ADVANCED SQL FOR BEGINNERS

Real world SQL for aspiring analysts

About SQL

SQL is a programming language used for extracting data from databases and preparing datasets for reporting, analytics and even machine learning. There are many database platforms – but, the good news is that if you learn it on one platform, then you can use it on any platform (ex. SQL Server, Snowflake, PostgreSQL, Oracle, MySQL, DB2, etc). Almost every organization in the United States uses SQL in some fashion. SQL can be used in processes such as basic ad-hoc queries, data preparation for visualization, and ETL/ELT.

Basic Ad-Hoc Queries

SQL can be written against databases for on-the-fly extracts. Perhaps, the need is a one-time extract to identify the scope of an operational issue that popped up in an organization. An analyst may write and execute a SQL query in a SQL editor, then dump the results to a spreadsheet tool and prepare a report manually.

Data Preparation for Visualization

If your goal is to become a complete analyst focused on both data preparation and visualization, then you will also want to learn more about business intelligence (BI) tools. A business intelligence tool allows you to marry data preparation with data visualization and storytelling. Most BI tools use SQL to produce datasets that are fed into visuals such as graphs, maps, and tables. Common BI tools include Tableau, Power BI, Looker, Domo, Qlik, etc.

Source Data In ETL/ELT Tools

There is often a need to take data from multiple data sources and integrate it all into one central database (ex. data warehouse or data lake). The process of loading data from multiple database platforms to a single source of truth is called ETL (Extract, Transform, Load) or ELT (Extract, Load, Transform). SQL is often used in these processes to select the right source data and transform it into a more usable format to simplify reporting and analytics.

Common Database Environment

When working with SQL, you should understand the basic database environment (at a high level). A SQL query must pull data from a database. A database is made up of schemas, each of which holds related tables. A database could be one of many that live on a server (either cloud or on-premise). Each of these components will be described in more detail.

Server

A server is what holds a company's databases and other tools. Over the past several years, there has been a massive shift from servers physically hosted on site to servers hosted in the cloud. Servers allow systems to run with a high level of performance and availability. Server management is generally handled by database administrators and network engineers and therefore, isn't something that you will be likely to handle.

Database

A database is hosted on a server and contains information organized in a usable manner. A database contains one or more schemas, each of which contains one or more tables.

Schema

A schema is a collection of similar or related information within a database. For example, a database may have three schemas – one for employee information, one for customer information and one for sales information. Within each schema, there may be one or more tables. A customer schema may contain multiple tables that describe each customer. For example, you may have one table for customer demographics and another for customer addresses (shipping and residential). In the sales schema, you may have one table that lists all purchases and another that stores currency exchange rates for each country.

Table

A table is an object within a schema that organizes data into rows and columns – just like an Excel spreadsheet. Tables typically have an unlimited number of rows along with many columns. This allows for databases to grow over time.

Column

A column is a measure or attribute within a table. A measure will typically be a numeric value and an attribute will typically be a text or date value. Attributes describe data and measures quantify it. Common column data types include:

String

Alphanumeric values including special characters.

Float

Numeric values including decimals.

Integer

Numeric values excluding decimals (whole numbers and may include negatives).

Date

Date values excluding time stamps.

Datetime

Date values including time stamps.

Common Database Structures

Relational Database

A relational database is a structure that stores data in various tables based on common elements. Related tables may be joined together on key fields.

For example, a supplier may have a database with:

- A sales table containing a list of everything that was sold along with a unique customer identifier.
- A customer table containing the same unique customer identifier along with various elements related to each customer (name, phone number, email address, gender, etc.).

These two tables can then be linked together on the unique customer identifier to provide a comprehensive view of what each customer is purchasing.

Star Schema

A star schema is a popular, specific type of relational database. It is made up of fact and dimension tables.

Fact Table

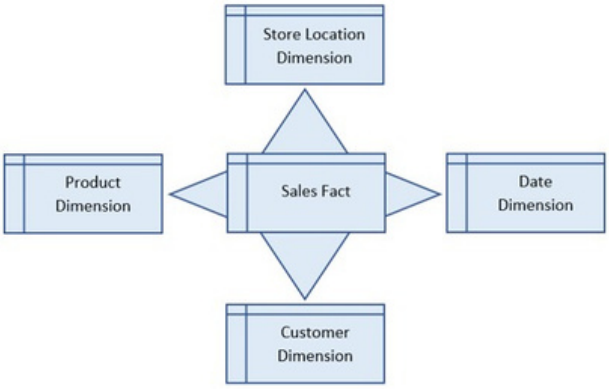
Fact tables will contain measurements and facts about business transactions. For example, a sales table would be considered a fact table. Fact tables may include metrics like price, quantity, tax rate and date of sale. In addition, the fact table will contain unique identifiers that help identify various attributes surrounding the sale. These unique identifiers are called foreign keys and they are used to join to the primary keys of dimension tables (like our sales-customer example in the Relational Database section above).

Dimension Table

Dimension tables hold attributes that are associated with fact tables. Examples of dimension tables include customer information, product information, or currency exchange rates. Dimension tables will have a primary key (like a unique identifier) that allows us to link back to the related foreign key on the fact table(s).

The key benefits of this structure are data storage and query performance. For example, let's say the customer purchases the same item each week (like paper towels). Instead of storing each customer's name, phone number, email address, gender, etc. each week (52 times per customer), we can store it one time and connect the fact and dimension tables together on the foreign key (in fact table) and primary key (in dimension table) and get the same result with significantly less storage. More to come on table joining.

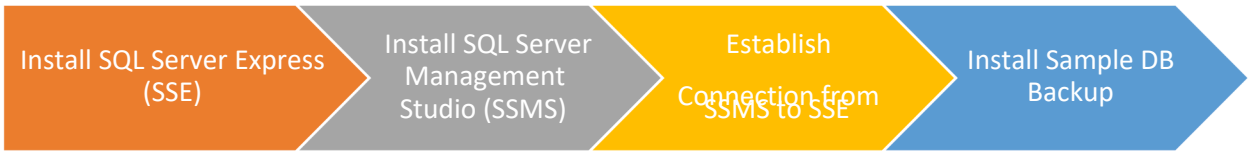
A star schema would look something like this:


Store Location Dimension
Product Dimension
Sales Fact
Date Dimension
Customer Dimension

Getting Started

This book will be based on the AdventureWorks2019 sample database in SQL Server. Detailed setup instructions, download links, and tool navigation tips may be found in the appendix.

Also included in the appendix is a database diagram for the Person, Human Resources, and Sales schemas. Our exercises are exclusively limited to those three schemas. Please refer to these database diagrams for a visual on table relationships.



Basic SQL Query Types

There are multiple types of SQL queries. There are queries that are used for database management and modifying data such as:

- Create Table - Create new tables from scratch
- Delete – Remove existing table data
- Update - Modify existing table data
- Insert Into - Insert records into tables

We will touch on the four topics listed above. However, we will mostly focus on queries where we select data to view and analyze. In these queries, the SELECT and FROM clauses must appear in each query. The WHERE clause is used in nearly every query, although it is not a requirement. A little bit more about these in the next few sections.

Select

The SELECT statement is used to identify the fields that you wish to view in the query output. Let's say that one table has 50 fields. If you want to view 10 of the 50 fields, you list them in this portion of the query. As you will find out later in this book, you will also perform mathematical and other functions in this section of the query. If you want to see every field in your output, use 'SELECT *'.

Example 1: Display a person's first, middle and last names along with his or her title.

```
SELECT Title, FirstName, MiddleName, LastName
```

Results: No results – you will get an error because you haven't specified a table to select from. This will be discussed in the next section.

From

The FROM statement is used to identify the tables that the query output will be selected from. As previously mentioned, a database may be made up of a couple of tables or hundreds of tables. Each table contains one or more fields of data. A field is like a column in an Excel table. Tables may be joined together to merge fields into one larger dataset. Joining will be discussed later in this book.

Each table used will be listed below the word FROM and will contain three components: the database name, the schema name, and the table name. As previously mentioned, a schema is used to organize similar groups of data within a complex database.

The syntax is as follows: FROM [Database Name].[Schema Name].[Table Name]

Example 2: Display a person's first, middle and last names from the Person table

```
SELECT Title, FirstName, MiddleName, LastName

FROM
AdventureWorks2019.Person.Person
```

Results (Snapshot Only): You should get 19,972 records.

Title	FirstName	MiddleName	LastName
NULL	Ken	J	Sánchez
NULL	Terri	Lee	Duffy
NULL	Roberto	NULL	Tamburello
NULL	Rob	NULL	Walters
Ms.	Gail	A	Erickson
Mr.	Jossef	H	Goldberg

Order By

The ORDER BY clause allows us to sort a dataset by any number of columns in ascending or descending order.

At the very end of a query, you can accomplish this by adding ORDER BY column1, column2, etc.

Ascending order is the default. If you want a column to be sorted in descending order, you can add DESC after the column name. For example, ORDER BY column1, column2 DESC. In this case, column1 will come in ascending order and column column2 will come in descending order.

Example 3: Using the example in the prior section, sort by last name then first name.

```
SELECT Title, FirstName, MiddleName, LastName

FROM
AdventureWorks2019.Person.Person

ORDER BY
LastName,
FirstName
```

Results (Snapshot Only): You should still get 19,972 records, but sorted.

Title	FirstName	MiddleName	LastName
Mr.	Syed	E	Abbas
Ms.	Catherine	R.	Abel
Ms.	Kim	NULL	Abercrombie
NULL	Kim	NULL	Abercrombie
NULL	Kim	B	Abercrombie
NULL	Hazem	E	Abolrous

Alias Names

Alias names are just like when a person has an alias name. It means that we can call a field or table something else.

Why would we want to call a field or table something else?

1. Each field shown in the SELECT statement lives in a database table. Therefore, we need the database table listed before each field name in our SELECT statement. Without it, the query won't know which table to pull each value from. When we reference a table, writing [A] is a hell of a lot easier than writing [Database Name].[Schema Name].[Table Name] every time we need to reference a table.
2. We sometimes need to rename columns because database naming conventions aren't always clean.
3. When we perform formulas on columns (in the SELECT statement), we need to provide a column name in our output.

It is very important to note that while you can reference an alias name provided to a ***table*** anywhere in the query, you are only able to reference the alias name provided to a ~~***column***~~ in the ORDER BY statement.

The syntax for a table alias is: [Database Name].[Schema Name].[Table Name] [Table Alias Name]

The syntax for a column alias is: [Table Alias Name].[Column Name] as [Column Alias Name]

Example 4: Alias name the Person table and each of the columns shown in the SELECT.

```
SELECT
A.Title as Title, -- Brackets not needed because no space in alias name
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name]

FROM
AdventureWorks2019.Person.Person A

ORDER BY
LastName,
FirstName
```

Results (Snapshot Only): You should get 19,972 records and you'll notice we now have a space between words in your column headers (example below).

Title	First Name	Middle Name	Last Name
NULL	Ken	J	Sánchez
NULL	Terri	Lee	Duffy
NULL	Roberto	NULL	Tamburello
NULL	Rob	NULL	Walters
Ms.	Gail	A	Erickson
Mr.	Jossef	H	Goldberg

Filtering Using Where

Filtering is one of the most important aspects of a SQL query. It is where many of the business rules are applied. Filtering takes place in three places: the FROM, WHERE and HAVING statements. However, this section will focus on filtering in the WHERE statement. Filtering in the FROM and HAVING statements will be discussed later in this book.

Filtering is as simple as it sounds. It's just a way to eliminate unwanted records from your query results. When you are in the learning stages of writing SQL queries, I highly recommend that you write on a separate sheet of paper each of the tables that you plan to use in your query then think through every single filter and list them for each table. If you miss even one filter, you can possibly return millions of unwanted records.

Filters use standard operators such as =, <>, >, <, in, like, between (descriptions shown in the next section).

You can include as many filters as you need, just separate them by AND or OR. If you use OR, please consider order of operations and use () at the beginning and end of the OR statement(s).

Common use of filters in WHERE clause:

1. Eliminate unwanted records (ex. old records, irrelevant records, etc.).
2. Limit to specific date ranges.
3. Limit to specific status codes.

Example 5: Filter our Person table to anyone listed as an employee (indicated by the code of EM) with the first name of Michael and modified during 2009.

```
SELECT
A.PersonType as [Person Type],
A.Title as Title,
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name]


FROM
AdventureWorks2019.Person.Person A

WHERE
A.PersonType='EM' AND
A.FirstName='Michael' AND
A.ModifiedDate BETWEEN '2009-01-01' and '2009-12-31'
```

Results: You will see six records.

Person Type	Title	First Name	Middle Name	Last Name
EM	NULL	Michael	W	Patten
EM	NULL	Michael	T	Entin
EM	NULL	Michael	NULL	Raheem
EM	NULL	Michael	Sean	Ray
EM	NULL	Michael	L	Rothkugel
EM	NULL	Michael	T	Vanderhyde

Common Operators to Use in Filter

As mentioned in the previous section, filtering is extremely important. This chapter will highlight some of the most common operators to be used in filters. Note – this list is not comprehensive as there could be other types of operators.

In a filter, you can compare fields against other fields, fields against fixed/hard-coded values, and fields against calculations (ex. date look backs). Each scenario will be practiced in this book. Note – Ensure that you are using the same data types; otherwise, you will get a data type conversion error. For example, you can't filter a string field by a number/date value.

Example 6: The below query shows filtering using operators such as equal to, greater than, less than and not equal to. We'll filter the SalesOrderDetail table using these operators. I didn't include the results here because this example is just for reference on proper syntax.

