**How to Take Your SQL from Zero to Data Scientist Level — Part 2/3**

**The ultimate guide for your SQL List: Learn the basic and advanced SQL queries that data scientists frequently use**

**Intro**

Are you looking to learn SQL at the standards of a good data science team? If so, you are in the right place! In [**Part 1**](https://medium.com/@alex.vamvakaris.ds/how-to-take-your-sql-from-zero-to-data-scientist-level-part-1-3-d7225d2d89ad), we set up SQL and executed a simple SELECT query on the AdventureWorks2019 demo database (with a touch of theory on SQL and relational databases). In this part of the series, we will dive deeper into SQL, starting from a simple two-line SELECT statement and working our way up to much more complicated queries.

**Part 2**

* Extract data with the SELECT statement
* How to JOIN different tables in the database
* Advanced SQL statements: the CASE expression, dealing with dates, CTEs, subqueries, and ranking functions

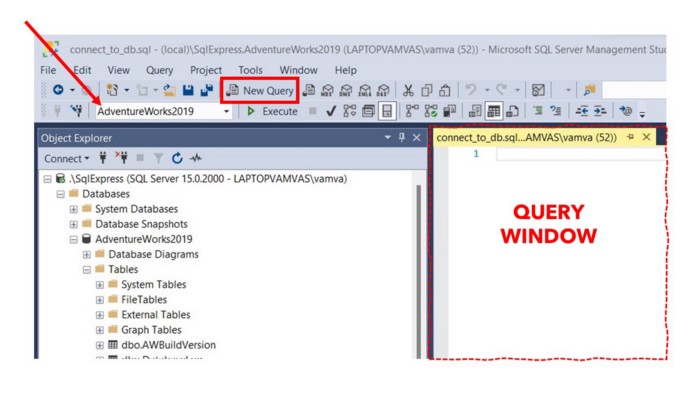
**Coming Soon**

* Part 3: Query optimization, testing your code, data science coding best practices, and plan for training

**Before We Start**

*\*These steps are not mandatory to follow the guide, but they will allow you to execute the queries and practice SQL on your own*

* Open SSMS and connect to a SQL server
* Select “New Query”from the toolbar, and the query window will open on the right panel
* Finally, change the database from master to AdventureWorks2019 to connect the query window to your demo database, and you are set to go



Connect to a database in SSMS and open a new query window [Image by the author]

*👉 If you need help with any of the above, there is a step-by-step guide in* [***Part 1***](https://medium.com/@alex.vamvakaris.ds/how-to-take-your-sql-from-zero-to-data-scientist-level-part-1-3-d7225d2d89ad)

**1. In the Land of SQL, the SELECT Statement is King**

As a data scientist, your primary purpose for using SQL will be to extract data from the database tables. For every such query, the SELECT statement will be used. The syntax for the SELECT statement is made up of the following six clauses, each aiming at refining the query so that only the required data is returned.

**SELECT** ( *column list )*

**FROM** ( *table reference )*

**WHERE** ( *filtering condition )*

**GROUP BY** ( *grouping specification )*

**HAVING** ( post-grouping *filtering condition )*

**ORDER BY** ( *order condition )*

**1.1 Order of Evaluation**

When you execute a SELECT statement, each clause is evaluated in a specific order. This is different from the order that SQL will be written (syntax as shown above). The order of evaluation is what happens under the hood. You can think of it as a virtual table, changing as we progress from one clause to the next until the result is returned. A good understanding of the evaluation order will not only help you write more efficient queries but will also help you progress faster as a beginner.

1. The **FROM** clause is mandatory and is executed first. It is used to select the table in the database from which we want to extract the data
2. The results are then used in the **WHERE** clause (if one is specified) to filter the rows of the table based on some condition
3. Then the results are used in the **GROUP BY** clause (if one is specified) to group the table
4. Then the grouped resultsare used in the **HAVING** clause (if one is specified — it also requires GROUP BY) to filter the grouped rows
5. After the last of the above clauses is executed, the results are passed on to the (mandatory) **SELECT** clause, where columns are selected
6. In the end, the **ORDER BY** clause is evaluated (if one is specified) to order the columns in the results

It is also worth mentioning that two of the most common questions in SQL interviews are to explain the difference between GROUP BY and HAVING (more on that later) and to walk through the order of evaluation of a SELECT statement.

