**Prerequisite: SQL Server Quick Guides 1-3 provided in weeks 1 learning content on Blackboard.**

# TSQL Cheat Sheet - All TSQL Demo code for AdventureWorks2019

Once you have covered the above Guides you can now proceed to covering the content of Microsoft **TSQL Modules 01-09** by simply running the **TSQL Demos** provided in this **TSQL Cheat Sheet - All TSQL Demo code for AdventureWorks2019**.

Whenever you are learning new Development Technologies it is good to prepare your own ‘cheat sheets’ as a valuable resource to help you freshen up your skills when needed.

# How to learn TSQL with MS AdventureWorks2019

You can learn a lot about SQL statements quickly just by running them and analyzing the results returned from the database. Interactive learning is the way to go. To run these queries, open SQL Server Management Studio (SSMS) and log into the localhost (covered in SQL Server Quick Guide 1).

Find the **New Query** button on the toolbar and one at a time, copy and paste the following T-SQL query syntax, then press the crimson **Execute button** to run the query and retrieve data. Review the results!

***At this point you may consider amending the queries to create your own examples.***

# TSQL Demo’s with MS AdventureWorks2019

The TSQL Demos produced by Module Leader work with [AdventureWorks2019](https://learn.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms) and the sample are designed to help you learn SQL Server and TSQL programming code (also referred to as Scripts).

**You have adequate access to TU Labs to support you when learning ‘Development Technology’ stacks. The onus is on you to utlise the resources provided.**

**We do try to support you with instructions on how to setup the development software on your PC and/or Laptops.**

**Remote SQL Study without a decent home machine**

We cannot do much if you do not have adequate home machines but we do recommend you consider utilising [***W3School – SQL Resource.***](https://www.w3schools.com/sql/default.asp)

[***W3School – SQL:***](https://www.w3schools.com/sql/default.asp)*provide an online SQL editor that allows you to practice your code using MS Adventurework Databases sample. Many historic Developers do share their online content and explains what is happening*[***(here’s an example)***](http://www.w3schools.com/sql/default.asp)*and do feel free to utilise* [***W3School***](https://www.w3schools.com/sql/default.asp)*resource.*

*- Do note W3School uses an older historic AdventureWorks from 2005 and is provided within our samples.*

*- In practice the 2019 TSQL cheat code examples should work with W3School – 2005 version but you may need to check the ‘date’ fields on records to retrieve data from the older version.*

*Do note you are not allowed to use any of the teaching databases for your ICA.*

*Do not use* ***TSQL, Adventureworks version, Northwind or World Wide Importers***

***Use the TU resources if you do not have adequate home setup. You are given space on your timetable for self-study on campus.***

# AdventureWorks2019 Sample Database Overview: Bottom of Form

Microsoft’s and the SQL Developer community have been learning TSQL coding with AdventureWorks since 2005. There are many resources, code samples and demos provided by the developer community online. Have a quick look over YouTube resource:

Youtube: [Adventureworks 2019 Database in SSMS](https://www.youtube.com/results?search_query=Adventureworks+2019+Database)

Supporting Youtube Videos:

Do remember the SQL Server Quick Guides [1-3] cover the content in the following videos:

[how to restore adventureworks2019.bak](https://www.youtube.com/watch?v=D_PZOhFxGQU)

[adventureworks2019 sql tutorials for you to follow along](https://www.youtube.com/results?search_query=adventureworks2019+sql+tutorial+)