Tip:

1. In Example 6, you'll see that we are not wrapping our comparison value, **5**, with single quotes because it's a number. We only wrap text and dates with single quotes as shown in Example 7 below.

```
SELECT
A.SalesOrderID,
A.SalesOrderDetailID,
A.CarrierTrackingNumber,
A.OrderQty,
A.ProductID,
A.SpecialOfferID,
A.UnitPrice,
A.UnitPriceDiscount,
A.LineTotal,
A.rowguid,
A.ModifiedDate

FROM
AdventureWorks2019.Sales.SalesOrderDetail A

WHERE
A.OrderQty = 5 OR -- Equal to
A.OrderQty > 5 OR -- Greater than
A.OrderQty >= 5 OR -- Greater than or equal to
A.OrderQty < 5 OR -- Less than
A.OrderQty <= 5 OR -- Less than or equal to
A.OrderQty <> 5 -- Not equal to
```

Example 7: The below query shows filtering using operators such as IN, LIKE and BETWEEN. We'll filter the Person table using these operators. Again, I didn't include the results here because this example is just for reference on proper syntax.

```
SELECT
A.PersonType as [Person Type],
A.Title as Title,
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name],
A.ModifiedDate as [Modified Date]


FROM
AdventureWorks2019.Person.Person A

WHERE
A.FirstName IN ('Thomas','Michael','Mary') AND
-- Use IN when you want to filter based on a list of values; Could be number, dates or
strings
A.ModifiedDate BETWEEN '2009-01-01' AND '2009-12-31' AND
-- Use BETWEEN when you want to filter based on a range of values; Could be number,
dates or strings
A.LastName LIKE '%ra%'
-- Use LIKE when you want to see if a string contains certain characters; Use % as
your wild character
-- '%ra' indicates that any character can exist BEFORE ra in the string and it will
end in ra
-- 'ra%' indicates that the string MUST begin with ra and may contain anything after
-- '%ra%' indicates that the string can contain ra anywhere in the string
```

OR vs. AND

A very important (sometimes overlooked) aspect of filtering is understanding the syntax associated with OR/AND. Just like a calculator in math class, SQL might not always process things in the order that you desire so your syntax needs to be specific. It's very simple, just use parentheses.

Example 8: Show anyone from the person table with a last name that contains the string 'MI' or 'TH'. Please only include employees.

Option A: No parentheses.

```
SELECT
A.PersonType as [Person Type],
A.Title as Title,
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name],
A.ModifiedDate as [Modified Date]


FROM
AdventureWorks2019.Person.Person A

WHERE
A.LastName LIKE '%mi%' OR
A.LastName LIKE '%th%' AND
A.PersonType = 'EM'
```

Results (Snapshot Only): You will see 553 records because the query is confused and doesn't know what order to apply the filters (incorrect option). Note, the results shown below are not the first six records as I wanted to show how we get multiple person type values instead of only the requested 'EM' value.

Person Type	Title	First Name	Middle Name	Last Name	Modified Date
EM	NULL	Dragan	K	Tomic	2/4/09
EM	NULL	Ben	T	Miller	3/2/10
SP	NULL	Linda	C	Mitchell	5/24/11
SC	Ms.	Margaret	J.	Smith	4/15/15
SC	Ms.	Sandra	J.	Altamirano	7/31/13
SC	Ms.	Jane	N.	Carmichael	5/30/13

Option B: Apply parentheses before and after the OR.

```
SELECT
A.PersonType as [Person Type],
A.Title as Title,
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name],
A.ModifiedDate as [Modified Date]


FROM
AdventureWorks2019.Person.Person A

WHERE
(
A.LastName LIKE '%mi%' OR
A.LastName LIKE '%th%'
) AND
A.PersonType = 'EM'
```

Reesults (Snapshot Only): You will see 20 records because the query knows to first look for any names that contain 'MI' or 'TH' THEN it will filter those people down to only employees (correct option). Note, in this case, I did paste the first six records as they all show the 'EM' 'Person Type' values.

Person Type	Title	First Name	Middle Name	Last Name	Modified Date
EM	NULL	James	R	Hamilton	1/27/09
EM	NULL	Terry	J	Eminhizer	2/23/09
EM	NULL	Sandeep	P	Kaliyath	1/10/10
EM	NULL	Vidur	X	Luthra	12/25/08
EM	NULL	Gigi	N	Matthew	1/9/09
EM	NULL	Mark	K	McArthur	1/16/09

Table Joins

Table joins are extremely simple, but one of the most important things you need to understand. They are used to join two (or more tables) together to:

1. Increase the number of fields available to display in your results.

2. Filter your records based on existing (or non-existing) scenarios that exist in related tables.

Don't waste time reading about joins in other books. It will just confuse you because you only need to know about two types of joins. The two types are INNER JOIN and LEFT JOIN (explained later in this section). In other articles, you'll see LEFT OUTER JOIN, RIGHT OUTER JOIN, RIGHT JOIN, FULL JOIN, NATURAL JOIN, etc.

Table joins belong in the FROM statement and follow the below syntax:

```
FROM
AdventureWorks2019.Person.Person A
```

```
                                INNER JOIN
  AdventureWorks2019.Person.EmailAddress B
  ON A.BusinessEntityID=B.BusinessEntityID
```

Specifies the type of join that you want to use (INNER JOIN or LEFT JOIN)

Specifies the field(s) that you want to marry together from each table; You will typically join on common field names, but the specific rules may depend on the database. You will notice that we identify which table each field comes from (using our table ALIAS names; ex. [A] and [B]). If multiple fields need to be joined together, separate them by AND (see below for an example).

```
FROM
AdventureWorks2019.Person.Person A INNER JOIN
AdventureWorks2019.Person.EmailAddress B
 ON A.BusinessEntityID=B.BusinessEntityID
 AND A.SampleJoinField=B.SampleJoinField -- Just for illustrative purposes
```

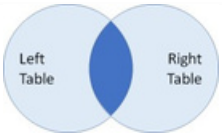
VERY IMPORTANT: When identifying the tables that you want to use in your query, make sure that you are confirming all relevant joins are applied. One missed join condition could result in millions of unwanted records (or billions depending on your database size).

Next, we'll look at the difference between INNER JOIN and LEFT JOIN.

Inner Join

An INNER JOIN will return records from two tables when ***both*** tables contain common values in the fields specified in your join condition.

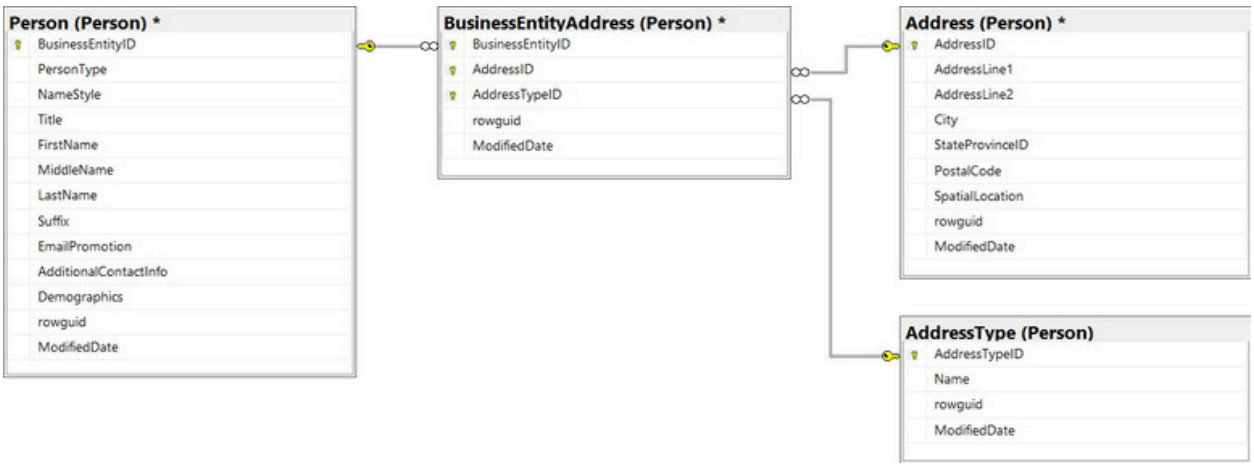
This is illustrated by the visual below where the 'Left Table' is the first table you list after the FROM statement and the 'Right Table' is the one listed next. The darker shade represents which records will be returned from each table.



Example 9: We will continue to look for employees with a last name containing 'MI' or 'TH', but let's ***only*** show the ones with the home city of Kenmore. Note – you'll need to touch a few tables to capture this information because the data points are spread out. The Entity Relationship (ER) diagram below shows how these tables relate to each other.

Tables Used: Sometimes, you pass through bridge tables to get to your destination. In Example 9, we need to bring in the BusinessEntityAddress and AddressType tables before we get to the Address table.

1. The Person table is the person master table that includes employees, customers, etc.
2. The Address table contains the street addresses of all business entities.
3. The BusinessEntityAddress table contains the unique address ID and address type for each person.
4. The AddressType table contains the description of the address type (ex. home, office, etc.).



Tips:

1. Since we only want to show the folks with a home city of Kenmore, we will use INNER JOINs – essentially, we are using the join to filter unwanted records (ie. eliminating people without a city of Kenmore).
2. Now that we're starting to add more filters and table joins, it becomes a little easier if you move some filters to your FROM statement (highlighted in green). I like doing this because it forces me to think about all relevant filters/joins for each table in my query…almost like a checklist.

```
SELECT
A.PersonType as [Person Type],
A.Title as Title,
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name],
A.ModifiedDate as [Modified Date],
D.City as City,
C.Name as [Address Type]

FROM
AdventureWorks2019.Person.Person A INNER JOIN
AdventureWorks2019.Person.BusinessEntityAddress B
 ON A.BusinessEntityID = B.BusinessEntityID INNER JOIN
AdventureWorks2019.Person.AddressType C
 ON B.AddressTypeID = C.AddressTypeID
 AND C.Name = 'Home' INNER JOIN
AdventureWorks2019.Person.Address D
 ON B.AddressID = D.AddressID
 AND D.City = 'Kenmore'

WHERE
(
A.LastName LIKE '%mi%' OR
A.LastName LIKE '%th%'
) AND
A.PersonType = 'EM'
```

Results (Snapshot Only): You will see three records, each having a home city of Kenmore.

Person Type	Title	First Name	Middle Name	Last Name	Modified Date	City	Address Type
EM	NULL	David	P	Hamilton	12/27/08	Kenmore	Home
EM	NULL	Denise	H	Smith	1/29/09	Kenmore	Home
EM	NULL	Dylan	A	Miller	2/1/09	Kenmore	Home

Left Join

A LEFT JOIN will return all records from the first table listed after your FROM statement (your left table) regardless of whether common values are present in your next table (your right table). If values are present in your right table, they will also be shown (if included in the SELECT clause).

Again, this is illustrated by the below visual where the 'Left Table' is the first table you list after the FROM statement and the 'Right Table' is the one listed next. The darker shade represents which records will be returned from each table.

Example 10: Let's stick with employees having a last name containing 'MI' or 'TH', but let's show ***all*** these employees regardless of their home city. However, we only want to show the home city if it says Kenmore.

Tables Used: Same as Example 9.

Tips:

1. We are using the same query from Example 9 as our starting point.
2. Since we don't want to eliminate employees without the home city of Kenmore, we will switch our last join to a LEFT JOIN. Essentially, we are using the join to identify and show the person's home city if it is Kenmore.

```
SELECT
A.PersonType as [Person Type],
A.Title as Title,
A.FirstName as [First Name],
A.MiddleName as [Middle Name],
A.LastName as [Last Name],
A.ModifiedDate as [Modified Date],
D.City as City,
C.Name as [Address Type]

FROM
AdventureWorks2019.Person.Person A INNER JOIN
AdventureWorks2019.Person.BusinessEntityAddress B
 ON A.BusinessEntityID = B.BusinessEntityID INNER JOIN
AdventureWorks2019.Person.AddressType C
 ON B.AddressTypeID = C.AddressTypeID
 AND C.Name = 'Home' LEFT JOIN
AdventureWorks2019.Person.Address D
 ON B.AddressID = D.AddressID
 AND D.City = 'Kenmore'
```

```
WHERE
(
A.LastName LIKE '%mi%' OR
A.LastName LIKE '%th%'
) AND
A.PersonType = 'EM'
```

Results (snapshot only): We are now back to the same 20 records from Example 7, Option B. Take notice of the NULL city values – this means that the LEFT JOIN couldn't find a Kenmore address for Gigi, Terry, James, etc. But, because we used a LEFT JOIN, we still see the records.

Person Type	Title	First Name	Middle Name	Last Name	Modified Date	City	Address Type
EM	NULL	Dylan	A	Miller	2/1/09	Kenmore	Home
EM	NULL	Gigi	N	Matthew	1/9/09	NULL	Home
EM	NULL	Terry	J	Eminhizer	2/23/09	NULL	Home
EM	NULL	James	R	Hamilton	1/27/09	NULL	Home
EM	NULL	Mark	K	McArthur	1/16/09	NULL	Home
EM	NULL	Thomas	R	Michaels	2/19/09	NULL	Home

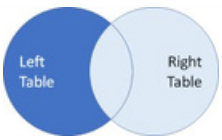
Using Left Join to Filter on Mismatches

A different way of using LEFT JOINs is to eliminate matching records across two tables. Often, we need to know when certain scenarios do ***not*** exist. For example:

1. We want to find out which customers haven't purchased anything recently.
2. We want to find out which employees haven't changed jobs recently.

To do this, we use a LEFT JOIN then exclude the matched records. This is accomplished by returning all rows where the table that we are LEFT JOINing ***to*** has null values. In other words, we failed to find a match; therefore, we want to keep the null records.

Finally, this is illustrated by the visual below where the 'Left Table' is the first table you list after the FROM statement and the 'Right Table' is the one listed next. The darker shade represents which records will be returned from each table.

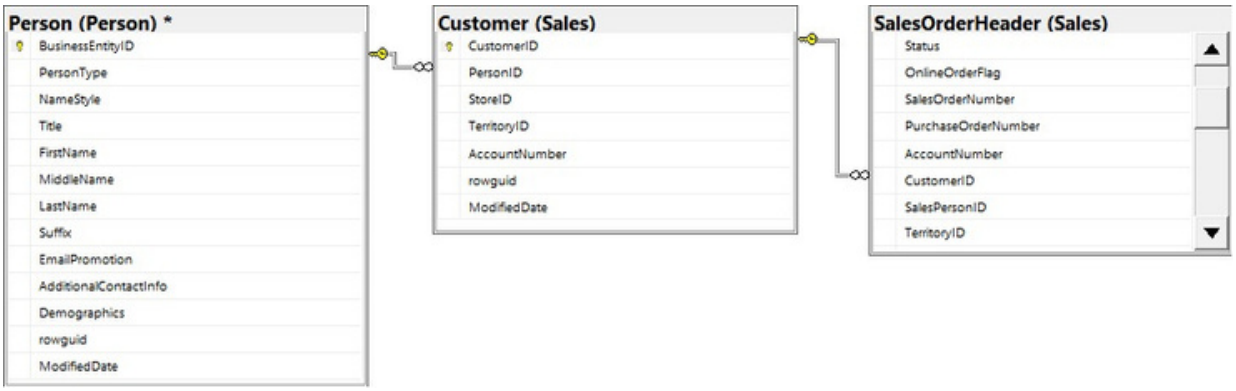


Example 11: Let's switch gears to the Sales schema and identify which customers haven't purchased anything in 2011. The Entity Relationship (ER) diagram below shows how these tables relate to each other.