**1.2 SELECT & FROM**

Let’s start at the simplest form of the SELECT statement by using only the two mandatory clauses SELECT and FROM:

* The table reference in the FROM clause takes the form of [database].[schema].[table]. We can omit the [database] part as we will only use the AdventureWorks2019 database in the examples and we are already connected (you can think of schemas as containers for “similar” tables)
* The DISTINCT and TOP keywords are expensive (memory-wise), so be careful to use them selectively

---=================================  
--- Select all columns and rows  
---=================================  
SELECT \*  
FROM [AdventureWorks2019].[Sales].[SalesOrderHeader]  
  
---=================================  
--- Return the top 10 rows and all columns  
---=================================  
SELECT TOP 10 \*  
FROM [Sales].[SalesOrderHeader]  
  
---=================================  
--- Select all rows for specified columns  
---=================================  
SELECT  
 [SalesOrderID],  
 [OrderDate],  
 [TotalDue]  
FROM [Sales].[SalesOrderHeader]  
  
---=================================  
--- Select distinct rows for specified columns  
---=================================  
SELECT DISTINCT  
 [SalesOrderID],  
 [OrderDate],  
 [TotalDue]  
FROM [Sales].[SalesOrderHeader]

**Calculated Columns**

You can **create calculated columns** by using arithmetic operators or functions. We can use the **AS** keyword to name a calculated column or rename the existing columns in the results.

---=================================  
--- Create calculated columns  
---=================================  
SELECT TOP 10  
 [SalesOrderID],  
 [TotalDue],  
 [TotalDue] AS Total\_Amount\_Due,  
 [TotalDue] \* 0.8 AS Gross\_Due,  
 ROUND([TotalDue] \* 0.8 , 1) AS Gross\_Due\_Round  
FROM [Sales].[SalesOrderHeader]

**The NULL Value**

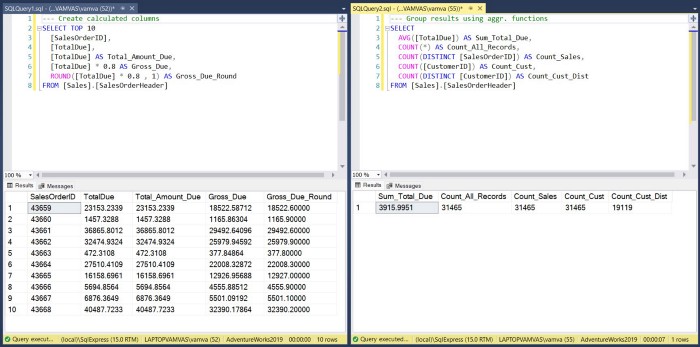
A key concept in SQL is the NULL value. The NULL value is used to mark absent data (no entry was made to that cell). This is different from a cell containing spaces or a zero value. Think of them as the NA of SQL.

**Aggregations**

You can also create aggregate results using functions as in the examples below:

* **COUNT(\*):** returns the count of all rows in the table
* **COUNT(*column*):** returns the count of all non-NULL records for the column
* **COUNT(DISTINCT *column*):** returns the count of distinct values for the column. NULL will also be counted as one of the values
* **COUNT(\*) = COUNT(*SalesOrderID*) =** **COUNT(DISTINCT *SalesOrderID*):** the SalesOrderID column is the primary key for the table and so is unique by row
* **COUNT(DISTINCT *CustomerID*)** < **COUNT(*CustomerID*):** the CustomerID column has duplicate values (a customer can be in more than one order)
* You can find the full list of aggregate functions for SQL Server [**here**](https://learn.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql?view=sql-server-ver16)

---=================================  
--- Group results using aggr. functions  
---=================================  
SELECT  
 AVG([TotalDue]) AS Sum\_Total\_Due,  
 COUNT(\*) AS Count\_All\_Records,  
 COUNT(DISTINCT [SalesOrderID]) AS Count\_Sales,  
 COUNT([CustomerID]) AS Count\_Cust,  
 COUNT(DISTINCT [CustomerID]) AS Count\_Cust\_Dist  
FROM [Sales].[SalesOrderHeader]



Results in SSMS from example SELECT queries [Image by the author]

I recommend you practice using COUNT and other aggregate functions for different columns (with and without NULL) to better understand how SQL behaves. It is also worth mentioning that the syntax in SQL isn't case sensitive. So we can write SELECT or SeLeCt, COUNT, or coUnt, and the query will still be executed. We will use capital when needed to improve code readability (we will cover coding best practices in Part 3 of the series).

**1.3 WHERE**

The WHERE clause takes the data selected by the FROM clause and evaluates each row based on some filtering condition:

* Those rows that evaluate to TRUE are returned as part of the query result
* Those that evaluate as UNKNOWN or FALSE are not included in the results (UNKNOWN is returned for NULL values)

In order to evaluate the filtering conditions, we will be using **Predicates**. There are many, so let’s take them one by one.

**Comparison Predicates**

These are the usual symbols you have been using in math for comparisons.