[Create a Database Diagram for Adventureworks 2019 Database in SSMS](https://www.youtube.com/watch?v=zxUNjnxquMI)

|  |  |  |
| --- | --- | --- |
| TSQL Cheat Sheet - All TSQL Demo code for AdventureWorks2019 | | |
| The TSQL Cheat Sheet Demo Code using AdventureWorks2019 Database Sample.  **Run the sections of TSQL code in SSMS - Query Window.**  The TSQL Cheat Sheet Code covers examples for the following TSQL Modules:   * Querying Tables with Select (TSQL03) * Querying Multiple Tables (TSQL04 and 09) * Sort and Filter Data (TSQL05 and 09) * Working with SQL Server Data (TSQL06) * Using DML to modify Data (TSQL07) * Using Built in Functions (TSQL08 and 09) * Grouping and Aggregating Data (TSQL09) | | |
| **1** | USE AdventureWorks2019;  SELECT CustomerID, SalesOrderID, [SubTotal], [TotalDue]  FROM Sales.SalesOrderHeader  WHERE CustomerID = 29825  -------------------------------------------------------- USE AdventureWorks2019;  SELECT \* FROM Sales.SalesOrderHeader  WHERE [OrderDate] BETWEEN '1/1/2011' AND '12/31/2011'  -----------------------------------------------------  USE AdventureWorks2019;  SELECT \* FROM [Purchasing].[PurchaseOrderHeader]  WHERE [OrderDate] BETWEEN '1/1/2011' AND '12/31/2011' AND [VendorID] = 1492 | Open up the AdventureWorks2019 database and retrieve either the sales order information for customer #1 (first query) or in the 2nd query one month's of data records from the SalesOrderHeader table.  The **USE** statement tells SSMS what database to operate on. The **\*** means to retrieve all the columns. The **FROM** specifies what table to retrieve the data from. The month of data retrieved is in the **WHERE** statement. You can add many different filters to the WHERE statement, which is pretty important because you rarely want ALL the data rows from a table. You should see 174 rows.  For specifying filters in the where clause dates and words have to be in 'single quotes' and numbers do not. You have to know the datatype of the column that you are filtering on first, before you craft your WHERE statement. You can do that by using SSMS's Object explorer to find the database and table then right-clicking the table name and columns to see the column names and their datatypes.  The BETWEEN statement is very helpful! What to see why? You can delete the WHERE clause and run the query again to see why. |
| **2** | USE AdventureWorks2019;  SELECT [SalesOrderID]  ,[CustomerID]  , [TerritoryID]  , STR([SubTotal]) AS [Sub Total], [TaxAmt]  , + '$' + CONVERT(varchar(12), [TotalDue], 1) AS [Total $ Due]  FROM Sales.SalesOrderHeader  WHERE [OrderDate] BETWEEN '1/1/2011' AND '12/31/2011'  -- a lot of SQL developers write their queries like this. | Open up the AdventureWorks table and retrieve one month's of data records from the SalesOrderHeader table. **Retrieve just the columns that are needed** for analysis. Notice the commas after each column EXCEPT THE LAST.  Normally you would copy the data into excel for formatting and report submission, or copy the SQL query into SQL Server reporting services when making a dataset. But if you wanted to report out quickly some query results you can use the STR(field name) function shown in the Sub total field above which strips out the decimal places, or the CONVERT(field name) function to add the comma and 2 decimal places formatting.  You can also see that a $ sign was appended to the field. This goofiness was replaced by the FORMAT(field name function in SQL Serve 2012). |