Tables Used: We'll use a couple new tables in this example:

1. The Customer table is associated with the sales transaction.
2. The Person table is the person master table that includes employees, customers, etc.
3. The SalesOrderHeader table is the sales table used to track details about the entire order.

   For example, shipping details, payment details, etc.




```
SELECT
A.CustomerID as [Customer ID],
A.AccountNumber as [Account Number],
B.FirstName as [First Name],
B.LastName as [Last Name],
C.OrderDate as [Order Date],
C.TotalDue as [Total Due]

FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID LEFT JOIN
AdventureWorks2019.Sales.SalesOrderHeader C
 ON A.CustomerID = C.CustomerID
 AND C.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

WHERE
C.TotalDue Is Null
```

Results (snapshot only): You will see 17,713 records. Each of these records will have a NULL Order Date and Total Due value because the combination of join and filter criteria eliminate all successful matches.

Customer ID	Account Number	First Name	Last Name	Order Date	Total Due
29485	AW00029485	Catherine	Abel	NULL	NULL
29488	AW00029488	Pilar	Ackerman	NULL	NULL
28866	AW00028866	Aaron	Adams	NULL	NULL
13323	AW00013323	Adam	Adams	NULL	NULL
21139	AW00021139	Alex	Adams	NULL	NULL
19419	AW00019419	Allison	Adams	NULL	NULL

VERY IMPORTANT: When using LEFT JOINs, you can easily turn them into INNER JOINs (on accident) if you aren't careful with your WHERE clause. For example, if you change 'Is Null' (highlighted above) to 'Is Not Null' then you've got an INNER JOIN since you will return only records where you ***found*** a match on the SalesOrderHeader table. Because of this, you should avoid placing filters in the WHERE clause from your right table(s) – except for this specific scenario (keeping mismatches only).

Aggregation

One of the most powerful uses of SQL is aggregation. Again, it is a very simple concept, but many people struggle with it. Aggregation is when you collapse a detailed dataset into a summarized one. Picture an Excel file with a data tab and a pivot table tab. SQL aggregation is the equivalent of the pivot table tab.

The most common aggregate functions that you will use are COUNT(), SUM(), MIN(), MAX() and AVG(). To aggregate a measure, we first define a group to perform the aggregation within. For example, if you want to calculate the sum of sales for each salesperson, your group definition is the salesperson and the aggregate function is the sum of sales.

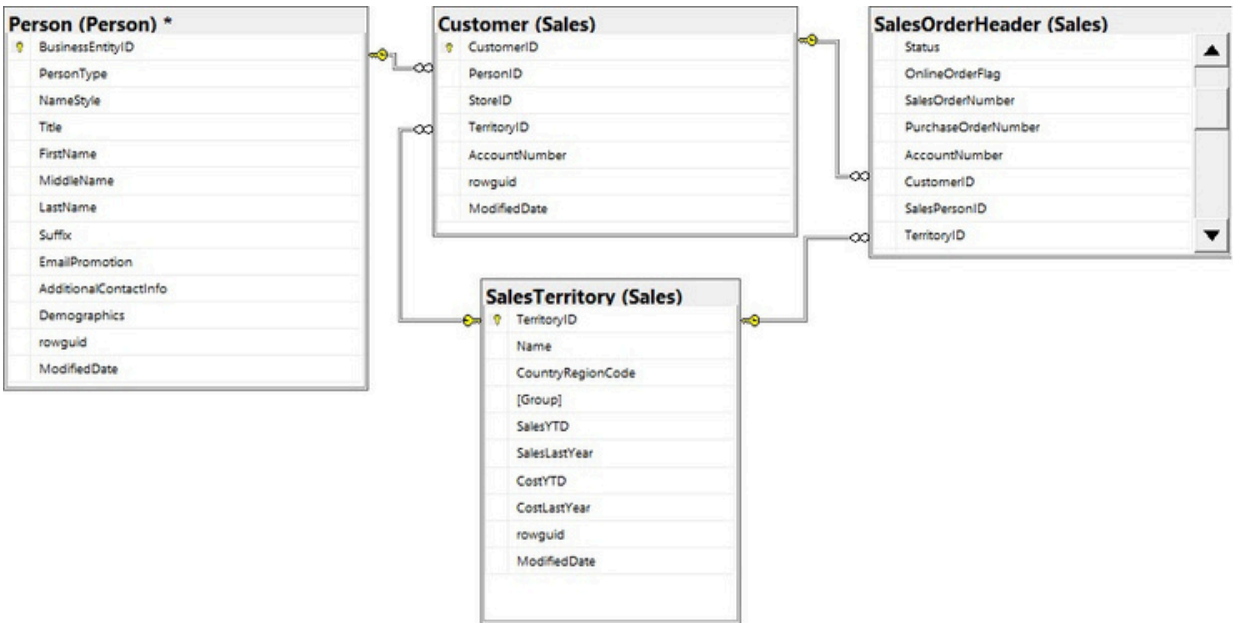
When any value within the defined group changes, the aggregation will restart (ie. SUM(), MIN(), etc).

Example 12: Sticking with the Sales schema, let's first list all the detailed sales transactions in 2011. Include the region, then we'll aggregate them in Example 13. The Entity Relationship (ER) diagram below shows how these tables relate to each other.

Tables Used: Same as Example 11, but add the SalesTerritory table:

1. The SalesTerritory table is the table that identifies the related territory for each of the

   locations.



```
SELECT
D.Name as [Territory Name],
A.CustomerID as [Customer ID],
A.AccountNumber as [Account Number],
B.FirstName as [First Name],
B.LastName as [Last Name],
C.OrderDate as [Order Date],
C.TotalDue as [Total Due]

FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID INNER JOIN
 AdventureWorks2019.Sales.SalesOrderHeader C
  ON A.CustomerID = C.CustomerID INNER JOIN
AdventureWorks2019.Sales.SalesTerritory D
 ON C.TerritoryID = D.TerritoryID

WHERE
C.OrderDate BETWEEN '2011-01-01' and '2011-12-31'
```

Results (Snapshot Only): You'll see 1,607 records like the below sample. As you can see, these records aren't usable if someone wants to know what was sold in the Central region.

Territory Name	Customer ID	Account Number	First Name	Last Name	Order Date	Total Due
Central	29486	AW00029486	Kim	Abercrombie	8/1/11	18768.2086
Central	29486	AW00029486	Kim	Abercrombie	10/31/11	75865.1515
Northeast	29487	AW00029487	Humberto	Acevedo	8/1/11	23095.3463
Northeast	29487	AW00029487	Humberto	Acevedo	10/31/11	19265.5486
Southeast	29484	AW00029484	Gustavo	Achong	8/1/11	4560.2864
Southwest	29170	AW00029170	Alexandra	Adams	12/3/11	3953.9884

Example 13: Summarize the dataset from Example 12 so we can see one record per territory. We want to return the below items:

1. How many sales transactions were recorded? TIP: use COUNT()
2. How many unique customers made a purchase? TIP: use COUNT(DISTINCT)
3. How much was sold? TIP: use SUM()
4. What was the smallest purchase? TIP: use MIN()
5. What was the largest purchase? TIP: use MAX()
6. What was the average purchase? TIP: use AVG()

Tables Used: Same from Example 12.

```
SELECT
D.Name as [Territory Name],
COUNT(*) as [Sales Transactions],
COUNT(DISTINCT A.CustomerID) as [Unique Customers],
SUM(C.TotalDue) as [Sales Amount],
MIN(C.TotalDue) as [Smallest Transaction],
MAX(C.TotalDue) as [Largest Transaction],
AVG(C.TotalDue) as [Average Transaction]

FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID INNER JOIN
 AdventureWorks2019.Sales.SalesOrderHeader C
  ON A.CustomerID = C.CustomerID INNER JOIN
AdventureWorks2019.Sales.SalesTerritory D
 ON C.TerritoryID = D.TerritoryID

WHERE
C.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

GROUP BY
D.Name -- TIP: I always just paste everything before my aggregate functions here (but
remove the Alias Names)
```

Results (Snapshot Only): You'll see only 10 records total and one line per territory (down from 1,607). The benefit of doing this is obvious – it allows you to bypass exporting a ton of detailed records to Excel then performing a pivot table on the detail to achieve the same outcome (faster and repeatable).

Territory Name	Sales Transactions	Unique Customers	Sales Amount	Smallest Transaction	Largest Transaction	Average Transaction
Germany	81	81	272780.9066	772.5036	10/27/10	3367.6655
Australia	463	463	1693032.742	772.5036	10/27/10	3656.6581
United Kingdom	117	117	400991.929	772.5036	10/27/10	3427.2814
Northeast	44	22	705672.1952	6.3484	11/7/84	16038.0044
Canada	149	106	2106905.873	64.242	6/8/74	14140.3078
Southwest	339	293	3144713.099	12.6968	5/9/21	9276.4398

Filtering Using Having

In a prior section, we described using the WHERE clause to apply filters. We've also seen filtering in the FROM statement. Now, we'll focus on filtering in the HAVING clause.

Filtering in the HAVING clause exclusively applies to aggregation. You would never use HAVING without aggregation. It is very similar to filtering in the WHERE statement with one BIG exception. This filter is applied at the ***very end*** of the query. You've already used filtering in the WHERE and FROM statements to return the detailed records that you want to be considered within the scope of your aggregation. Now, you want to filter on the ***results of the aggregation***.

Example 14: Summarize the dataset from Example 13 so we can see one record per territory.

We only want to see territories with over 50 sales transactions and $1M in sales. We want to display the same fields from Example 13.

Tables Used: Same from Example 12.

```
SELECT
D.Name as [Territory Name],
COUNT(*) as [Sales Transactions],
COUNT(DISTINCT A.CustomerID) as [Unique Customers],
SUM(C.TotalDue) as [Sales Amount],
MIN(C.TotalDue) as [Smallest Transaction],
MAX(C.TotalDue) as [Largest Transaction],
AVG(C.TotalDue) as [Average Transaction]


FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID INNER JOIN
 AdventureWorks2019.Sales.SalesOrderHeader C
  ON A.CustomerID = C.CustomerID INNER JOIN
AdventureWorks2019.Sales.SalesTerritory D
 ON C.TerritoryID = D.TerritoryID

WHERE
C.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

GROUP BY
D.Name

HAVING
COUNT(*) <= 50 OR
SUM(C.TotalDue) <= 1000000
```

Results: You'll now only see five records. Playing off Example 13's query and output, you can see that some territories get dropped because they don't satisfy our HAVING conditions. Two things:

1. Notice how important paying attention to detail is. I used > 50 instead of >= 50 and we dropped Central. Had we used >= 50, then Central would have passed our test. This can be seen in the 2nd table below.
2. You'll notice that in the syntax of the HAVING statement, we use the actual functions instead of the alias names. This is consistent throughout different SQL platforms. You can refer to a function or a formula pretty much anywhere in the query. You treat the whole formula as a column instead of the alias name, no matter how long or complex it is. The ***only*** place that you can use the column alias name is in the ORDER BY because it is performed at the very end of the query.

Territory Name	Sales Transactions	Unique Customers	Sales Amount	Smallest Transaction	Largest Transaction	Average Transaction
Australia	463	463	1693032.742	772.5036	10/27/10	3656.6581
Canada	149	106	2106905.873	64.242	6/8/74	14140.3078
Southwest	339	293	3144713.099	12.6968	5/9/21	9276.4398
Northwest	224	194	2620943.826	472.3108	3/25/86	11700.642
Southeast	70	34	1847744.578	22.3061	8/19/89	26396.3511

3. The five records that dropped from the having clause are shown below (Central is record five):

Territory Name	Sales Transactions	Unique Customers	Sales Amount	Smallest Transaction	Largest Transaction	Average Transaction
Germany	81	81	272780.9066	772.5036	10/27/10	3367.6655
United Kingdom	117	117	400991.929	772.5036	10/27/10	3427.2814
Northeast	44	22	705672.1952	6.3484	11/7/84	16038.0044
France	70	70	236268.627	772.5036	10/27/10	3375.2661
Central	50	26	1126645.75	160.605	1/23/57	22532.9149

Performing Basic Math

Every possible use for data can't be pre-determined when database architects are building databases and data warehouses. Therefore, it is common to use math in queries.

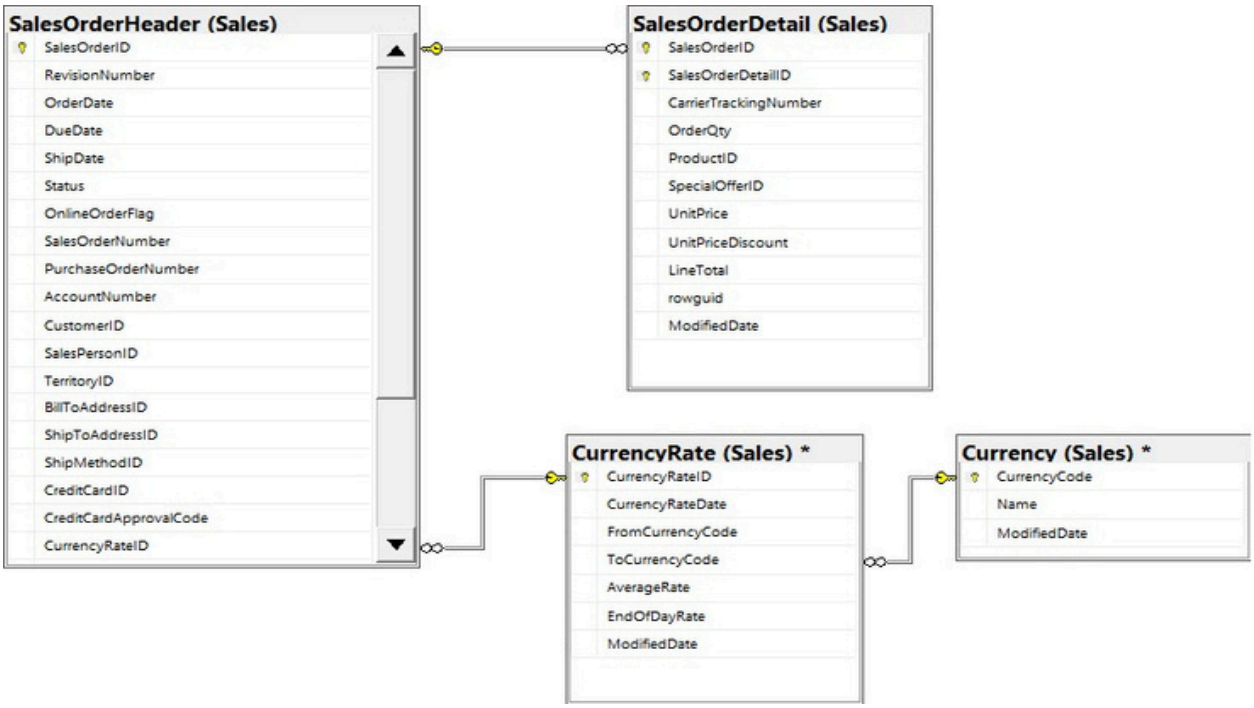
Regular math operators apply to SQL (+, -, *, /). Just make sure that you are performing math on numerical data types (floats or integers). For example, you can't take a string that looks like a number and add it to an actual number. The field will have to be converted to a number first.