---=================================  
--- Equal   
---=================================  
SELECT \*  
FROM [Sales].[SalesOrderHeader]  
WHERE [TerritoryID] = 6  
  
---=================================  
--- Not Equal  
---=================================  
SELECT  
 [SalesOrderID],  
 [TerritoryID]  
FROM [Sales].[SalesOrderHeader]  
WHERE [TerritoryID] <> 6  
  
---=================================  
--- Greater or equal  
---=================================  
SELECT  
 [SalesOrderID],  
 [TerritoryID]  
FROM [Sales].[SalesOrderHeader]  
WHERE [TerritoryID] >= 6

**BETWEEN Predicate**

The BETWEEN predicate is used in conjunction with the AND keyword to identify a range of values that can be included as a search condition.

---=================================  
--- Filter by numerical range  
---=================================  
SELECT  
 [SalesOrderID],  
 [TerritoryID]  
FROM [Sales].[SalesOrderHeader]  
WHERE [TerritoryID] BETWEEN 2 AND 6  
  
---=================================  
--- Filter by date range  
---=================================  
SELECT COUNT(\*)  
FROM [Sales].[SalesOrderHeader]  
WHERE [OrderDate] BETWEEN '2013-06-01' AND '2013-06-30'

**IN Predicate**

Determines whether a column value matches any of the values in a list.

---=================================  
--- Filter by values in a list  
---=================================  
SELECT  
 [SalesOrderID],  
 [TerritoryID]  
FROM [Sales].[SalesOrderHeader]  
WHERE [TerritoryID] IN (2, 3, 6)

**LIKE Predicate**

If any predicate can be fun, it is the LIKE predicate. With the LIKE predicate, you can specify values that are only similar to the values stored in the database (the wild character %, used in the examples below, differs between vendors).

---=================================  
--- Values that start with 43  
---=================================  
SELECT TOP 5  
 [SalesOrderID]  
FROM [Sales].[SalesOrderHeader]  
WHERE [SalesOrderID] LIKE '43%'  
  
---=================================  
--- Values that contain 43  
---=================================  
SELECT   
 COUNT(DISTINCT [CustomerID])  
FROM [Sales].[SalesOrderHeader]  
WHERE [SalesOrderID] LIKE '%43%'

**IS NULL**

As we covered before, using the normal predicates, a NULL value will always be evaluated as unknown, and so nothing will be returned. So we need another predicate that will return either true or false. For this reason, SQL provides the IS NULL condition.

---=================================  
--- Select only values with NULL  
---=================================  
SELECT   
 [SalesOrderID],  
 [CurrencyRateID]  
FROM [Sales].[SalesOrderHeader]  
WHERE [CurrencyRateID] IS NULL

**Multiple Comparisons**

If more than one search condition is present, they are combined using logical operators (OR, AND, NOT). The rule in terms of execution precedence is that Parenthesis > NOT > AND > OR. Be mindful to spot-check the results for multiple comparisons (in the example below, version 1 will give different results to version 2 just by changing the parenthesis).

---=================================  
--- Version 1  
---=================================  
SELECT \*  
FROM [Sales].[SalesOrderHeader]  
WHERE   
[totaldue] < 3000 AND ([TerritoryID] < 2 OR [TerritoryID] > 7)   
  
---=================================  
--- Version 2  
---=================================  
SELECT \*  
FROM [Sales].[SalesOrderHeader]  
WHERE   
([totaldue] < 3000 AND [TerritoryID] < 2) OR [TerritoryID] > 7

**1.4 GROUP BY**

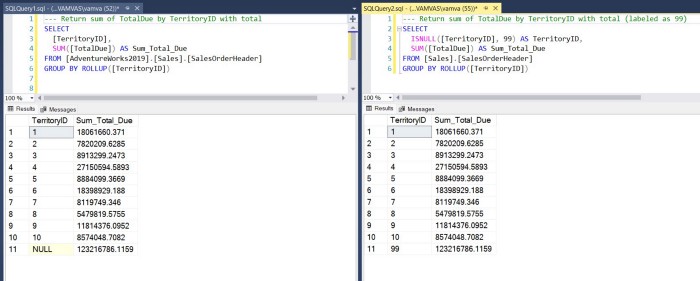
The GROUP BY clause is used to group the data (passed on from FROM or WHERE) in the same way you use pivot tables in excel or the group\_by() in dplyr in R. Using the GROUP BY without aggregates will return the distinct combination of values for the selected columns (same result as using DISTINCT).