| **3** | USE AdventureWorks2019;  SELECT T.[Name], [SalesOrderID],[CustomerID],  S.[TerritoryID],[SubTotal],[TaxAmt],[TotalDue]  FROM Sales.SalesOrderHeader AS S  INNER JOIN [Sales].[SalesTerritory] AS T  ON T.TerritoryID =S.TerritoryID  WHERE [OrderDate] BETWEEN '1/1/2011' AND '12/31/2011' | This query is the same as the prior except that the data is made much more understandable by the inclusion of the English name of the territory (rather than relying on an ID# for sales Territory). The territory name is in another table so we have to do an INNER JOIN to connect to that table.  What you have to do is identify a common field in both tables then write the **INNER JOIN** syntax that connects them. It helps to use the **AS** alias to provide a one-letter 'AKA other name' for the table. So INNER JOIN is the syntax used to pull values from a second or third table (or more). Since data is separated into different tables you will most likely have to use a lot of INNER JOINS to get the columns of data that you want. This is tedious however so its a good idea to save your queries so you do not have to re-run them.  Also critical is to now see if any of the field names are in both tables. If they are then you have to specify table.[columnname]. As you can see this is made easier by using the one letter table name aliases. |
| **4** | USE AdventureWorks2019;  SELECT T.[Name], [CustomerID], [SalesOrderID], [SubTotal],[TaxAmt],[TotalDue]  FROM Sales.SalesOrderHeader AS S  INNER JOIN [Sales].[SalesTerritory] AS T ON T.TerritoryID =S.TerritoryID  WHERE [OrderDate] BETWEEN '1/1/2011' AND '12/31/2011'  ORDER BY T.Name, CustomerID | This query is the same as the prior except that the SQL syntax **ORDER BY** is used to sort the returned results first by the territory name, then by customers within that territory. The order of columns is adjusted to place them in a preferred order to match the search criteria. Notice that some columns have [] brackets and some don't. When you drag a column name from SQL Server's Object Explorer into the query, you automatically get brackets around the column name. This is fine and actually looks cool (MDX queries look similar).  The brackets are ONLY mandatory when a column name has a space within it.  The territory ID is deleted since we have the name. Hmm what table has the customer name in it. Wouldn't our reports look better if they had names? Can you find it and add another line with an INNER JOIN? |
| **5** | USE AdventureWorks2019;  SELECT COUNT(\*)  FROM Sales.SalesOrderHeader  WHERE [DueDate] BETWEEN '1/1/2011' AND '12/31/2011' | Here is a simple way to **count** the number of rows of data that exist in a table or in a range. It is often helpful to count up the # of rows before or after performing and INSERT INTO statement when you are copying data from one table to another. |
| **6** | USE AdventureWorks2019;  SELECT DISTINCT([TerritoryID])  FROM Sales.SalesOrderHeader  ORDER BY TerritoryID | Sometimes you will need to pull out values from one column of data inside a table. If the values have a high cardinality (just means there are many different values in the column), and the values repeat, then its a problem to retrieve just a single instance of each value.  This is where the **SELECT DISTINCT**(Column Name) syntax is very helpful. It will select just a single instance of each value in the column. |
| **7** | USE AdventureWorks2019;  SELECT t.Name, SUM(s.SubTotal) AS [Sub Total],  STR(Sum([TaxAmt])) AS [Total Taxes],  STR(Sum([TotalDue])) AS [Total Sales]  FROM Sales.SalesOrderHeader AS s  INNER JOIN Sales.SalesTerritory as t ON  s.TerritoryID = t.TerritoryID  GROUP BY t.Name  ORDER BY t.Name  ---------------------------------------------------- | This query introduces the most powerful **GROUP BY** syntax which groups records on a specified column, here Territory name. The GROUP BY term needs to be one of the columns specified in the SELECT portion of the query - usually first. The Sum(fieldname) AS [Field Name] syntax is used to **SUM** up the column based on whatever GROUP BY term is specified. You can also **COUNT**(Column Name), or **AVG**(Column Name).  Also show is a little bit of formatting **STR()**, you can wrap the STR() around a column name or calculation as shown to add a bit of formatting. There is a FORMAT operation shown later (which is new to SQLServer 2019). |
