start using the PARTITION BY clause to see how we can create our first window.

In our example above, we got a total of all sales ever made. We’ve decided that’s a little large, so we’d like to see the totals broken up by the month they were sold. Since order dates are a property of the order, we’ll need to include the Orders table this time.

As before, let’s first see how we would do this without a window function. We will group by the order year, using the YEAR() and MONTH() built-in date functions, then sum the totals the same way as before.

SELECT YEAR(o.OrderDate) AS OrderYear,

MONTH(o.OrderDate) AS OrderMonth,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

GROUP BY YEAR(o.OrderDate), MONTH(o.OrderDate)

ORDER BY OrderYear, OrderMonth;

Notice that because our YEAR() and MONTH() functions are scalar functions, not aggregate functions, they have to be included in our GROUP BY clause.

As before, we want to get all the rows, not an aggregated set of rows. We also want access to column details, similar to what we had before. We’re going to rewrite this same query using a window function. We will have to use PARTITION BY to set up our window, similar to how we used GROUP BY above.

SELECT o.OrderID,

o.OrderDate,

ol.OrderLineID,

ol.Description,

ol.Quantity \* ol.UnitPrice AS TotalLinePrice,

SUM(ol.Quantity \* ol.UnitPrice) OVER ( PARTITION BY YEAR(o.OrderDate), MONTH(o.OrderDate) ) AS MonthlyTotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

ORDER BY OrderDate;

Scroll through the results. Notice that all of the “MonthlyTotalSales” values are identical until you move to a new month. The year and month are our window – it is how we are partitioning the data set created in our FROM clause.

## Aggregate Functions

<https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql>

We are already fairly familiar with aggregate functions available as window functions – they’re the same aggregate functions we’ve already been using with the GROUP BY clause. The ones we are using most commonly are:

**AVG** Returns the mean of a set of numbers.  
**MIN** Returns the lowest value in a set.  
**MAX** Returns the maximum value in a set.  
**SUM** Returns the sum of a set of numbers.  
**COUNT** Returns the number of not null values in a set.

There are others available, but these are the most frequently used.

## Ranking Functions and ORDER BY

<https://docs.microsoft.com/en-us/sql/t-sql/functions/ranking-functions-transact-sql>

Ranking functions allow us to rank rows within a window. There are currently four functions available:

**ROW\_NUMBER**Applies a row number to each row, as they appear in the sort order. Identical values will be given different row numbers, based on which happens to appear first.  
 **RANK**Similar to row number, but identical values are given the same rank. This will result in gaps in the ranking. For example, if the top two rows in the list are tied, the rank for the first three rows would be 1, 1, 3.

**DENSE\_RANK**Similar to rank, but eliminates the gaps caused by identical values. If the top two rows in the list are tied, the dense rank for the first three rows would be 1, 1, 2.

**NTILE**Returns the current n-tile as defined by an integer passed to the function. For example, you could pass 4 to create quartiles, 5 for quintiles, 10 for deciles, or 100 for percentiles.

To get a meaningful rank, we have to tell SQL Server how to rank our rows by providing a sort order. With window functions, we do that by specifying an ORDER BY clause.

Let’s try to find our most active customers in WideWorldImporters. First, let’s try to write a query that will get us the total number of orders by customer. We want to get names instead of IDs, so let’s join in the customer table as well.

SELECT c.CustomerName,

COUNT(\*) As OrderCount

FROM Sales.Orders o

JOIN Sales.Customers c

ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerName;

This time, let’s use this query as our starting point. To keep things clear, let’s pull this query into a CTE so we can more easily focus on our ranking.

-- Original query in a CTE

WITH OrdersByCust AS (

SELECT c.CustomerName,

COUNT(\*) As OrderCount

FROM Sales.Orders o

JOIN Sales.Customers c

ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerName )

SELECT \*

FROM OrdersByCust;

Let’s add all four rank functions to see them side by side. We will be ranking these customers by descending OrderCount, so that will be in our ORDER BY clauses. We want the customers who order most frequently ranked first, so we will use a descending sort order.

WITH OrdersByCust AS (

SELECT c.CustomerName,

COUNT(\*) As OrderCount

FROM Sales.Orders o

JOIN Sales.Customers c

ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerName )

-- Build our rank query here

SELECT CustomerName,

OrderCount,

ROW\_NUMBER() OVER( ORDER BY OrderCount DESC ) AS RowNum,

RANK() OVER( ORDER BY OrderCount DESC ) AS RankVal,

DENSE\_RANK() OVER( ORDER BY OrderCount DESC ) AS DenseRankVal,

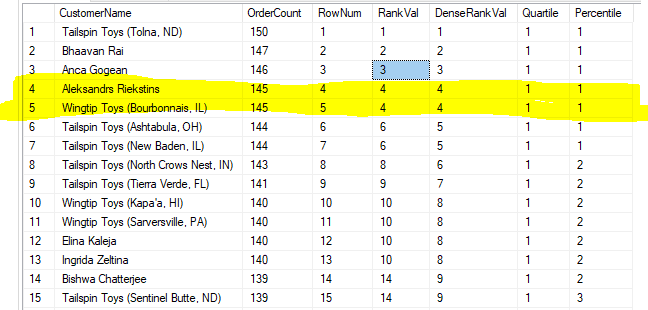
NTILE(4) OVER( ORDER BY OrderCount DESC ) As Quartile,