---=================================  
--- Return distinct values in TerritoryID  
---=================================  
SELECT  
 [TerritoryID]  
FROM [AdventureWorks2019].[Sales].[SalesOrderHeader]  
GROUP BY [TerritoryID]  
  
---=================================  
--- Return sum of TotalDue by TerritoryID  
---=================================  
SELECT  
 [TerritoryID],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [AdventureWorks2019].[Sales].[SalesOrderHeader]  
GROUP BY [TerritoryID]

**Totals and Subtotals**

We can also get totals or subtotals using the **ROLLUP** operator (expensive memory-wise). As you can see from the example below (left panel), the total is represented as NULL. We can fix that by replacing it with a new value using the **ISNULL** function.

---=================================  
--- Return sum of TotalDue by TerritoryID with total  
---=================================  
SELECT  
 [TerritoryID],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [AdventureWorks2019].[Sales].[SalesOrderHeader]  
GROUP BY ROLLUP([TerritoryID])  
  
---=================================  
--- Return sum of TotalDue by TerritoryID with total (labeled as 99)  
---=================================  
SELECT  
 ISNULL([TerritoryID], 99) AS TerritoryID,  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [Sales].[SalesOrderHeader]  
GROUP BY ROLLUP([TerritoryID])



Results in SSMS from using GROUP BY with ROLLUP [Image by the author]

**1.5 HAVING**

The HAVING clause is exactly the same as the WHERE but can only be applied after the GROUP BY. In other words, WHERE is used to filter the result of FROM, whereas HAVING is used to filter the result of GROUP BY. Remember the analogy of a virtual table passing from one clause to the next.

---=================================  
--- Filter using column  
---=================================  
SELECT  
 [TerritoryID],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [Sales].[SalesOrderHeader]  
GROUP BY [TerritoryID]  
HAVING [TerritoryID] = 1  
  
---=================================  
--- Filter using aggregate  
---=================================  
SELECT  
 [TerritoryID],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [Sales].[SalesOrderHeader]  
GROUP BY [TerritoryID]  
HAVING SUM([TotalDue]) >= 15000000

**1.6 ORDER BY**

Okay, one last clause left, and luckily it is the simplest. We can use the ORDER by clause to order the results from SELECT in an ascending (default in most vendors) or descending order.

---=================================  
--- Sort by ascending order  
---=================================  
SELECT  
 [TerritoryID],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [Sales].[SalesOrderHeader]  
GROUP BY [TerritoryID]  
ORDER BY SUM([TotalDue])  
  
---=================================  
--- Combine all clauses and sort by descending order  
---=================================  
SELECT  
 [TerritoryID],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [Sales].[SalesOrderHeader]  
WHERE [TerritoryID] >= 3  
GROUP BY [TerritoryID]  
HAVING SUM([TotalDue]) >= 15000000  
ORDER BY [TerritoryID] DESC

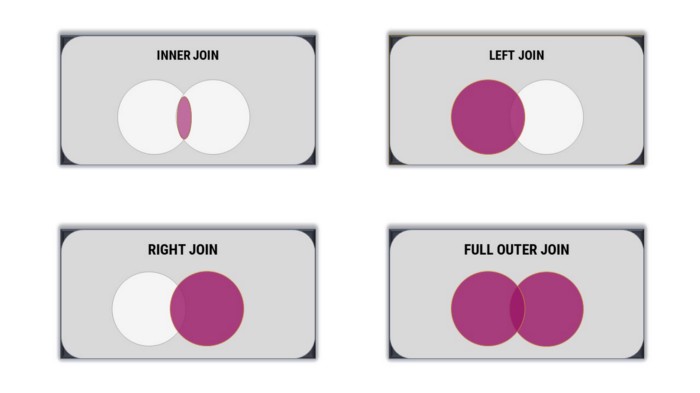
**2. How to JOIN Different Tables**



Photo by [Matt Koffel](https://unsplash.com/@mattkoffel_?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/bridge?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

More often than not, you will need to retrieve information from multiple tables in your queries. There are four main types of JOIN clauses (although, in reality, only the INNER and LEFT JOIN are frequently used):

* **INNER JOIN:** returns only the records that exist in both tables
* **LEFT JOIN:** returns only the records that exist in the left table
* **RIGHT JOIN:** returns only the records that exist in the right table
* **FULL OUTER JOIN:** returns all the records from both tables



Visualizing JOINS using Venn diagrams [Image by the author]