| **8** | USE AdventureWorks2019;  SELECT CustomerID, SUM([TotalDue]) AS Totals  FROM Sales.SalesOrderHeader  GROUP BY CustomerID  ORDER BY Totals DESC | This is similar to the query above, but it sorts the returned values on a different column than the GROUP BY column specified. The default is ascending which is smallest to largest (ASC) so you never have to specify that, but if you prefer largest to smallest sort - descending which is pretty useful in business, specify **DESC**. |
| **9** | USE AdventureWorks2019;  SELECT Top 10 CustomerID, STR(SUM(TaxAmt)) AS Totals  FROM Sales.SalesOrderHeader  WHERE [OrderDate] BETWEEN '1/1/2011' AND '12/31/2011'  GROUP BY CustomerID  ORDER BY Totals DESC | The SELECT **TOP N** syntax just cuts down the # of rows that are returned in the results set after the GROUP BY and ORDER BY syntax do their job. Its probably a good idea to use the TOP N syntax on the same column as the GROUP BY. |
| **10** | USE AdventureWorks2019;  SELECT CustomerID, SalesOrderID,  STR(SUM([SubTotal])) AS [Sub Total],  STR(SUM([TotalDue])) as [Total w Taxes & Freight]  FROM Sales.SalesOrderHeader  GROUP BY ROLLUP(CustomerID, SalesOrderID) | Record # 5, it has totals for Customer 1. The ROLLUP syntax can provide a subtotal for the terms specified in the **ROLLUP** term. Because SalesOrderID is one of the columns specified in the output, it needs to be included in the ROLLUP statement. |
| **11** | USE AdventureWorks2019;  SELECT [SalesOrderID], [SalesOrderDetailID], [ProductID], [OrderQty]  , STR([UnitPrice] \* [OrderQty]) AS [Line Total]  FROM [Sales].[SalesOrderDetail]  ------------------------------------------------------------------------------- USE AdventureWorks2019;  SELECT [SalesOrderID], COUNT([SalesOrderDetailID]) AS [# Line Items], STR(SUM([UnitPrice] \* [OrderQty])) AS [Sales Order Total]  FROM [Sales].[SalesOrderDetail]  GROUP BY SalesOrderID  ------------------------------------------------------------------------------  USE AdventureWorks2019;  SELECT p.Name AS ProductName,  NonDiscountSales = (OrderQty \* UnitPrice),  Discounts = ((OrderQty \* UnitPrice) \* UnitPriceDiscount)  FROM Production.Product AS p  INNER JOIN Sales.SalesOrderDetail AS sod ON p.ProductID = sod.ProductID  ORDER BY ProductName DESC; | Compare the results of the two queries. The first provides a list of line items on each Sales Order, the product ID, qty sold and a calculated field called line total. So the qty sold is calculated and displayed for every row in the retrieved results set. IF you used Visual Studio and ran this query in a data Adapter, then you would have direct access to the results set as they would be inside a in-memory data Table (which easily could be saved to a permanent database table back safely inside SQL Server). Ana analyst could use this table of values in a report and do further grouping to see totals for each Sales Order.  You can can also do the totals in a **GROUP BY** query as shown in the second query. So analyze these two queries and behold the power of the **GROUP BY** syntax.  The second query uses one record (row) per SalesOrder (each sales order in the SalesOrderHeader table has numerous line items in it - shown in the SalesOrderDetail table) - so theoretically you should be able to total up the line items in the SalesOrderDetail table and see the same values and the summary value shown in the salesorder header table (make sure to factor in for shipping and taxes). This query shows the total # of line items on the sales order and the total $ amount for the sales order  The third query uses a new way to give names to calculated fields, rather than use the AS [Column name] syntax, you can use the Column Name = (cools calculation), |