NTILE(100) OVER( ORDER BY OrderCount DESC ) As Percentile

FROM OrdersByCust

ORDER BY OrderCount DESC;

Let’s review the results, particularly around areas where we have duplicate values:



Row 4 and 5 have the same value. Notice how rows 4, 5 and 6 are treated by the different functions.

ROW\_NUMBER keeps numbering rows, without detecting duplicate values. RANK ranks all rows, but gives duplicate values the same rank number, leaving gaps. DENSE\_RANK closes the gaps left by RANK.

## Analytic Functions

<https://docs.microsoft.com/en-us/sql/t-sql/functions/analytic-functions-transact-sql>

Analytic functions are powerful functions that are quite unlike anything else we’ve worked with so far. Using analytic functions allows us to retrieve values from other rows in the same window. Just like our ranking functions, analytic functions need an order to be meaningful.

There are several functions available, but we’re just going to focus on a few:

**LEAD** Gets a value from the next row in window.

**LAG** Gets a value from the previous row in window.

**FIRST\_VALUE** Gets a value from the first row in a window.

Let’s rank our salespeople by the number of sales they’ve made, then try to compare them to each other. First, let’s build a query to get total sales by salesperson.

SELECT p.FullName,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

JOIN Application.People p

ON o.SalespersonPersonID = p.PersonID

GROUP BY p.FullName;

As before, we’re going to put this query in a CTE so we can more easily work with it and see our analytic functions in action.

WITH SalesBySalesperson AS (

SELECT p.FullName,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

JOIN Application.People p

ON o.SalespersonPersonID = p.PersonID

GROUP BY p.FullName

)

SELECT FullName,

TotalSales AS MySales,

LEAD(TotalSales) OVER ( ORDER BY TotalSales DESC ) AS NextLowest,

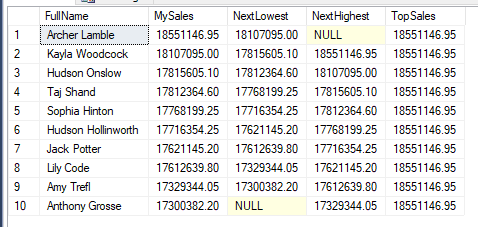
LAG(TotalSales) OVER ( ORDER BY TotalSales DESC ) As NextHighest,

FIRST\_VALUE(TotalSales) OVER ( ORDER BY TotalSales DESC ) AS TopSales

FROM SalesBySalesperson

ORDER BY TotalSales DESC;

Let’s look at the results to see how these functions behave:



Notice the NULL for LEAD (NextLowest) and LAG (NextHighest). This is happening because there is no next/previous row in our window.

# Working with Windows

We have looked at ORDER BY and PARTITION BY independently. Now let’s work with some complex examples that combine the two. This will allow us to see how analytic functions and ranking functions work within windows.

## Setup

For these examples, we are going to work with monthly sales by salesperson. We can do this by modifying our previous example to add in year and month.

WITH MonthlySalesBySalesperson AS (

SELECT p.FullName,

YEAR(o.OrderDate) AS SalesYear,

MONTH(o.OrderDate) AS SalesMonth,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

JOIN Application.People p

ON o.SalespersonPersonID = p.PersonID

GROUP BY p.FullName, YEAR(o.OrderDate), MONTH(o.OrderDate)

)

SELECT \*

FROM MonthlySalesBySalesperson

ORDER BY SalesYear, SalesMonth

## Example 1

We want to see a monthly ranking of our salespeople, based on descending sales. We’ll use our CTE from above as the source. This is going to require us to use both a partition and a sort order.  
  
Because we want to see values by month, year and month will be our window. We define that using the PARTITION BY clause. We want to rank by sales, so that will be in our ORDER BY clause.