When you use math or functions, it is very important that you are using the actual fields and formulas (NOT the alias names). No matter how complex a field is, you can still use it in a formula (see field [Currency Equiv/Non US] in Example 15 below).

Example 15: Calculate the total order price, currency adjusted total order price for non-US countries, and total line price using the SalesOrderHeader and SalesOrderDetail tables.

Tables Used: SalesOrderHeader will be used again along with a few others.

1. SalesOrderHeader table – used to track details about the entire order. For example, shipping details, payment details, etc.
2. SalesOrderDetail table – used to track details about the specific purchases within the entire order. For example, quantities, unit price, product(s) sold, etc.
3. CurrencyRate table – used to track the specific exchange rates for each currency.
4. Currency table – used to track the currency code name.



Tips:

1. Even though SQL Server does a pretty good job with basic order of operations, I highly recommend using parentheses to tell the query what order to process the math (see [Currency Equiv/Non US] and [Total Line Price] for examples).
2. Notice the use of LEFT JOINs. If those were instead INNER JOINs, then any record with a NULL [Currency Rate ID] would be eliminated from the results.

```
SELECT
A.SalesOrderID AS [Sales Order ID],
A.SubTotal + A.TaxAmt + A.Freight AS [Total Order Price],
A.CurrencyRateID AS [Currency Rate ID],
(A.SubTotal + A.TaxAmt + A.Freight) * C.AverageRate AS [Currency Equiv/Non US],
C.AverageRate AS [Currency Rate],
D.Name AS [Currency Name],
(B.OrderQty * B.UnitPrice) - (B.OrderQty * B.UnitPriceDiscount) AS [Total Line Price]

FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
ON A.SalesOrderID=B.SalesOrderID LEFT JOIN
AdventureWorks2019.Sales.CurrencyRate C
 ON A.CurrencyRateID=C.CurrencyRateID LEFT JOIN
 AdventureWorks2019.Sales.Currency D
  ON D.CurrencyCode=C.ToCurrencyCode
```

3. The same mathematical columns can be used in your WHERE statement - if you need to filter your results on the math outcome (remember you must include the entire field, not the ALIAS NAME like below).

```
WHERE
A.SubTotal + A.TaxAmt + A.Freight > 100000
```

Results (snapshot only): You will see 121,317 records. The below results are NOT the first six records of the output nor are they complete orders (I scrolled down until I could find something where the Currency Rate was populated.

Sales Order ID	Total Order Price	Currency Rate ID	Currency Equiv/Non US	Currency Rate	Currency Name	Total Line Price
43659	23153.2339	NULL	NULL	NULL	NULL	80.746
43660	1457.3288	NULL	NULL	NULL	NULL	419.4589
43660	1457.3288	NULL	NULL	NULL	NULL	874.794
43661	36865.8012	4	53975.2195	1.4641	Canadian Dollar	809.76
43661	36865.8012	4	53975.2195	1.4641	Canadian Dollar	714.7043
43661	36865.8012	4	53975.2195	1.4641	Canadian Dollar	1429.4086

Using Common Functions

Just like we described in the prior section, we often have the need to use functions to customize our results. Functions are pre-determined calculations/actions that you can leverage to simplify your query. It would be impossible for us to list them all in this book, but a simple internet search will return thousands of possible functions. You can even write your own custom functions on most SQL platforms.

Common function definitions are included below:

Function Name	Description	Why is it Important?
CAST()	Used to convert data types (ex. string to number).	Sometimes, you need to use strings that look like numbers in math formulas. You are not allowed to mix data types. Therefore, we convert it to a number first.
CONCAT()	Used to connect two strings into one longer string.	If you want to merge the first name with the last name into one field, you would use concat.
REPLACE()	Used to replace a character(s) with another character(s).	If you want to clean a field that may have two spaces, you could replace two spaces with one.
RTRIM()	Used to remove trailing spaces from the end of a string.	Often, strings values carry extra blank spaces after the text. It is best practice to eliminate those spaces.
LTRIM()	Used to remove leading spaces from the start of a string.	Same as RTRIM(), but reversed.
SUBSTRING()	Used to extract a specified # of characters from a string.	If you have a lengthy text field and you want to return a subset of text, you can use this function to strip out all unwanted text.
ROUND()	Rounds a decimal to a specified precision.	Allows you to return a cleaner value in your results.
FLOOR()	Rounds a decimal down.	Sometimes you just need the base of a number and don't want to round it.
CEILING()	Rounds a decimal up.	Sometimes you need to round up for various business reasons.
RIGHT()	Returns a specified # of characters from right to left.	It may be used to return the last 4 of a phone number or credit card number to protect your customer's private information.
LEFT()	Returns a specified # of characters from left to right.	It may be used to return the area code of a phone number to identify the location of the caller.
UPPER()	Converts lowercase characters to upper case.	May be used to standardize strings because lower case or proper case may not align with other data sources.

Example 16: Build a query using each of the functions described above.

Tables Used: Same as Example 15.

Tips:

1. To compare the original field values to the results, run a query without the functions.
2. The [Cast Example] looks like a number, but it's a string.
3. The [Replace Example] takes a few inputs (separated by a comma):
    a. The string field that you want to clean.
    b. The character(s) that you want to replace.
    c. The character(s) that you want to replace with.
4. The [Substring Example] also takes a few inputs (separated by a comma):
    a. The string field that you want to extract from.
    b. The string starting position that you want to begin extracting from.
    c. The number of characters that you want to extract.

```
SELECT CAST(A.Status AS NVARCHAR) AS [Cast Example],
CONCAT(D.CurrencyCode,D.Name) AS [Concat Example],
D.CurrencyCode + D.Name AS [Concat Example 2],
REPLACE(A.AccountNumber,'-','') AS [Replace Example],
RTRIM(A.PurchaseOrderNumber) AS [Rtrim Example],
LTRIM(A.PurchaseOrderNumber) AS [Ltrim Example],
SUBSTRING(A.AccountNumber,9,6) AS [Substring Example],
ROUND(A.TotalDue,2) AS [Round Example],
FLOOR(A.TotalDue) AS [Floor Example],
CEILING(A.TotalDue) AS [Ceiling Example],
RIGHT(A.AccountNumber,4) as [Right Example],
LEFT(A.AccountNumber,4) as [Left Example], UPPER(D.Name)
AS [Upper Example]

FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
ON A.SalesOrderID=B.SalesOrderID LEFT JOIN
AdventureWorks2019.Sales.CurrencyRate C
 ON A.CurrencyRateID=C.CurrencyRateID LEFT JOIN
 AdventureWorks2019.Sales.Currency D
  ON D.CurrencyCode=C.ToCurrencyCode
```

Results (snapshot only): You'll see 121,317 records with each of our functions applied.

Cast Example	Concat Example	Concat Example 2	Replace Example	Rtrim Example	Ltrim Example
5		NULL	104020000117	PO18850127500	PO18850127500
5		NULL	104020000117	PO18850127500	PO18850127500
5	CADCanadian Dollar	CADCanadian Dollar	104020000442	PO18473189620	PO18473189620
5	CADCanadian Dollar	CADCanadian Dollar	104020000442	PO18473189620	PO18473189620
5	CADCanadian Dollar	CADCanadian Dollar	104020000442	PO18473189620	PO18473189620
5	CADCanadian Dollar	CADCanadian Dollar	104020000442	PO18473189620	PO18473189620

Continued….

Substring Example	Round Example	Floor Example	Ceiling Example	Right Example	Left Example	Upper Example
117	1457.33	1457	1458	117	10/4/23	NULL
117	1457.33	1457	1458	117	10/4/23	NULL
442	36865.8	36865	36866	442	10/4/23	CANADIAN DOLLAR
442	36865.8	36865	36866	442	10/4/23	CANADIAN DOLLAR
442	36865.8	36865	36866	442	10/4/23	CANADIAN DOLLAR
442	36865.8	36865	36866	442	10/4/23	CANADIAN DOLLAR

Case Statement

The case statement is very easy if you are familiar with 'IF' statements in Excel. It's an alternative with similar syntax. I use it in most queries.

The case statement can be used in the SELECT, FROM, WHERE, GROUP BY and HAVING statements. The entire case statement should be treated as a field when used anywhere in the query. As always, don't try to use the column alias name from the case statement.

Syntax:

_____

CASE
WHEN test condition THEN outcome
WHEN test condition 2 THEN outcome 2
ELSE outcome for all remaining conditions
END

Example 17: Determine how many people from the Person table have a last name that starts with A through M compared to N through Z.

Table Used: The Person table is used again.

```
SELECT
CASE

 WHEN   SUBSTRING(A.LastName,1,1)BETWEEN  'A'AND  'M'THEN 'First Half'
 WHEN   SUBSTRING(A.LastName,1,1)BETWEEN 'N' AND 'Z' THEN 'Second Half'
 ELSE 'Unknown'
END AS [Placement in Alphabet],
COUNT(*) AS [People Count]

FROM
AdventureWorks2019.Person.Person A

GROUP  BY  CASE
WHENSUBSTRING
WHENSUBSTRING  (A.LastName,1,1) BETWEEN  'A'AND  'M'THEN 'First Half'
               (A.LastName,1,1) BETWEEN 'N' AND 'Z' THEN 'Second Half'
 ELSE 'Unknown'
END
```

Results: You'll see two records. In this case, we blended the case statement with aggregation, so you can start to see how things are coming together.

Placement in Alphabet	People Count
First Half	11082
Second Half	8890

Date Functions

Another specific type of function that frequently comes in handy is date functions. This is critical to being able to automate your queries. You can read the current date on the server then add years, months, days, weeks to the date to dynamically calculate start and end dates based on the current date. This can be used anywhere in the query, but we usually apply automated look backs in the WHERE statement.

We'll cover three date functions in this book: DATEPART(), DATEADD() and DATEDIFF(). To start, we'll pass the current date into the DATEPART() and DATEADD() formulas. The current date is identified using the getdate() function. You can replace getdate() with any field name from your database.

DATEPART() Syntax:

```
SELECT
DATEPART(M,getdate()) as Month, -- Extracts the month from date
DATEPART(YYYY,getdate()) as Year, -- Extracts the year from date
DATEPART(D,getdate()) as [Day of Month], -- Extracts the day of month from date
DATEPART(DW,getdate()) as [Day of Week] -- Extracts the day of week from date
```

DATEADD() Syntax:

```
SELECT
DATEADD(M,-6,getdate()) as Months, -- Subtracts 6 month from date
DATEADD(YYYY,1,getdate()) as Years, -- Adds 1 year to date
DATEADD(D,-30,getdate()) as Days -- Subtracts 30 days from date
```

DATEDIFF() Syntax: Just replace starting_date/ending_date with the dates you want to use:

```
DATEDIFF(M,starting_date,ending_date) -- Calculates the months between two dates
DATEDIFF(YYYY,starting_date,ending_date) -- Calculates the years between two dates
DATEDIFF(D,starting_date,ending_date) -- Calculates the days between two dates
```

Example 18: Display the first day and last day of the prior month and the first day and last day of the current year. Since we aren't using any fields from the database, we don't need a FROM statement. It's rare to do this – I typically only do this when I'm testing date formulas.

Tables Used: No tables used in this example.

Tips:

1. This result will change every day because it is based on the getdate() function which is the current date that the query is running.
2. The green highlight is included to show you that you can embed date functions within other date functions.

Results: You'll see one record.

Start of Prior Month	End of Prior Month	Start of Current Year	End of Current Year
2/1/23	2/28/23	1/1/23	3/4/23

Subqueries

Sometimes, you need to embed queries within queries. This phenomenon is called a subquery. It can be used in the FROM and WHERE statements. In rare cases, it can be found in the SELECT and the HAVING. In this book, we'll focus on the FROM and WHERE.

To keep this simple, a subquery creates a dynamic result set that should be treated just like a table. The benefit of using a subquery is that you can use the data in your database to apply filtering to a query or add additional information. This is particularly useful if you have a constantly changing database. This makes maintenance significantly easier. For example, you won't have to maintain a hardcoded list of values in your WHERE statement. Instead, a subquery will keep filtering dynamic.

Another use case for a subquery is if you want to mix both detailed and aggregate data together. For example, using our previous territory examples – perhaps we want to understand the sales distribution across each territory. In other words, what % of total sales is each territory responsible for. We could use a subquery to pull in the total sales across all territories (the denominator), then divide the sales for each territory (the numerator) by the denominator.

The syntax for a subquery is the same as a regular query (ie. everything we've been discussing thus far). The difference is you wrap it in parentheses then insert it into the appropriate statement within your query and treat it like a table. We'll show an example of both FROM and WHERE.

Example 19: Consider the below query and dataset for your examples (taken from Example 12).

Tables Used: Same as Example 12.

```
SELECT
D.Name as [Territory Name],
A.CustomerID as [Customer ID],
A.AccountNumber as [Account Number],
B.FirstName as [First Name],
B.LastName as [Last Name],
C.OrderDate as [Order Date],
C.TotalDue as [Total Due]



FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID INNER JOIN
 AdventureWorks2019.Sales.SalesOrderHeader C
  ON A.CustomerID = C.CustomerID INNER JOIN
AdventureWorks2019.Sales.SalesTerritory D
 ON C.TerritoryID = D.TerritoryID

WHERE
C.OrderDate BETWEEN '2011-01-01' and '2011-12-31'
```

Results (Snapshot Only): You'll still see 1,607 records that look like the below sample. As you can see, there are multiple records for each customer.

Territory Name	Customer ID	Account Number	First Name	Last Name	Order Date	Total Due
Central	29486	AW00029486	Kim	Abercrombie	10/31/11	75865.1515
Northeast	29487	AW00029487	Humberto	Acevedo	10/31/11	19265.5486
Southeast	29484	AW00029484	Gustavo	Achong	8/1/11	4560.2864
Southwest	29170	AW00029170	Alexandra	Adams	12/3/11	3953.9884
Northwest	28678	AW00028678	Ben	Adams	10/24/11	3953.9884
Southeast	29491	AW00029491	Carla	Adams	12/1/11	13771.7748

Example 20: We want to use the above dataset, but instead of seeing each transaction for each customer, return only the last transaction for each customer.