| **12** | SELECT S.ProductID, p.[Name], S.SpecialOfferID, SO.[Description]  , COUNT([OrderQty]) AS [Count]  , (AVG(UnitPrice))AS [Average Price]  , STR(SUM(LineTotal)) AS SubTotal  FROM Sales.SalesOrderDetail AS S  INNER JOIN [Sales].[SpecialOffer] AS SO  ON SO.[SpecialOfferID] = S.SpecialOfferID  INNER JOIN [Production].[Product] AS p ON p.[ProductID] = S.ProductID  GROUP BY S.ProductID, p.[Name], S.SpecialOfferID, SO.[Description]  ORDER BY S.ProductID, [Count] DESC | Total counts and $ revenue are provided for the different pricings (no discount or special offers) for each product. This report allows the analyst to see which pricing works the best for the sales of each product. The report can be improved by adding a date aspect.  The query has some important aspects to learn. First if the same field name is in two tables, then you have to specify the table that the field name is from (everywhere in the query). Also you can see the lines of code where the commas are first in the line, rather than the comma being the last item on the previous line. This is a pro trick, in that it helps you not forget the commas, and to make sure you do not place a comma after the last comma in the select statement.  One of the rules of the SQL syntax is that every field that is in the SELECT portion that is not a aggregating field (count, avg, sum, etc.) HAS to be in the GROUP BY |
| **13** | USE AdventureWorks2019;  SELECT p.ProductID, [Name], AVG([UnitPrice]) AS [Average List Price]  FROM Production.Product p  INNER JOIN [Sales].[SalesOrderDetail] s  ON p.ProductID = s.[ProductID]  GROUP BY p.ProductID, [Name]  Having AVG([UnitPrice]) > 1000  ORDER BY p.ProductID | What is the average price that we are selling our products for (factoring in promotional discounts)? The query shows the average list price for each product - for only the products that sold for >1000. The calculation of the average list price occurs in the GROUP BY expression. The filtering has to occur after the GROUP BY, so you do not use a WHERE statement after the GROUP BY, you use a **HAVING** statement |
| **14** | USE AdventureWorks2019;  SELECT sod.ProductID, AVG(sod.UnitPrice) AS [Average Price]  FROM Sales.SalesOrderDetail as sod  INNER JOIN Sales.SalesOrderHeader as soh  ON soh.SalesOrderID = sod.SalesOrderID  WHERE OrderQty > 10 AND YEAR(soh.OrderDate) = '2011'  GROUP BY ProductID  ORDER BY [Average Price] DESC | The following example finds the average price of each type of product where the product sku was sold more than 10 units for any selected year (you can change the 2011 to 2001, 2002). This query uses the **YEAR(column name)** syntax to extract the year from the OrderDate field.  You can orders the results by average price (lowest price on top). First you filter the columns, then join the tables, then filter the rows, then group the records, then order the table results.  Its important to get comfortable using the WHERE statement to filter the results returned before you group them. The **ORDER BY** statement can read the alias field. |
| **15** | USE AdventureWorks2019;  SELECT ProductID, STR(SUM(LineTotal)) AS Total  FROM Sales.SalesOrderDetail  GROUP BY ProductID  HAVING SUM(LineTotal) BETWEEN 1000000 AND 2000000  ORDER BY Total | To see the products that have had total sales between $1M and $2M  Here is another example of using GROUP BY syntax, the **HAVING** filter, and the ORDER BY sorting process. Notice you can use the alias Total in the ORDER BY but not the **HAVING.** |
| **16** | USE AdventureWorks2019;  SELECT PC.Name AS Category, PS.Name AS Subcategory,  DATEPART(yy, SOH.OrderDate) AS [Year]  , 'Q' + DATENAME(qq, SOH.OrderDate) AS [Qtr]  , STR(SUM(DET.UnitPrice \* DET.OrderQty)) AS [$ Sales]  FROM Production.ProductSubcategory AS PS  INNER JOIN Sales.SalesOrderHeader AS SOH  INNER JOIN Sales.SalesOrderDetail DET ON SOH.SalesOrderID = DET.SalesOrderID  INNER JOIN Production.Product P ON DET.ProductID = P.ProductID  ON PS.ProductSubcategoryID = P.ProductSubcategoryID  INNER JOIN Production.ProductCategory PC ON PS.ProductCategoryID = PC.ProductCategoryID  WHERE YEAR(SOH.OrderDate) = '2002' or YEAR(SOH.OrderDate) ='2011'  GROUP BY DATEPART(yy, SOH.OrderDate), PC.Name, PS.Name, 'Q'  + DATENAME(qq, SOH.OrderDate), PS.ProductSubcategoryID  ORDER BY Category, SubCategory, [Qtr] | The use of INNER JOIN's can get complex - here three tables are joined together at the same time.  