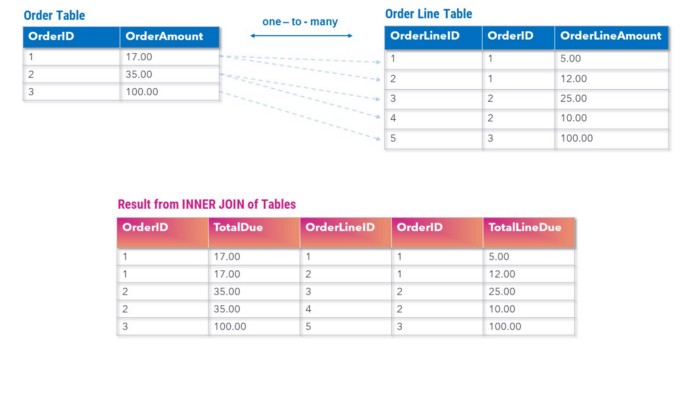
Okay, so the type of join determines which records (rows) are retained from the tables. We can then use one or more shared columns to link the tables together:

* Select the type of join
* Give aliases to the tables (S and T in our example)
* Specify the shared column to be used for joining the tables using the ON keyword

---=================================  
--- Inner join the sales and territory tables  
---=================================  
SELECT  
 S.[TerritoryID],  
 T.[Name]  
FROM [Sales].[SalesOrderHeader] S  
INNER JOIN [Sales].[SalesTerritory] T   
 ON S.TerritoryID = T.TerritoryID  
  
---=================================  
--- Left join the sales and customer tables (more complex)  
---=================================  
SELECT  
 S.[TerritoryID],  
 T.[Name],  
 SUM([TotalDue]) AS Sum\_Total\_Due  
FROM [Sales].[SalesOrderHeader] S  
LEFT JOIN [Sales].[SalesTerritory] T  
 ON S.TerritoryID = T.TerritoryID  
GROUP BY   
 S.[TerritoryID],  
 T.[Name]  
HAVING T.[Name] IN ('Australia','United Kingdom')

**Dealing with Duplicates**

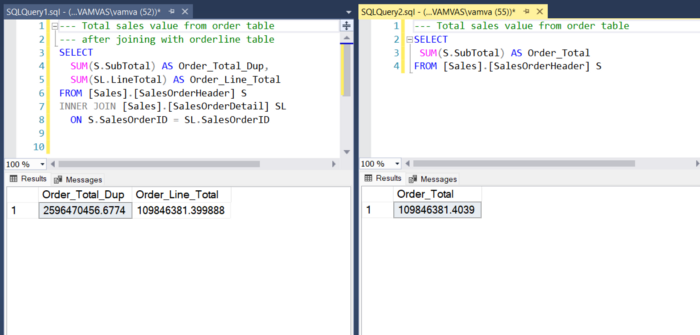
There is one major consideration when joining tables. When the relationship between two tables is one-to-many, the joined result will have duplicates.



Joining tables with one-to-many relationship [Image by the author]

This can be challenging and lead to errors, especially when aggregations are involved. I suggest consistently verifying the level that the column is at. In the example above, the OrderAmount value is unique at the Order level (OrderID), but the joined table will have duplicates and should not be used in aggregates (we will cover duplicates and how to QA test your code in Part 3).

---=================================  
--- Total sales value from order table  
---=================================  
SELECT  
 SUM(S.SubTotal) AS Order\_Total --- correct  
FROM [Sales].[SalesOrderHeader] S  
  
---=================================  
--- Total sales value from order table  
--- after joining with orderline table  
---=================================  
SELECT  
 SUM(S.SubTotal) AS Order\_Total\_Dup, --- duplicates  
 SUM(SL.LineTotal) AS Order\_Line\_Total --- correct  
FROM [Sales].[SalesOrderHeader] S  
INNER JOIN [Sales].[SalesOrderDetail] SL  
 ON S.SalesOrderID = SL.SalesOrderID



Duplicates example in a one-to-many relationship in SSMS [Image by the author]

**3. Advanced SQL Queries (My Heroes)**



Photo by [Erik Mclean](https://unsplash.com/@introspectivedsgn?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/COMIC-BOOKS?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

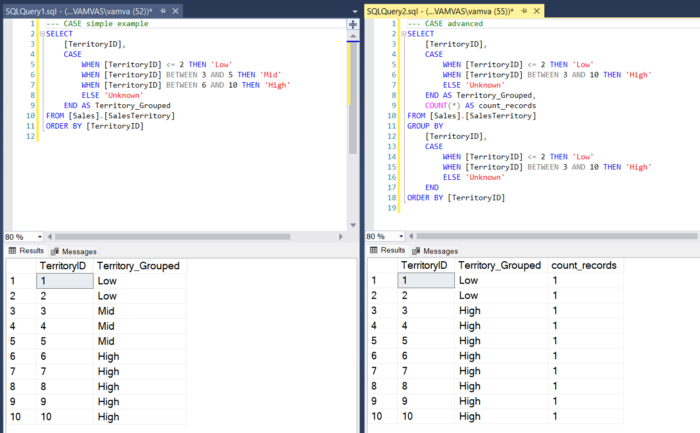
To wrap up, I also wanted to show you some more advanced tools you will need as a data scientist. In the past six years, I have coded over 1,500 hours in SQL, and these tools were my heroes!