Tables Used: Same as Example 12.

Tips:

1. Create a subquery to grab the latest OrderDate for each CustomerID.
2. Give the MAX(OrderDate) an alias name of MaxDt.
3. If you run the subquery separately, you will get two columns that look like the sample below – one row per CustomerID.

CustomerID	MaxDt
11000	6/21/11
11001	6/17/11
11002	6/9/11
11003	5/31/11
11004	6/25/11
11005	6/1/11

4. You can see that the results look like a table. Therefore, we can treat it like one and join our main query to this dynamic result set to filter out any OrderDates that took place prior to the MAX(OrderDate) – also known as our MaxDt after we gave our second column an alias name.

```
SELECT
D.Name as [Territory Name],
A.CustomerID as [Customer ID],
A.AccountNumber as [Account Number],
B.FirstName as [First Name],
B.LastName as [Last Name],
C.OrderDate as [Order Date],
C.TotalDue as [Total Due]

FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID INNER JOIN
 AdventureWorks2019.Sales.SalesOrderHeader C
  ON A.CustomerID = C.CustomerID INNER JOIN
AdventureWorks2019.Sales.SalesTerritory D
 ON C.TerritoryID = D.TerritoryID INNER JOIN
(
SELECT
F.CustomerID,
MAX(F.OrderDate) as MaxDt

FROM
AdventureWorks2019.Sales.SalesOrderHeader F

WHERE
F.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

GROUPBY

FON C.CustomerID = E.CustomerID
 AND C.OrderDate = E.MaxDt

WHERE

C.OrderDate BETWEEN '2011-01-01' and '2011-12-31'
```

Results (Snapshot Only): The results are trimmed from 1,607 to 1,406 and you now see one record per customer. The green highlight is really the new concept (the subquery). The yellow highlight shows you that the rest of this process is no different than how we treat a table.

Territory Name	Customer ID	Account Number	First Name	Last Name	Order Date	Total Due
Southwest	29757	AW00029757	Garth	Fort	10/1/11	29689.3502
Central	29668	AW00029668	Connie	Coffman	10/1/11	472.3108
Northwest	29611	AW00029611	Jared	Bustamante	10/1/11	3398.5212
Canada	29620	AW00029620	Joan	Campbell	10/1/11	9215.6816
Canada	30058	AW00030058	Mike	Taylor	10/1/11	4560.2864
Canada	29548	AW00029548	Ann	Beebe	10/1/11	680.0689

Example 21: Use the same dataset that we used in Example 14, but now we want to show all detailed transactions that took place on the busiest sales day of the year.

Tables Used: Same as Example 12.

Tips:

1. The TOP 1 condition does exactly what it says. It only returns the first record in your result set. In this case, it would return the first OrderDate.
2. The ORDER BY is necessary when using the TOP 1 because you need to dictate which TOP 1 record you want to select. For example, if you don't include an ORDER BY, then your TOP 1 clause will return one random record. On the other hand, if your ORDER BY is in ascending order, then you will receive the earliest OrderDate.
3. You don't always have to display the field that you are sorting on. When using a subquery in your WHERE statement, it is intended to create one or more values to use as a filter (like an 'In' filter from our 'Common Operators to Use in Filter' section). Therefore, you don't typically display more than one column in a subquery used in the WHERE statement.

```
SELECT
D.Name as [Territory Name],
A.CustomerID as [Customer ID],
A.AccountNumber as [Account Number],
B.FirstName as [First Name],
B.LastName as [Last Name],
C.OrderDate as [Order Date],
C.TotalDue as [Total Due]

FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.PersonID = B.BusinessEntityID INNER JOIN
 AdventureWorks2019.Sales.SalesOrderHeader C
  ON A.CustomerID = C.CustomerID INNER JOIN
AdventureWorks2019.Sales.SalesTerritory D
 ON C.TerritoryID = D.TerritoryID

WHERE
C.OrderDate IN
(
SELECT TOP 1 F.OrderDate

FROM
AdventureWorks2019.Sales.SalesOrderHeader F

WHERE
F.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

GROUP BY
F.OrderDate

ORDER BY COUNT(*) DESC)
```

Results (Snapshot Only): You'll see 96 detailed transactions; all of which were made on 10/1/2011.

Territory Name	Customer ID	Account Number	First Name	Last Name	Order Date	Total Due
Southwest	29757	AW00029757	Garth	Fort	10/1/11	29689.3502
Central	29668	AW00029668	Connie	Coffman	10/1/11	472.3108
Northwest	29611	AW00029611	Jared	Bustamante	10/1/11	3398.5212
Canada	29620	AW00029620	Joan	Campbell	10/1/11	9215.6816
Canada	30058	AW00030058	Mike	Taylor	10/1/11	4560.2864
Canada	29548	AW00029548	Ann	Beebe	10/1/11	680.0689

Union & Union All

Part of data analytics includes blending datasets from different data sources and producing a unified solution. The way to do this using SQL is through a UNION statement. A UNION statement combines two different datasets – if they each produce three common things:

1. The same number of columns.
2. The same order of columns.
3. The same data types for each column.

The way we do this is by writing two select statements and combining them with the word UNION between them. For example, select statement1 UNION select statement2.

It is important to understand the difference between UNION and UNION ALL. A ***UNION*** will include only unique records between the first and second select statements. A ***UNION ALL*** will include all records produced in the first select statement along with all records produced in the second select statement, even if the same record exists in both queries.

Why would I use a UNION ALL? Sometimes when we pull data from two different data sources, the same records exist in each data source. Depending on the business need, we may need to show both occurrences instead of just the unique instance. This is outlined in our next two examples.

Example 22: Provide a list of unique first and last names for customers and employees. If the same name exists more than one time, we only want to see it once.

Tables Used: The Customer and Person tables will be used again.

```
SELECT
B.FirstName,
B.LastName



FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
ON A.PersonID=B.BusinessEntityID

UNION

SELECT
B.FirstName,
B.LastName

FROM
AdventureWorks2019.HumanResources.Employee A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.BusinessEntityID=B.BusinessEntityID
```

Results (Snapshot Only): You will see 19,188 records. There won't be any duplication of first and last names in this result set.

FirstName	LastName
A.	Leonetti
A. Scott	Wright
Aaron	Adams
Aaron	Alexander
Aaron	Allen
Aaron	Baker

Example 23: Provide a list of all first and last names for customers and employees. If the same name exists more than once, we want to see each occurrence to ensure we don't under count.

Tables Used: The Customer and Person tables will be used again.

```
SELECT
B.FirstName,
B.LastName


FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
ON A.PersonID=B.BusinessEntityID

UNION ALL

SELECT
B.FirstName,
B.LastName

FROM
AdventureWorks2019.HumanResources.Employee A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.BusinessEntityID=B.BusinessEntityID
```

Results (Snapshot Only): You will see 19,409 records. The reason that you get 221 more records using UNION ALL is because there are 215 names that appear more than one time across customers and employees. In the next section, we will discuss Common Table Expressions where you will see the list of names that appear more than once.

FirstName	LastName
Jennifer	Adams
Jeremy	Adams
Jesse	Adams
Jonathan	Adams
Jordan	Adams
Jordan	Adams

Common Table Expressions (CTEs)

A Common Table Expression, also known as a CTE or WITH statement, is when you define one or more named temporary datasets that can be referenced and used later in your query. They are a nice alternative to a subquery because they are much easier to read.

The basic structure of a CTE is: WITH query1 as (select statement), query2 as (select statement) select * from query1 a inner join query2 b on a.field=b.field

In this basic example, our two named datasets are query1 and query2. The two datasets are treated exactly like a table then joined together further downstream to get a larger dataset. There isn't a limit on how many CTEs that you can use. As you build more complex queries, it is common to use five or even ten CTEs.

Example 24: Using Example 23, rewrite the code using CTEs then list the names appearing more than once.

Tables Used: The Customer and Person tables will be used again.

```
WITH CustomerNames AS -- First named dataset ( SELECT B.FirstName as [First Name], -- Use the alias names when referring to a named dataset (aka a CTE) B.LastName as [Last Name]
```

```
FROM
AdventureWorks2019.Sales.Customer A INNER JOIN
AdventureWorks2019.Person.Person B
ON A.PersonID=B.BusinessEntityID
),

EmployeeNames AS -- Second named dataset
(
SELECT
B.FirstName as [First Name],
B.LastName as [Last Name]

FROM
AdventureWorks2019.HumanResources.Employee A INNER JOIN
AdventureWorks2019.Person.Person B
 ON A.BusinessEntityID=B.BusinessEntityID
),
```

```
MergedNames AS -- Combine first and second named data sets
(
SELECT
A.[First Name],
A.[Last Name]

FROM
CustomerNames A

UNION ALL -- Using union all to ensure we don't eliminate duplications

SELECT
A.[First Name],
A.[Last Name]

FROM
EmployeeNames A
)

SELECT
A.[First Name],
A.[Last Name],
COUNT(*) as Occurrences

FROM
MergedNames A

GROUP BY
A.[First Name],
A.[Last Name]

HAVING -- Having clause allows us to only those names with > 1 occurrence
COUNT(*) > 1
```

Results (Snapshot Only): You will see 215 records, most of which show two occurrences each. Refer to the comments in the query to explain what each component is doing.

First Name	Last Name	Occurrences
Kim	Abercrombie	2
Pilar	Ackerman	2
Jay	Adams	2
Jordan	Adams	2
Amy	Alberts	2
Jordan	Allen	2

Duplication Checking

One of the fastest ways to lose credibility as an analyst is to produce a report that contains duplicate records. The most common way to introduce duplicates is through table joins. If you don't have your table joins set up properly, your records will duplicate.

The easiest way to avoid duplicates is to check for them at each step of writing your query.

Follow the steps in the next example as best practice.

Example 25: Create a list of customer names who placed an order in 2011.

Tables Used: The SalesOrderHeader table will be used again.

Step 1: Identify the customers with sales made in 2011.

```
SELECT
A.CustomerID



FROM
AdventureWorks2019.Sales.SalesOrderHeader A

WHERE
A.OrderDate BETWEEN '2011-01-01' and '2011-12-31'
```

Step 2: Determine if any of the customer IDs appear with more than one sale using basic aggregation.

```
SELECT
A.CustomerID,
COUNT(*) as Occurrences


FROM
AdventureWorks2019.Sales.SalesOrderHeader A

WHERE
A.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

GROUP BY
A.CustomerID

HAVING
COUNT(*)>1
```

Step 3: If you see any duplications, remove them from your query using a subquery or CTE and make this the starting point of the final query.

Using a subquery:

```
SELECT
A.CustomerID,
COUNT(*) as Occurrences


FROM
AdventureWorks2019.Sales.SalesOrderHeader A

WHERE
A.OrderDate BETWEEN '2011-01-01' and '2011-12-31'

GROUP BY
A.CustomerID

HAVING
COUNT(*)>1
```

Using a CTE:

```
WITH
UniqueCustomers AS
(
SELECT DISTINCT -- Add distinct to make your list of Customer IDs unique
A.CustomerID


FROM
AdventureWorks2019.Sales.SalesOrderHeader A

WHERE
A.OrderDate BETWEEN '2011-01-01' and '2011-12-31'
)

SELECT *

FROM
UniqueCustomers
```

Step 4: Repeat steps 1 – 3 for any additional tables that you bring in, then build your final query. We'll use the CTE for the final output.

Tip:

1. You should also ensure that you don't see an unexpected reduction in records from one step to the next step. We had 1,406 records in Step 3 and our final dataset shows 1,406 records.

```
WITH
UniqueCustomers AS
(
SELECT DISTINCT -- Add distinct to make your list of Customer IDs unique
A.CustomerID

FROM
AdventureWorks2019.Sales.SalesOrderHeader A

WHERE
A.OrderDate BETWEEN '2011-01-01' and '2011-12-31'
)

SELECT
A.CustomerID,
C.FirstName,
C.LastName

FROM
UniqueCustomers A INNER JOIN
AdventureWorks2019.Sales.Customer B
 ON A.CustomerID = B.CustomerID INNER JOIN
AdventureWorks2019.Person.Person C
 ON B.PersonID = C.BusinessEntityID
```

Results (Snapshot Only): You will see 1,406 records with no duplication on the customer ID.

CustomerID	FirstName	LastName
29486	Kim	Abercrombie
29487	Humberto	Acevedo
29484	Gustavo	Achong
29170	Alexandra	Adams
28678	Ben	Adams
29491	Carla	Adams

Window Functions

We have looked at traditional aggregation, but there's a way of performing aggregation without reducing/collapsing your record counts. Sometimes, we have the need to consider a mix of detail and aggregation to determine things like row numbers within a defined group or % of total calculations. Window functions will help us accomplish this mix.

A window function is when you define a group (ie. your ***window***) then apply some aggregation within that defined group/window and display it amongst your detail records. Once any value changes within your defined window, it will reset for the next window.