Notice the use of **DATEPART(column name)** - you specify what part of the date that you want to extract from the date field - here we use the yy to get the year, and qq to get the month.  Also notice the concatenation 'Q' + DATENAME(date field). this allows the creation of a field that reads Q3 or Q$ etc.  Go ahead and experiment with the ORDER BY syntax to see different ways to sort the contents. |
| **17** | USE AdventureWorks2019;  SELECT CustomerID, SalesOrderID, TaxAmt, SUM(TaxAmt)  OVER(PARTITION BY CustomerID ORDER BY CustomerID)  FROM Sales.SalesOrderHeader | The OVER clause can be used to compute aggregated values such as moving averages, cumulative averages, running totals, or a TOP N per group result. This feature is not installed on the ot-devst-12. |
| **18** | USE AdventureWorks2019;  SELECT [Name],  SUM(CASE WHEN YEAR([OrderDate]) = 2001 THEN ([SubTotal]) END) AS [2001],  SUM(CASE WHEN YEAR([OrderDate]) = 2002 THEN ([SubTotal]) END) AS [2002],  SUM(CASE WHEN YEAR([OrderDate]) = 2011 THEN ([SubTotal]) END) AS [2011],  SUM(CASE WHEN YEAR([OrderDate]) = 2004 THEN ([SubTotal]) END) AS [2004],  SUM([SubTotal]) AS Total  FROM [Sales].[SalesOrderHeader] AS S  INNER JOIN [Sales].[SalesTerritory] AS T ON T.[TerritoryID] = S.TerritoryID  GROUP BY [Name]  ORDER BY [NAME] | When you realize that SQL Server performs the aggregation of the >31,000 records in less than a second, then you get an idea of how powerful database technology is. When you need a lot of data crunched and aggregated, let SQL Server do it for you!  This syntax is sort of a crosstab, that uses a select statement. Every single row in the SalesOrder Header table is categorized and totalled based on the YEAR of the order date or the sales order. |
| **19** | SELECT s.[ProductSubcategoryID], s.[Name], p.[Name], p.[ProductNumber], p.[SafetyStockLevel], p.[ReorderPoint], inv.[Quantity], inv.Shelf, inv.Bin  FROM [Production].[Product] as p  INNER JOIN [Production].[ProductInventory] as inv  ON p.[ProductID] = inv.[ProductID]  INNER JOIN [Production].[ProductSubcategory] as s  ON s.ProductCategoryID = p.ProductSubcategoryID  WHERE inv.[Quantity] < p.[ReorderPoint] AND p.ProductSubcategoryID IS NOT NULL  ORDER BY s.[ProductSubcategoryID] | Here you see how to select only the rows that are not NULL (have null values in a certain column). Here the low-cost parts (nuts and bolts) are filtered out since they are consumed during product and are not tracked in a product sub-category (Product categories and sub-categories are used for sales catalog tracking). The low cost nuts and bolts have the term NULL in their product subcategory ID field. While every part is important, this report is designed to identify the parts that need to be re-stocked. |
| **20** | USE [AdventureWorks2019]  SELECT CONCAT([FirstName], + ' ', +[LastName]) AS Employee  , t.[TerritoryID],t.[Name] AS [Territory], s.SalesLastYear as [Emp Sales Last Year]  , [SalesQuota] AS [Emp Sales Quota], s.SalesYTD AS [Emp Sales YTD]  , [Bonus] AS [Emp Bonus], [CommissionPct] as [Emp Commission%]  , [HireDate], [MaritalStatus], t.[SalesLastYear] AS [Territory Sales Last Year]  , t.[SalesYTD] AS [Territory Sales YTD]  FROM HumanResources.Employee as e  inner join Person.Person as p on p.BusinessEntityID = e.BusinessEntityID  inner join Sales.SalesPerson as s on s.BusinessEntityID = e.BusinessEntityID  inner join [Sales].[SalesTerritory] as t on t.[TerritoryID] = s.TerritoryID  ORDER BY TerritoryID | Here is a base query to perform human resources/salesrep performance analysis |

# Resources:

One example provided by MSDN.

* <http://msdn.microsoft.com/en-us/library/ms187731(SQL.100).aspx>,

Additional examples available at

* <http://technet.microsoft.com/en-us/library/ms187731.aspx>