**3.1 CASE**

The CASE statement allows you to create calculated fields based on multiple conditions:

* The syntax starts with the keyword **CASE** followed by a series of **WHEN** clauses and ends with the **END** keyword
* Each **WHEN** clause consists of a condition and a result if that condition evaluates to TRUE (clauses are evaluated in the order in which they appear)
* If none of the conditions are met, the **ELSE** clause will be executed (if not used, unmatched values will be NULL)
* The **ELSE** clause is optional, but I highly recommend it to ensure your code works as intended

---=================================  
--- CASE simple example  
---=================================  
SELECT  
 [TerritoryID],  
 CASE   
 WHEN [TerritoryID] <= 2 THEN 'Low'  
 WHEN [TerritoryID] BETWEEN 3 AND 5 THEN 'Mid'  
 WHEN [TerritoryID] BETWEEN 6 AND 10 THEN 'High'  
 ELSE 'Unknown'  
 END AS Territory\_Grouped  
FROM [Sales].[SalesTerritory]  
ORDER BY [TerritoryID]  
  
---=================================  
--- CASE advanced  
---=================================  
SELECT  
 [TerritoryID],  
 CASE   
 WHEN [TerritoryID] <= 2 THEN 'Low'  
 WHEN [TerritoryID] BETWEEN 3 AND 10 THEN 'High'  
 ELSE 'Unknown'  
 END AS Territory\_Grouped,  
 COUNT(\*) AS count\_records  
FROM [Sales].[SalesTerritory]  
GROUP BY  
 [TerritoryID],  
 CASE   
 WHEN [TerritoryID] <= 2 THEN 'Low'  
 WHEN [TerritoryID] BETWEEN 3 AND 10 THEN 'High'  
 ELSE 'Unknown'  
 END  
ORDER BY [TerritoryID]



Using the CASE statement [Image by the author]

**3.2 Dates**

As a data scientist, you will need to deal with dates in your data quite often. That will usually involve using the below two functions.

**DATEPART()**

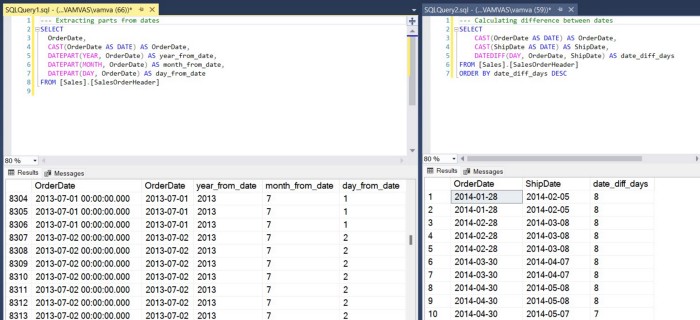
Used to extract a specific part of a date such as the year, week, hour, etc. It takes two arguments as input; the unit of measure used to extract the part of the date and the date itself. We can also use the **CAST** function to change the data type of the date columns (in our example below, we changed it from datetime to date).

---=================================  
--- Extracting parts from dates  
---=================================  
SELECT  
 CAST(OrderDate AS DATE) AS OrderDate,  
 DATEPART(YEAR, OrderDate) AS year\_from\_date,  
 DATEPART(QUARTER, OrderDate) AS quarter\_from\_date,  
 DATEPART(MONTH, OrderDate) AS month\_from\_date  
FROM [Sales].[SalesOrderHeader]

**DATEDIFF()**

Used to calculate the difference between two dates. It takes three arguments as input; the unit of measure, the start date, and the end date (from left to right).

---=================================  
--- Calculating difference between dates  
---=================================  
SELECT  
 CAST(OrderDate AS DATE) AS OrderDate,  
 CAST(ShipDate AS DATE) AS ShipDate,  
 DATEDIFF(DAY, OrderDate, ShipDate) AS date\_diff\_days  
FROM [Sales].[SalesOrderHeader]  
ORDER BY date\_diff\_days DESC



Working with dates in SSMS [Image by the author]

**3.3 CTE and Subqueries**

You will find subqueries in most textbooks teaching SQL, but this is not why I added them in this section. My reason is that I wanted to showcase their disadvantages and how you will be much better off using CTEs (Common Table Expressions).

**Subqueries**

A subquery is a SELECT statement that is nested within another SELECT statement. They can also be used in the FROM clause of a SELECT statement. Either way, the result from the subquery will be used in the main SELECT statement. Let’s see some examples of how that works.