The standard format of a window function is as follows:

AGG FUNCTION(FIELD NAME) OVER(PARTITION BY your defined window ORDER BY the order in which you wish to summarize)

Where the AGG FUNCTION is no different than what you learned in the 'Aggregation' section (ie. SUM, MIN, MAX, COUNT), but includes other functions such as:

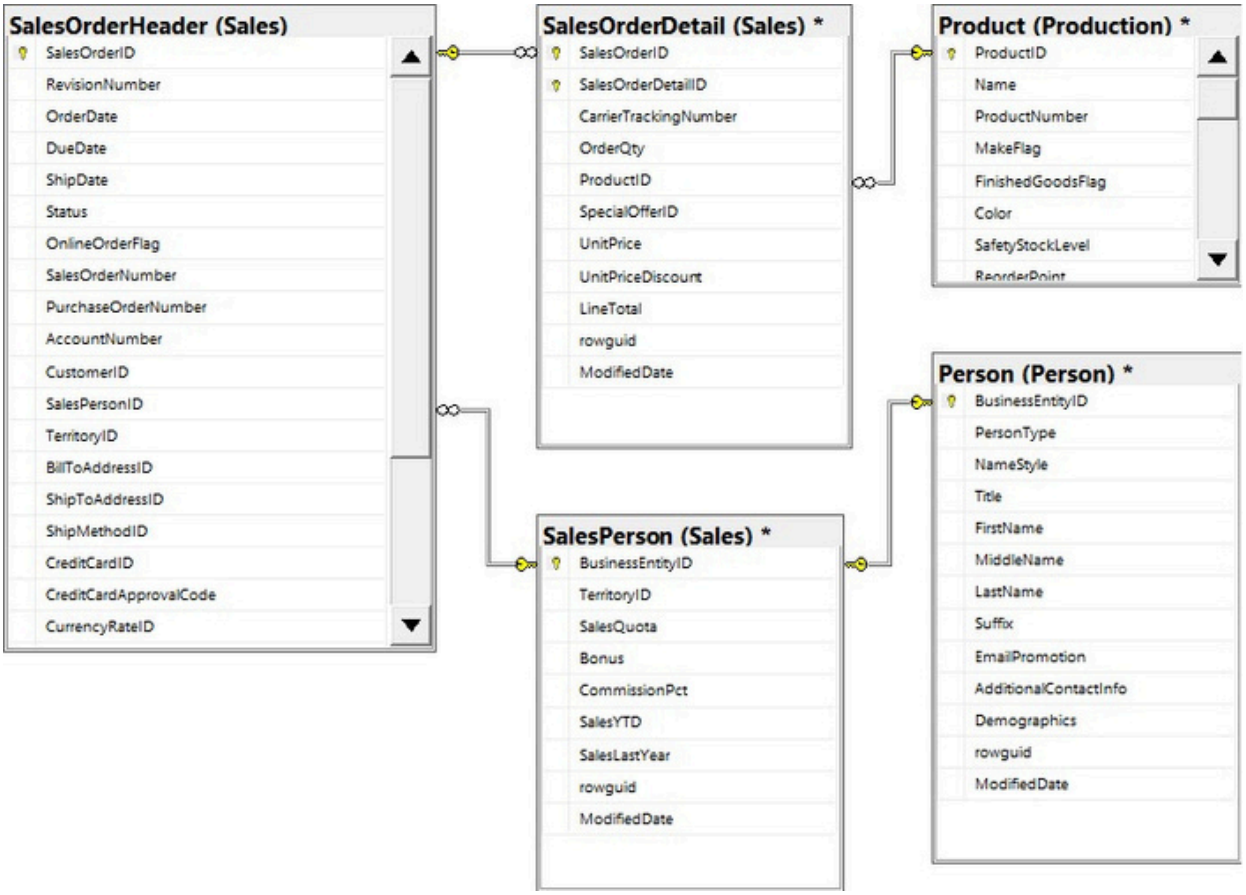
1. LAG (returns previous row's data within your window).
2. LEAD (returns next row's data within your window).
3. ROW_NUMBER (returns incremental numeric sequence within your window).

Example 26: For each salesperson, show the total sales for each product and include the following:

1. The total sales for each product (basic aggregation using a CTE).
2. Of the total sales across all products for each salesperson, what percentage does each product account for (% of total sales using division with the SUM window function).
3. Each salesperson's product ranking by total sales (ROW_NUMBER window function).
4. The decrease in sales for each product for each salesperson (LAG window function).

Tables Used: The SalesOrderHeader, SalesOrderDetail, and Person tables will be used again along with the SalesPerson and Product tables.

1. SalesPerson table – Used to show metrics related to each salesperson such as bonus and commission percentage.
2. Product table – Used to show details about each product for sale.




```
WITH
ProductSales AS
(
SELECT
A.SalesPersonID,
D.FirstName + ' ' + D.LastName AS Salesperson, -- Merge first/last names separated by space
E.Name AS [Product Name],
SUM(B.LineTotal) as [Total Sales]


FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
 ON A.SalesOrderID=B.SalesOrderID INNER JOIN
AdventureWorks2019.Sales.SalesPerson C -- You will drop records because not all sales have a salesperson ID
 ON A.SalesPersonID=C.BusinessEntityID INNER JOIN
AdventureWorks2019.Person.Person D
```

```
 ON C.BusinessEntityID=D.BusinessEntityID INNER JOIN
AdventureWorks2019.Production.Product E
 ON B.ProductID=E.ProductID

GROUP BY
A.SalesPersonID,
D.FirstName + ' ' + D.LastName,
E.Name
)

SELECT
A.SalesPersonID,
A.Salesperson,
A.[Product Name],
A.[Total Sales],
A.[Total Sales] -- Basic division using Total Sales with SUM window function
/
SUM(A.[Total Sales])
OVER(
PARTITION BY -- We want to summarize by SalesPersonID so we include this in the
PARTITION BY
A.SalesPersonID -- Don't use ORDER BY because it doesn't matter with SUM, COUNT, AVG,
MIN, MAX
) AS [% of Total],

ROW_NUMBER()
OVER(
PARTITION BY -- We want to summarize by SalesPersonID so we include this in the
PARTITION BY
A.SalesPersonID
ORDER BY -- Use ORDER BY because the order of this window is important
A.[Total Sales] DESC
) AS [Product Rank],

LAG(A.[Total Sales])
OVER(
PARTITION BY -- We want to summarize by SalesPersonID so we include this in the
PARTITION BY
A.SalesPersonID
ORDER BY -- Use ORDER BY because the order of this window is important
A.[Total Sales] DESC
)
-
A.[Total Sales] AS [Change From Prior] -- Basic subtraction using Total sales with LAG
window function

FROM
ProductSales A
```

Results (Snapshot Only): You will see 3,590 records. You should see the Product Rank reset at each SalesPersonID and the total of the % of Total should add up to 100 for each SalesPersonID (although, rounding may impact the precision causing a slight variation from 100).

SalesPersonID	Salesperson	Product Name	Total Sales	% of Total	Product Rank	Change From Prior
274	Stephen Jiang	Mountain-200 Black, 46	39244.3286	0.035933	1	NULL
274	Stephen Jiang	Road-250 Black, 44	36388.4625	0.033318	2	2855.8661
274	Stephen Jiang	Mountain-200 Silver, 46	32711.8592	0.029952	3	3676.6033
274	Stephen Jiang	Mountain-200 Silver, 38	29281.5882	0.026811	4	3430.271
274	Stephen Jiang	Road-250 Black, 52	28848.9825	0.026415	5	432.6057
274	Stephen Jiang	Road-250 Red, 58	28220.6925	0.02584	6	628.29

Views

Sometimes when working with data, we have a need to reuse the same dataset over and over. To save time from having to re-write the same query, we can save them as views. Once saved as a view, we never have to write the query again and we can select from it (like a table).

This is important because as an analyst, you want to ensure that you have consistency across reports. Re-writing the same query over and over introduces the risk of missing steps. A view brings more stability since it's written one time and used throughout your environment.

They syntax is as follows:

```
USE [database name]
GO
CREATE VIEW [schema name].[view name]
AS
Select statement
```

The only caveat with creating a view is that you cannot use the same column name more than one time. Imagine a table with the same column name – the system would be confused and not know which one to use. The same holds true for a view

Example 27: Create a view blending the SalesOrderDetail table with the SalesOrderHeader table.

Tables Used: The SalesOrderHeader and SalesOrderDetail tables will be used again.

Tip:

___

1. To see the logic behind any of the views listed on your server, you can right click on the view name in the Object Explorer, then Script View as, then Create To, then New Query Editor Window, then the query behind the view will appear in a new query window. To edit the view, make your updates in this new query window and run the code - the changes will be reflected automatically.

```
USE AdventureWorks2019
GO

CREATE VIEW Sales.vSalesOrders -- Define your view name
AS

SELECT
A.*, -- Easier to use A.* than listing out every single field from SalesOrderHeader
B.SalesOrderDetailID,
B.CarrierTrackingNumber,
B.OrderQty,
B.ProductID,
B.SpecialOfferID,
B.UnitPrice,
B.UnitPriceDiscount,
B.LineTotal

FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
 ON A.SalesOrderID=B.SalesOrderID
```

Results: No results to show as we are creating an object, not a dataset. But, you should be able to select * from your new view without error.

```
SELECT * FROM AdventureWorks2019.Sales.vSalesOrders
```

Creating Tables

This book is not intended for a data architect or database administrator, so we won't spend too much time on creating and managing tables. However, as an analyst, you will surely have to create a table at some point – most likely as a reference.

There are two ways to create a table. You can create a table by sending the results of a query to a table or you can create a table by defining the table structure in a more traditional way.

To create a table based on query results, add INTO [database name].[schema name].[table name] after the SELECT clause (Right before the FROM statement).

To create a table based on a new definition, the syntax would be as follows:

```
USE [databasename]
GO
CREATE TABLE [schema name].[view name]
(
[Field1] [data type],
[Field2] [data type]
)
```

Note, you can get much more in depth by adding constraints and primary keys. The average analyst is not going to have to do this. Therefore, I am not including it in this book.

Example 28: Using the example from the 'Views' section, create the same merged dataset into a table.

Tables Used: The SalesOrderHeader and SalesOrderDetail tables will be used again.

```
SELECT
A.*, -- Easier to use A.* than listing out every single field from SalesOrderHeader
B.SalesOrderDetailID,
B.CarrierTrackingNumber,
B.OrderQty,
B.ProductID,
B.SpecialOfferID,
B.UnitPrice,
B.UnitPriceDiscount,
B.LineTotal


INTO AdventureWorks2019.Sales.tSalesOrders -- This line creates your new table

FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
 ON A.SalesOrderID=B.SalesOrderID
```

Results: No results to show as we are creating an object, not a dataset. But, you should be able to select * from your new table without error.

```
SELECT * FROM AdventureWorks2019.Sales.tSalesOrders
```

Example 29: Create a small reference table that lists potential new product names with release date.

Tables Used: No tables will be used.

```
USE AdventureWorks2019
GO


CREATE TABLE Production.Pipeline
(
ProductName NVARCHAR(60) NOT NULL, -- NOT NULL means this field cannot accept null
values (ie. must be populated)
ReleaseDate DATETIME NOT NULL,
rowguid uniqueidentifier ROWGUIDCOL  NOT NULL -- Auto-generated unique identifier
column
)
```

Results: No results to show as we are creating an object, not a dataset. But, you should be able to select * from your new table without error.

```
SELECT * FROM AdventureWorks2019.Production.Pipeline
```

Inserting Records

Often, we need to insert data into an existing table. From an analyst's perspective, this could be important if you have a reference table that you are using to guide your query results. If so, there is a high likelihood that you will need to append data at some point. To do so, we can write an INSERT INTO statement.

Using an INSERT INTO statement, we can either insert data manually, one by one, or we can use the output of a select statement to do so.

To do so manually, the syntax would be as follows: INSERT INTO table VALUES (a, b, c, etc.)

To do so using a query, the syntax would be as follows: INSERT INTO table SELECT statement

Just note that the output of your select statement must match the table definition that you are inserting into. In other words, you can't insert 11 columns into a 10-column table (and vice versa). The data types must also match.

Example 30: Manually insert a new product, Hoverboard with an effective date of today into the table you created in Example 29.

Tables Used: The Pipeline table that you just created will be used.

```
INSERT INTO AdventureWorks2019.Production.Pipeline


VALUES ('Hoverboard', getdate(), newid()) -- Use getdate() to fetch today's date and
newid() to generate a unique value for the rowguid field
```

Results: No results to show as we are inserting records into an object. But, you should be able to see the row you inserted by using select * from your new table without error.

```
SELECT * FROM AdventureWorks2019.Production.Pipeline
```

Example 31: Use the results of a query to insert a new product, Hoverboard 2.0 with an effective date of today + 30 days into the table you created in Example 26.

Tables Used: The Pipeline table that you just created will be used.

```
INSERT INTO AdventureWorks2019.Production.Pipeline


SELECT
ProductName + ' 2.0' as ProductName, -- Adding 2.0 to the Hoverboard entry we just
made
DATEADD(D,30,getdate()) as ReleaseDate,
NEWID() as rowguid

FROM AdventureWorks2019.Production.Pipeline
```

Results: No results to show as we are inserting records into an object. But, you should be able to see the row you inserted by using select * from your new table without error.

```
SELECT * FROM AdventureWorks2019.Production.Pipeline
```

Deleting Records

Just like inserting records, sometimes we must remove records if business rules change. To do this, we use a DELETE statement. We can either delete the entire table or delete certain rows.

The syntax for deleting all records is as follows: DELETE FROM tablename

The syntax for deleting specific records is as follows: DELETE FROM tablename WHERE filter condition

Example 32: Delete the first Hoverboard record you inserted into the project Pipeline table.

First, identify the rowguid of that specific record to ensure you delete the correct one.

Tables Used: The Pipeline table that you just created will be used.

```
DELETE


FROM
AdventureWorks2019.Production.Pipeline

WHERE
rowguid='3A8893BA-AD42-495F-8204-F788C8B1EF66' -- Note, your rowguid will be different
than my rowguid. Please copy/paste from your Pipeline table
```

Results: No results to show as we are deleting records from an object. But, you should be able to see only the Hoverboard 2.0 product remaining by using select * from your new table.

```
SELECT * FROM AdventureWorks2019.Production.Pipeline
```

Updating Records

Deleting and inserting records aren't the only changes that we need to make to tables. Sometimes, we need to update records as well.

The syntax to update records is as follows: UPDATE tablename SET fieldname = something WHERE filter condition

Example 33: Update Hoverboard 2.0 to Hoverboard 3.0 in the Pipeline table. First, identify the rowguid of that specific record to ensure you update the correct one.

Tables Used: The Pipeline table that you just created will be used.

```
UPDATE AdventureWorks2019.Production.Pipeline


SET
	ProductName='Hoverboard 3.0'

WHERE
	rowguid='F32BB3FA-482B-460B-883F-1D21FB7F9680' -- Note, your rowguid will be
different than my rowguid. Please copy/paste from your Pipeline table
```

Results: No results to show as we are updating a record in an object. But, you should be able to see only the Hoverboard 3.0 product by using select * from your new table without error.

```
SELECT * FROM AdventureWorks2019.Production.Pipeline
```

Grand Finale

That concludes each of the lessons in this book. We've covered many topics, so it seems appropriate that we build one final, comprehensive query. The final query will include everything we've learned.