WITH MonthlySalesBySalesperson AS (

SELECT p.FullName,

YEAR(o.OrderDate) AS SalesYear,

MONTH(o.OrderDate) AS SalesMonth,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

JOIN Application.People p

ON o.SalespersonPersonID = p.PersonID

GROUP BY p.FullName, YEAR(o.OrderDate), MONTH(o.OrderDate)

)

SELECT FullName,

SalesYear,

SalesMonth,

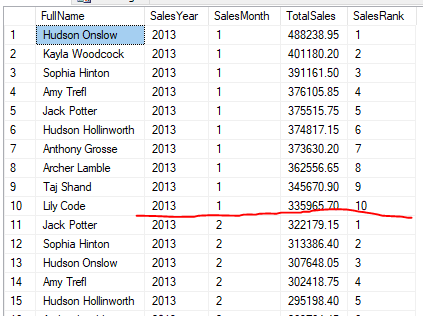
TotalSales,

RANK() OVER ( PARTITION BY SalesYear, SalesMonth ORDER BY TotalSales DESC ) AS SalesRank

FROM MonthlySalesBySalesperson

ORDER BY SalesYear, SalesMonth;

Reviewing the results, notice that there is a new set of rank values for each distinct combination of SalesYear and SalesMonth. This is because SalesYear and SalesMonth defines our window. The rank function is applied against each window.



## Example 2

This time, we want to see how much a salesperson’s sales are increasing or decreasing month over month. To get the previous month, we will use the LAG function. We will limit our search to 2016 sales only to help keep the list smaller.

We want to look by salesperson, that will be our window as defined by the PARTITION BY clause. We will want to order our set by year and month, so we can be sure that we’re looking at the previous month.

WITH MonthlySalesBySalesperson AS (

SELECT p.FullName,

YEAR(o.OrderDate) AS SalesYear,

MONTH(o.OrderDate) AS SalesMonth,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

JOIN Application.People p

ON o.SalespersonPersonID = p.PersonID

GROUP BY p.FullName, YEAR(o.OrderDate), MONTH(o.OrderDate)

)

SELECT FullName,

SalesYear,

SalesMonth,

TotalSales,

LAG(TotalSales) OVER ( PARTITION BY FullName ORDER BY SalesYear, SalesMonth ) AS PreviousMonthSales,

TotalSales - LAG(TotalSales) OVER ( PARTITION BY FullName ORDER BY SalesYear, SalesMonth ) AS Change

FROM MonthlySalesBySalesperson

WHERE SalesYear = 2016

ORDER BY FullName, SalesYear, SalesMonth;

Looking at the results, we can see the LAG resets every time we hit a new salesperson. As expected, the first row in each window returns NULL for the LAG function.

## Example 3

We want to rank our salespeople by monthly sales, like in our first example. On each row, we also want to include the name of the top salesperson and how far behind the sales leader our current salesperson is.

To do this, we can use the FIRST\_VALUE function. Notice that when we use analytic functions, we can pick any column from the row. So, the OVER clause defines which row we should get our value from, but the parameter to FIRST\_VALUE defines the column.  
  
WITH MonthlySalesBySalesperson AS (

SELECT p.FullName,

YEAR(o.OrderDate) AS SalesYear,

MONTH(o.OrderDate) AS SalesMonth,

SUM(ol.Quantity \* ol.UnitPrice) AS TotalSales

FROM Sales.Orders o

JOIN Sales.OrderLines ol

ON o.OrderID = ol.OrderID

JOIN Application.People p

ON o.SalespersonPersonID = p.PersonID

GROUP BY p.FullName, YEAR(o.OrderDate), MONTH(o.OrderDate)

)

SELECT FullName,

SalesYear,

SalesMonth,

TotalSales,

RANK() OVER ( PARTITION BY SalesYear, SalesMonth ORDER BY TotalSales DESC ) AS MonthlyRank,

FIRST\_VALUE(FullName) OVER ( PARTITION BY SalesYear, SalesMonth ORDER BY TotalSales DESC ) AS MonthlyLeader,

TotalSales - FIRST\_VALUE(TotalSales) OVER ( PARTITION BY SalesYear, SalesMonth ORDER BY TotalSales DESC) AS AmountBehindLeader

FROM MonthlySalesBySalesperson

ORDER BY SalesYear, SalesMonth, TotalSales DESC;

# Homework

1. On the Sales.OrderLines table, write a query that will get OrderLine, OrderID, Description, the total line price (i.e. quantity \* unit price) and the total value of that order (i.e. partition by order).
2. Modify the query created above to display the percentage of the total order that this line represents (i.e. this line’s total price / the order’s total value)
3. Using Sales.OrderLines, for each order, rank the order lines by their total value.
4. Using the MonthlySalesBySalesperson CTE available to you in the lesson, create the following queries:
   1. Create a query that will rank the year and month for each salesperson by sales. So, the resulting query show the years and months ranked from best to worst for each salesperson.
   2. Create a query that will rank salespeople by monthly sales, showing the name and sales of the next highest person on the list.
   3. Create a query that will rank salespeople by monthly sales, but also include the average sales for that month. Show how far above or below the average each salesperson is.
   4. Create query that will rank salespeople by monthly sales. Calculate what percentage of all sales that month were made by each salesperson. (i.e. sales for this salesperson / total sales for the month).