---=================================  
--- Using nested subqueries  
---=================================  
SELECT   
 COUNT([CustomerID])  
FROM [Sales].[SalesOrderHeader]  
WHERE [CustomerID] IN ( --- nested subquery starts here  
 SELECT [CustomerID]  
 FROM [Sales].[Customer]  
 WHERE [TerritoryID] = 4  
)  
  
---=================================  
--- Using subqueries in the FROM  
---=================================  
SELECT   
 COUNT(a.[CustomerID]) AS Count\_Cust  
FROM ( --- subquery starts here  
 SELECT \*  
 FROM [Sales].[SalesOrderHeader]  
 WHERE [TerritoryID] = 4  
) a -- you need to add an alias at the end

**CTEs**

Although CTEs have the same functionality as subqueries (create a temporary result that can then be used in the main SELECT), they also have three main advantages:

* Reusable in different parts of the query
* Easy to validate and understand (more readable)
* CTEs can often be more efficient than subqueries

---======================  
--- Using a single CTE  
---======================  
WITH   
select\_customers AS (  
 SELECT \*  
 FROM [Sales].[SalesOrderHeader]  
 WHERE [TerritoryID] = 4  
)  
SELECT   
 COUNT([CustomerID])  
FROM select\_customers  
  
---======================  
--- Using multiple CTEs  
---======================  
WITH   
select\_customers AS (  
 SELECT   
 [CustomerID]  
 FROM [Sales].[Customer]  
 WHERE [TerritoryID] = 4  
),  
select\_orders AS (  
 SELECT   
 CAST(OrderDate AS DATE) AS OrderDate,  
 [SalesOrderID],  
 [CustomerID],  
 [SubTotal]  
 FROM [Sales].[SalesOrderHeader]  
 WHERE [SubTotal] >= 2000  
)  
SELECT  
 c.[CustomerID],  
 SUM([SubTotal]) AS Sum\_Total\_Due  
FROM select\_customers c  
INNER JOIN select\_orders o  
 ON c.[CustomerID] = o.[CustomerID]  
GROUP BY c.[CustomerID]  
ORDER BY Sum\_Total\_Due

When deciding whether to use a CTE or a subquery, it’s important to consider the complexity of the query, the performance requirements, and the readability of the query. In general, CTEs are a good choice for complex queries that are used multiple times, while subqueries are a good choice for simpler queries that are used only once.

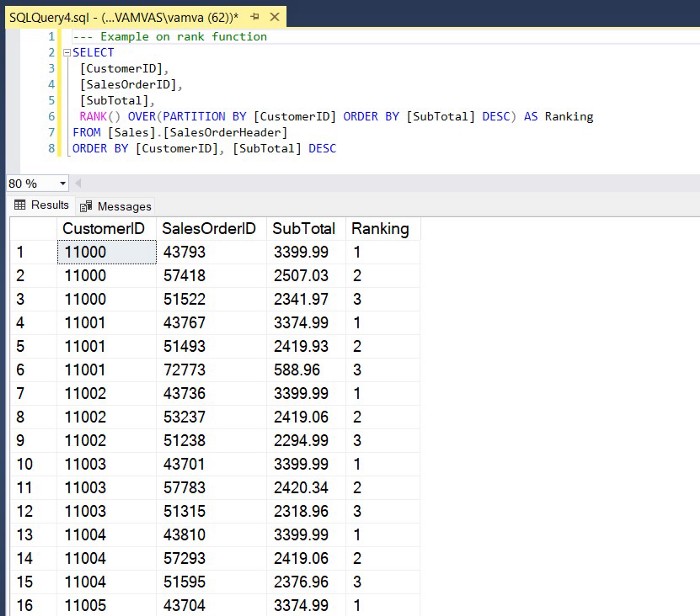
**3.4 Ranking Functions**

Ranking (window) functions are the most complicated part of the article. However, you will need to use them in your data science queries, and on a side note, these are often asked in interviews.

As the name suggests, they are used to rank rows, but each does it slightly differently. But I am getting ahead of myself. Let’s first check the syntax with the example below:

* We will use **RANK()** as the ranking function in the SalesOrderHeader table
* We want to partition the data so that RANK() does not apply to the whole dataset but to a subset of rows each time. In our case, we used **PARTITION BY** CustomerID, so the ranking was applied to each customer separately
* Finally, we want to rank orders for each customer based on the SubTotal column (i.e., the value of the order) and start from 1 for the highest spend and so on. This is where the familiar **ORDER BY** came into play