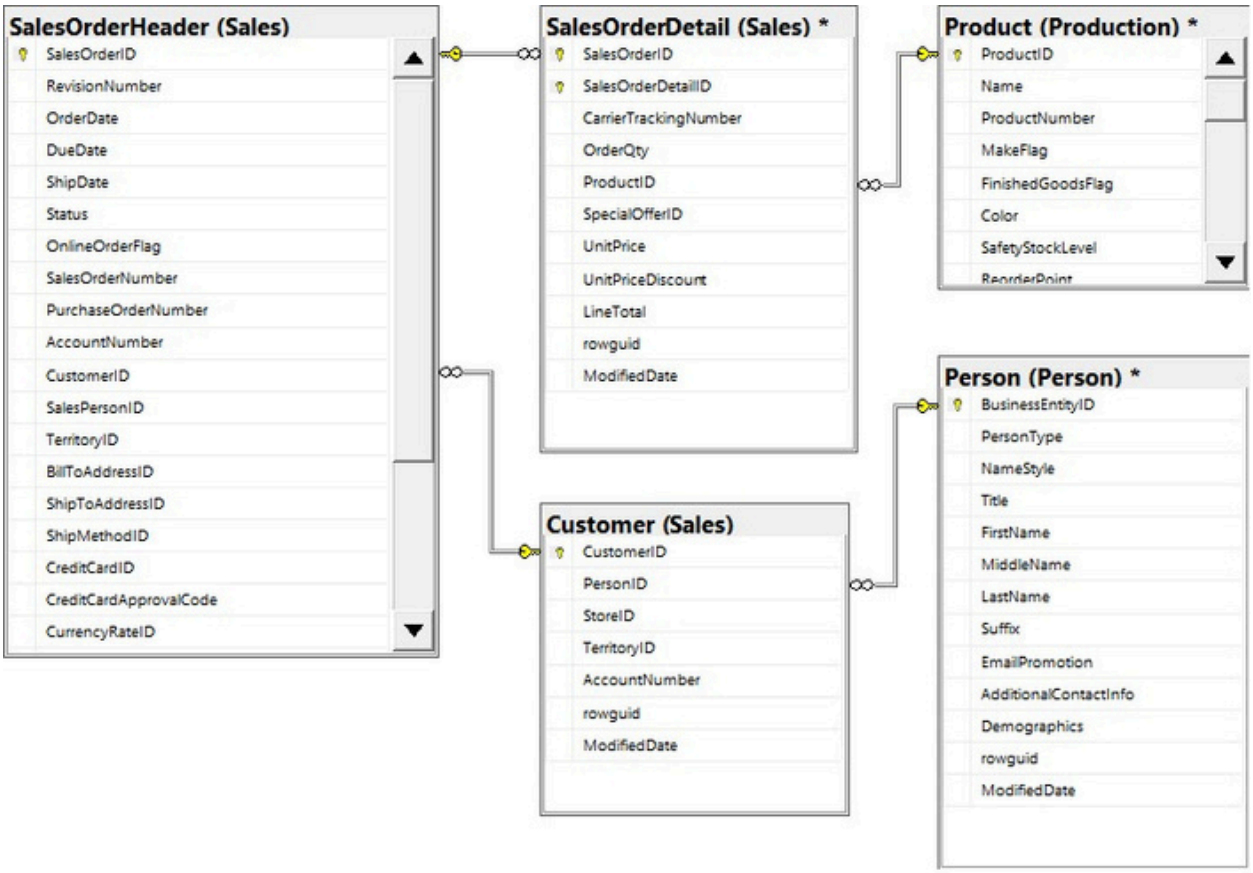
Example 34: For each of our customers, show the following:

1. What is the first product purchased by each customer along with date of first purchase? –

   Hint – use a CTE, subquery and aggregation.
2. Combine the first name with the last name, separated by a space. Hint – use '+' to concatenate (combine) the first name with the last name. You may also use the concat() function.
3. How many sales transactions did each customer have during their first year and what was the total sales during the same time period? Hint – use another CTE.
4. How many sales transactions were over $1,500 during each customer's first year? Hint – use a case statement within a SUM() function.
5. What percentage of sales transactions were over $1,500 during each customer's first year? Hint – use a window function and round to two decimals.
6. Restrict to the customers with only 3 or more transactions over $1,500. Hint – use your HAVING.
7. Adjust for currency where appropriate. Hint – use a LEFT JOIN to the Currency reference table.

Tables Used: The SalesOrderHeader, SalesOrderDetail, Product, and Person tables will be used again along with the customer table.

1. Customer table – used to track details about each customer. For example, store ID, territory ID, and account number.



Tips:

1. Open a blank Excel file and identify each of the columns that you want to display.
2. Map each column to its proper table and list any filter conditions that you need to consider.
3. Take the distinct list of tables and determine if there are any other related tables that you need to join.
4. Arrange the tables in your list so related tables are paired together.
5. List the join conditions that you need to connect related tables; be sure to think through INNER vs. LEFT JOIN.
6. Write out in plain terms any math formulas that you need to include.
7. If you need any subqueries, write, and execute them separately to ensure that you get the temp table/list results that you desire; then just copy and paste them into your main query in your FROM or WHERE statements (remember to wrap in parentheses).
8. Take all the above tips and build your query one piece at a time.
9. Ensure that you have no duplications by running each step separately to confirm counts.

```
WITH
FirstPurchase AS -- 1st CTE identifies first product ordered
(
SELECT
A.CustomerID,
E.FirstName + ' ' + E.LastName as [Customer Name], -- + ' ' + merges first/last name
A.OrderDate as [First Order Date], -- Subquery join restricts to 1st order dt
DATEADD(M,12,A.OrderDate) as EndOfFirstYear, -- Dateadd gives end dt of 1 year period
B.ProductID,
C.Name as [Product Name]


FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
 ON A.SalesOrderID = B.SalesOrderID LEFT JOIN
 AdventureWorks2019.Production.Product C
  ON B.ProductID = C.ProductID LEFT JOIN
AdventureWorks2019.Sales.Customer D
 ON A.CustomerID = D.CustomerID LEFT JOIN
AdventureWorks2019.Person.Person E
ON D.PersonID = E.BusinessEntityID INNER JOIN
( -- Subquery gets first order id per customer
SELECT
A.CustomerID,
MIN(B.SalesOrderDetailID) as FirstOrder -- MIN(SalesOrderDetailID) fetches first
product purchased only

FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
AdventureWorks2019.Sales.SalesOrderDetail B
 ON A.SalesOrderID = B.SalesOrderID

GROUP BY
A.CustomerID) F
 ON A.CustomerID = F.CustomerID
 AND B.SalesOrderDetailID = F.FirstOrder -- Join drops all purchases after first
product ordered per customer
```

```
),

LargeOrdersYear1 AS -- 2nd CTE gets orders made in 1st year
(
SELECT
A.CustomerID,
SUM(A.TotalDue) as [Total Sales], -- Aggregation gets total sales per customer
COUNT(DISTINCT A.SalesOrderID) as [Number of Orders], -- COUNT(DISTINCT) gets unique #
of SalesOrderIDs per customer
SUM
(
CASE -- SUM(CASE) helps w/ conditional counting
 WHEN A.TotalDue >= 1500 THEN 1
 ELSE 0
END
) AS [Number of Orders > 1500]

FROM
AdventureWorks2019.Sales.SalesOrderHeader A INNER JOIN
FirstPurchase B
 ON A.CustomerID = B.CustomerID
 AND A.OrderDate BETWEEN B.[First Order Date] AND B.EndOfFirstYear -- Join considers
orders within each customer's first year

GROUP BY
A.CustomerID

HAVING -- HAVING eliminates customers with <= 3 orders under $1500
SUM
(
CASE
 WHEN A.TotalDue >= 1500 THEN 1
 ELSE 0
END
) > 3
)

SELECT -- Final query pieces everything together
A.CustomerID,
A.[Customer Name],
A.[First Order Date],
A.[Product Name],
B.[Total Sales],
B.[Number of Orders],
B.[Number of Orders > 1500],
B.[Total Sales],
ROUND
(
B.[Total Sales]
/
SUM(B.[Total Sales])
OVER(PARTITION BY 1 -- Use PARTITION BY 1 when spanning full dataset (not a specific
group)
)
, 4
) AS [% of Total]
```

```
FROM
FirstPurchase A INNER JOIN
LargeOrdersYear1 B
 ON A.CustomerID = B.CustomerID
```

Results (snapshot only): You'll see 393 records. It is very important to understand how each of these components come together to solve a problem. Each component is simple, as we've learned throughout this book. If I can provide any advice, it would be to not overthink your approach. Keep it simple, take it one step at a time, and follow these planning steps to make development easier.

CustomerID	Customer Name	First Order Date	Product Name	Total Sales	Number of Orders	Number of Orders > 1500	Total Sales	% of Total
30030	Stefano Stefani	6/30/12	Road-650 Red, 62	37559.6527	5	5	37559.6527	0.0006
29784	Jeanie Glenn	6/30/12	Road-250 Red, 52	71124.6621	5	5	71124.6621	0.0011
30076	Diane Tibbott	7/1/11	AWC Logo Cap	147920.7787	5	5	147920.7787	0.0024
29538	Brenda Barlow	6/30/13	Touring-3000 Blue, 50	76266.9402	4	4	76266.9402	0.0012
29615	Mari Caldwell	6/30/13	Touring-1000 Yellow, 50	340396.3242	4	4	340396.3242	0.0056
29907	Karan Khanna	6/30/13	Touring-1000 Yellow, 54	127453.7862	4	4	127453.7862	0.0021

Conclusion

In closing, I just want to remind you that a query is no different than a puzzle. A 1,000-piece puzzle is no harder than a 100-piece one. You use the same few concepts and fit them together one piece at a time ultimately building a masterpiece. You can do this, don't overthink it!

What's Next?

Stay tuned for the next book in our series which will focus on data visualization. In the next book, we will start with the basics of Tableau and we'll close with advanced visualization concepts.

In the meantime, practice, practice, practice. Start using these concepts in your current position or in your free time for fun. Challenge yourself and try to do different things.

If you are ready for live training, contact us at robert.simon@datacoaching.io.

Appendix

Tool Installation and Environment Set-up

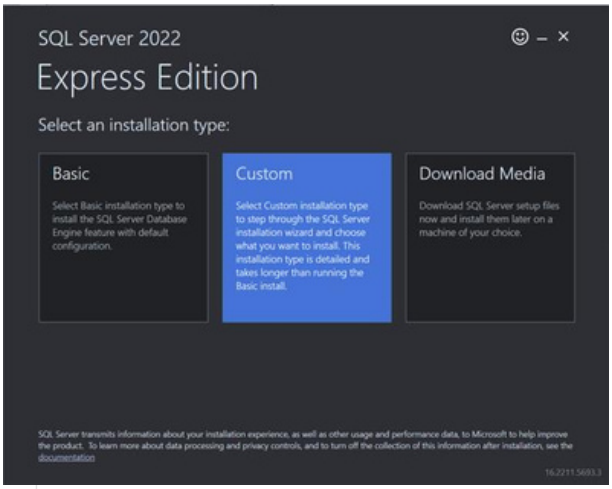
This book will be based on the AdventureWorks2019 sample database in SQL Server.

SQL Server Express Installation

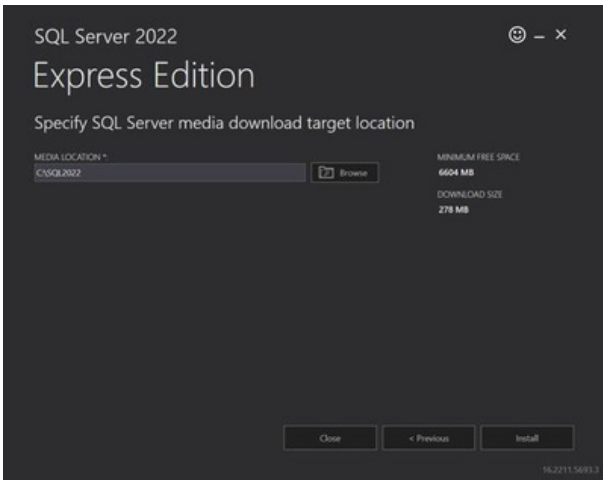
SQL Server Downloads | Microsoft

SQL Server Express is the actual database that will contain the raw data stored in tables. You will not be writing queries from the SQL Server Express application, but rather from SQL Server Management Studio. However, SQL Server Express is a requirement for you to use SQL Server Management Studio.

1. Select Download now under Express in the 'Or, download a free specialized edition' section.
2. When the file is done downloading, open the file and allow the changes if prompted.
3. When the installer opens, select the Custom option.



4. Accept the default Media Location, then select Install.



5. From your SQL Server Installation Center window, Select New SQL Server standalone installation or add features to an existing installation.
6. After the file downloads, open it (.exe).
7. License Terms – Select I accept the license terms and Privacy Statement, then Next.
8. Global Rules – Select Next after Operation completed.
9. Microsoft Update – Select Use Microsoft Update to check for updates, then Next.
10. Install Rules – Select Next.
11. Azure Extension for SQL Server – Uncheck Azure Extension for SQL Server, then Next.
12. Feature Selection – Accept default selections, then Next.
13. Instance Configuration – Accept default selections, then Next.
14. Server Configuration – Accept default selections, then Next.
15. Database Engine Configuration – Select Mixed Mode (SQL Server authentication and Windows authentication) then enter a password. **Take note of your password and your SQL Server administrator string. My string is ROBERT\simo5**. Then Next.
16. Installation Progress – When this completes, it will go straight to the Complete page. Locate your Summary log file, then Close.

SQL Server Management Studio Installation

[Download SQL Server Management Studio (SSMS) - SQL Server Management Studio (SSMS) | Microsoft Learn]

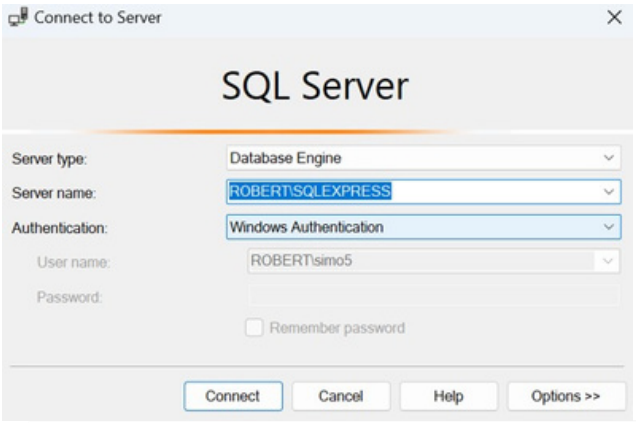
SQL Server Management Studio is the application that you will use to interact with the database stored in SQL Server Express (installed in the previous section).

1. Select Free Download for SQL Server Management Studio (SSMS). At the time of this book, it is version 19.0.1.
2. After the file downloads, open it (.exe).
3. When the installer opens, accept the default location, and select Install.
4. Allow Microsoft to make changes to your computer.
5. Once you receive the message that Setup was Completed, close the window.