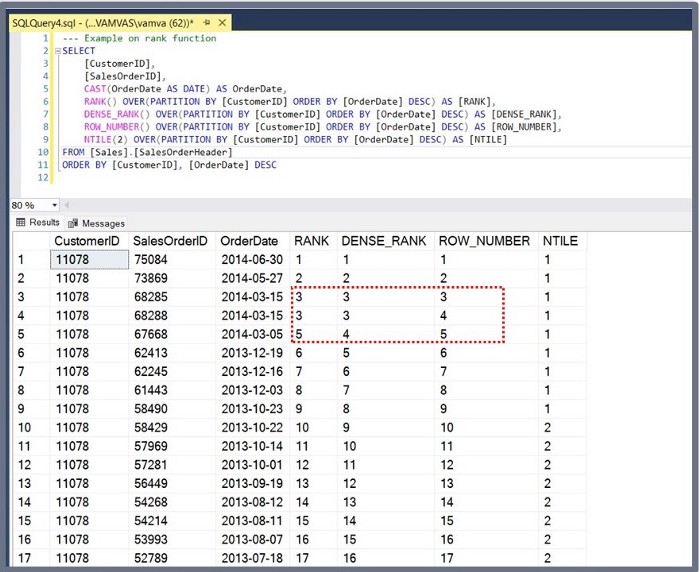
---=================================  
--- Using the rank function  
---=================================  
SELECT   
 [CustomerID],  
 [SalesOrderID],  
 [SubTotal],  
 RANK() OVER(PARTITION BY [CustomerID] ORDER BY [SubTotal] DESC) AS Ranking  
FROM [Sales].[SalesOrderHeader]  
ORDER BY [CustomerID], [SubTotal] DESC



Example of how RANK() works in SQL [Image by the author]

You will frequently use four ranking functions in SQL as a data scientist. Let’s see the differences between them using data from the **SalesOrderHeader** table.

---=================================  
--- Example using all ranking functions  
---=================================  
SELECT   
 [CustomerID],  
 [SalesOrderID],  
 CAST(OrderDate AS DATE) AS OrderDate,  
 RANK() OVER(PARTITION BY [CustomerID] ORDER BY [OrderDate] DESC) AS [RΑΝΚ],  
 DENSE\_RANK() OVER(PARTITION BY [CustomerID] ORDER BY [OrderDate] DESC) AS [DENSE\_RANK],  
 ROW\_NUMBER() OVER(PARTITION BY [CustomerID] ORDER BY [OrderDate] DESC) AS [ROW\_NUMBER],  
 NTILE(2) OVER(PARTITION BY [CustomerID] ORDER BY [OrderDate] DESC) AS [NTILE]  
FROM [Sales].[SalesOrderHeader]  
WHERE [CustomerID] = 11078  
ORDER BY [CustomerID], [OrderDate] DESC



Example of how ranking functions differ [Image by the author]

* **ROW\_NUMBER()**: Assigns a unique sequential number to each row within a result set
* **RANK():** Returns the rank of each row within a result set partition. In the case of ties, the rank of a specific row is one plus the number of rank values that come before it
* **DENSE\_RANK()**: Returns the rank of each row within a result set partition. In the case of ties, the rank of a specific row is one plus the number of distinct rank values that come before it
* **NTILE(n):** Divides the rows in a result set into n equal groups (or tiles) and assigns a rank to each row based on the group it belongs to. This function can be used to calculate the median and other percentiles

And with that, we come to an end. We have covered quite a bit of ground, so I believe **congratulations** are in order! 🚀🚀🚀

**I also included some practice exercises to help you hone your SQL skills.**

1. Find the total amount due (SalesOrderHeader.TotalDue), grouped by month and year (SalesOrderHeader.OrderDate). Then filter the rows to only keep January to March. Finally, rename the columns and order the results by year and month (descending)
2. Find the average amount (SalesOrderDetail**.**LineTotal), grouped by year (SalesOrderHeader.OrderDate) and product name (Product.Name)
3. Try exercise 2, using CTEs
4. From exercise 2, use ROW\_NUMBER() to keep only the top 5 highest performing products (LineTotal aggregate) in each month. Use CTEs
5. Use the CASE statement to create a new calculated field that splits the total amount due (SalesOrderHeader.TotalDue) into five buckets (low, medium, etc.), and then for each bucket, find the average freight cost (SalesOrderHeader.Freight)

I hope you found this article helpful and have fun querying!

**Stay Tuned for Part 3!**

If you enjoyed reading this article and want to learn more, don’t forget to [**subscribe**](https://medium.com/@alex.vamvakaris.ds/subscribe) to get my stories sent directly to your inbox.

On the link below, you can also find a free PDF Walkthrough on completing a Customer Cluster Analysis in a real-life business scenario using data science techniques and best practices in R.