Connecting SQL Server Management Studio to SQL Server Express

Before we are ready to use the tools we just installed, we must establish the connection from SQL Server Management Studio to SQL Server Express.

1. Open SQL Server Management Studio.
2. My instance had the Server type, Server name and authentication already populated. I'm on a Windows machine so if you have a different OS, you may need to switch to SQL Authentication.

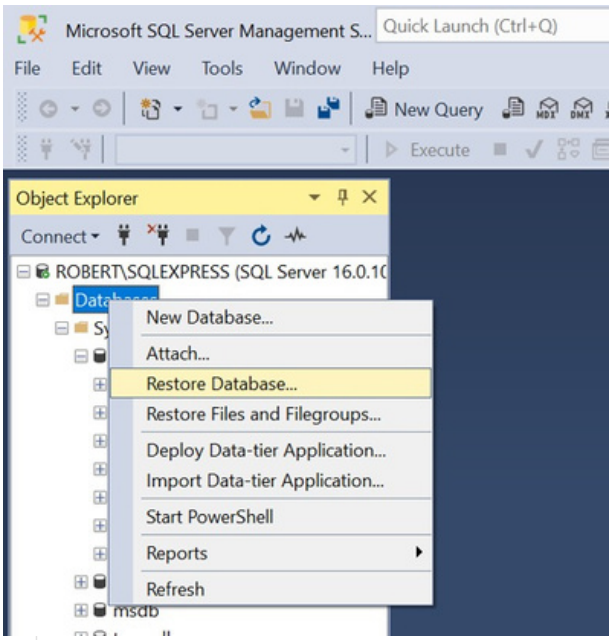


3. Press Connect.
4. If you are on a Mac and SQL Server Management Studio doesn't work, try installing Azure Data Studio and follow steps located here: [How to Install Azure Data Studio on a Mac (database.guide).]
5. If you can successfully connect, you will be ready to import a sample database and begin writing queries.

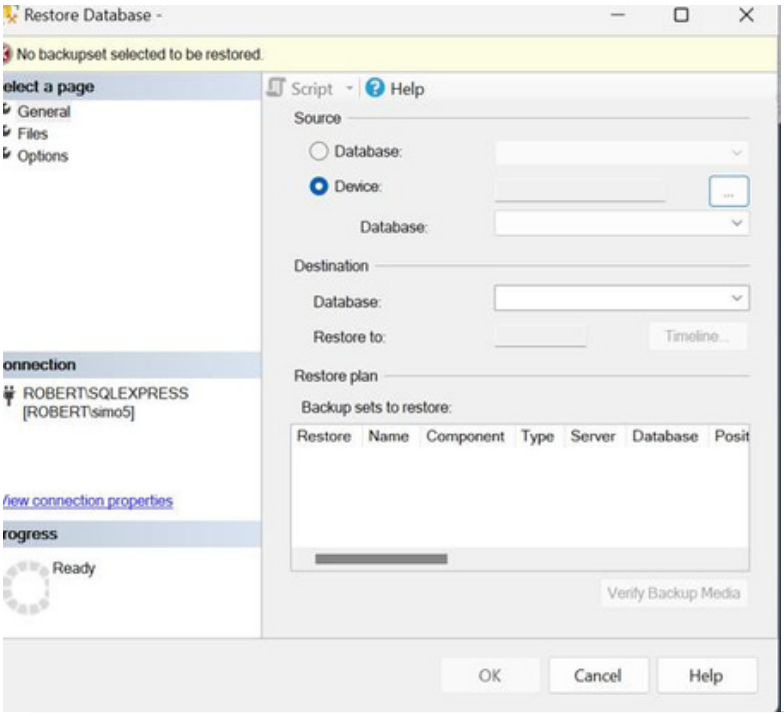
Installation of Sample Database

Before we are ready to use the tools we just installed, we must establish the connection from SQL Server Management Studio to SQL Server Express.

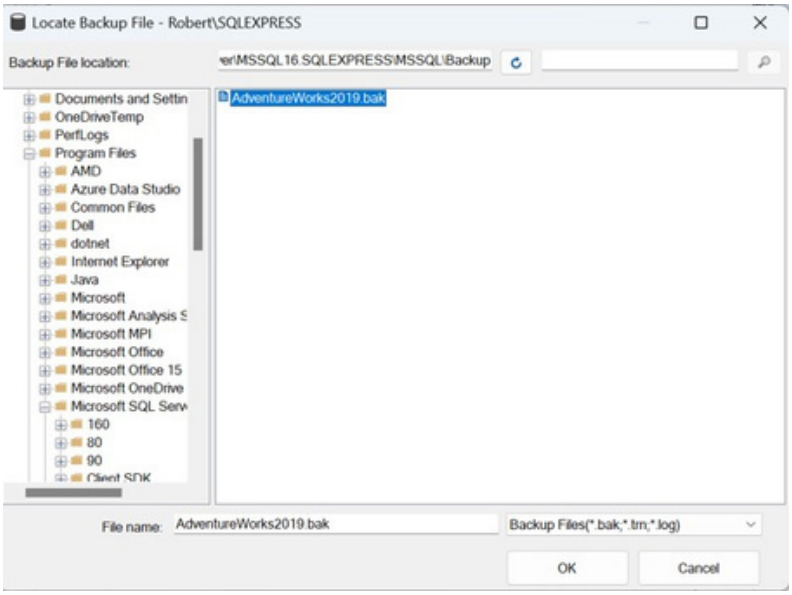
1. Download the OLTP format of the AdventureWorks database (AdventureWorks2019.bak) located here: [AdventureWorks sample databases - SQL Server | Microsoft Learn]
2. Move the .bak file to a similar location as this Backup path: C:\Program Files\Microsoft SQL Server\MSSQL16.SQLEXPRESS\MSSQL\Backup
3. Once the download is complete, open SQL Server Management Studio.
4. Within SQL Server Management Studio, right click on Databases then select Restore Database.



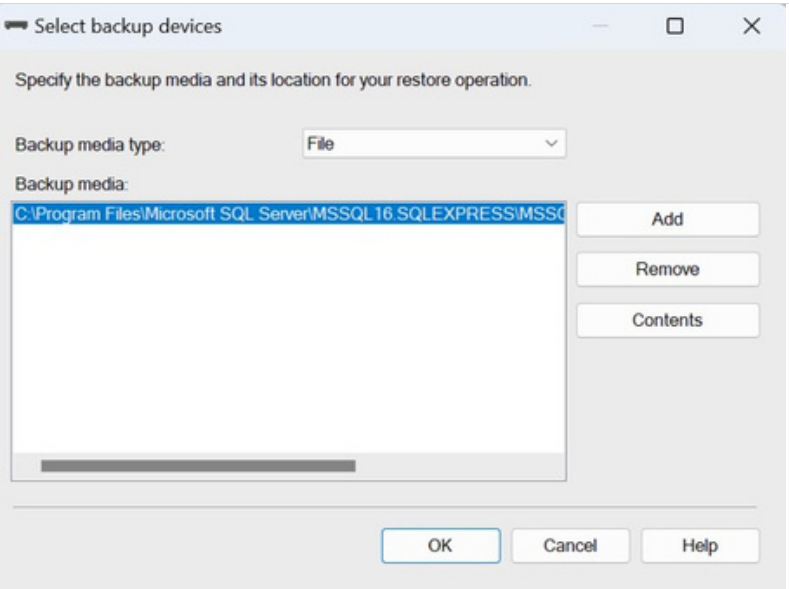
5. Select Device then click the ellipsis to the right of Device to search for your backup.



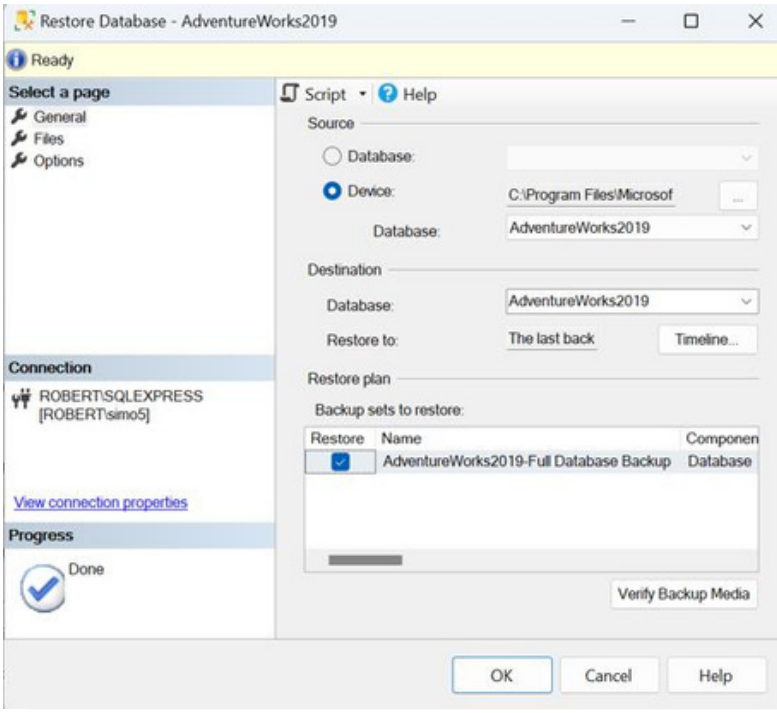
6. Select Add, then when the new window opens, select AdventureWorks2019.bak, then OK.



7. When the selection window closes, in the Select backup devices window, select the Backup media that is now showing, then OK.



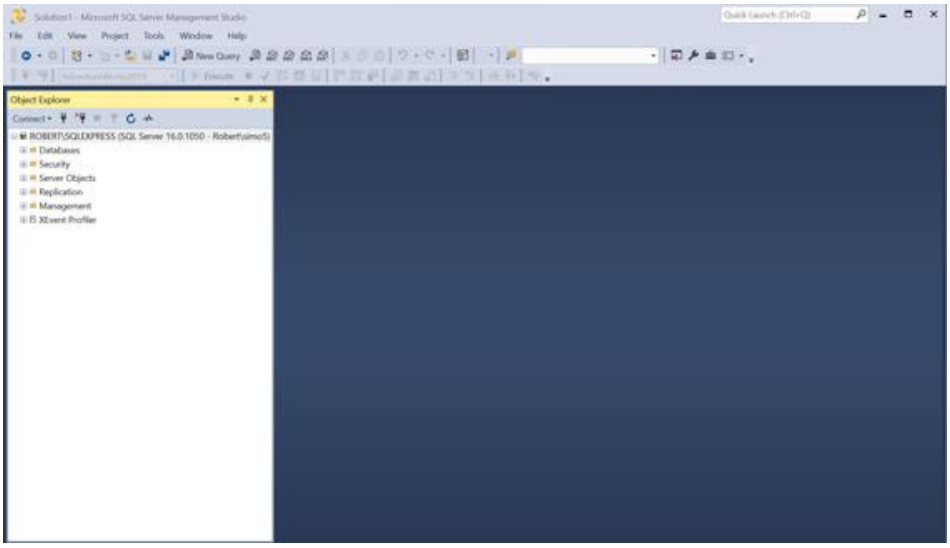
8. In your Restore Database window, ensure your parameters look like this, then press OK.



9. You should receive the following message: Database 'AdventureWorks2019' restored successfully.
10. You are now ready to start writing SQL – Go back to the Select section.

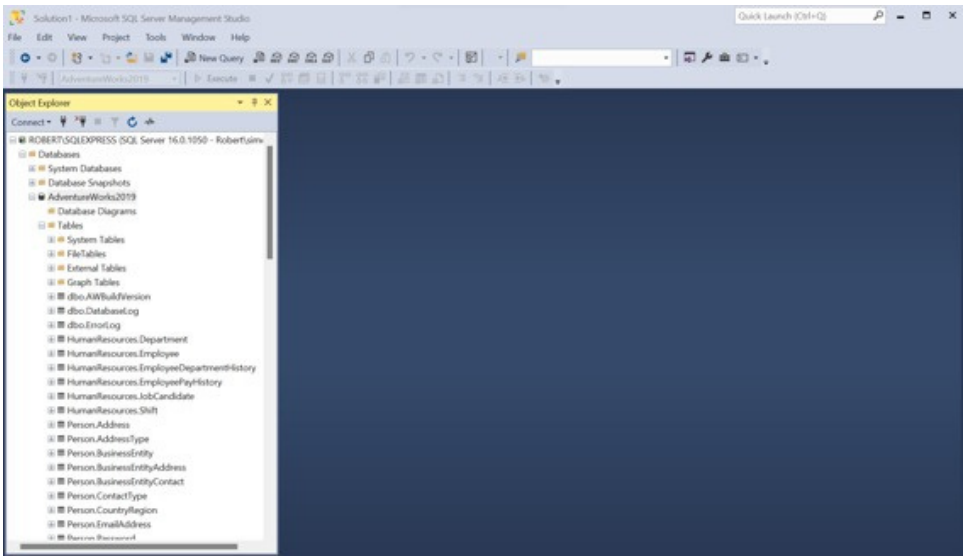
SQL Server Management Studio Navigation

1. SQL Server Management Studio (SSMS) is very simple to use. Once you login after installation, you will be presented with a screen like the below.



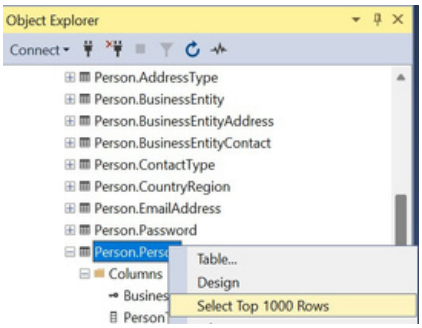
Database Navigation

1. Expand the Databases Node in the Object Explorer window, then expand AdventureWorks2019, then expand each table that you wish to explore to view the individual columns.



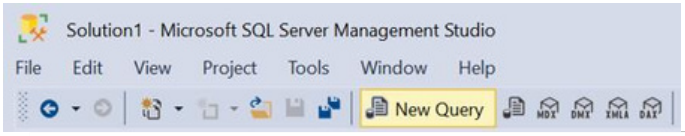
Preview Table

1. Right click on table name then ‘Select Top 1000 Rows’.

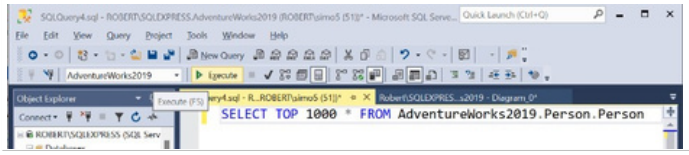


Create and Run Query

1. Select New Query in top menu bar.

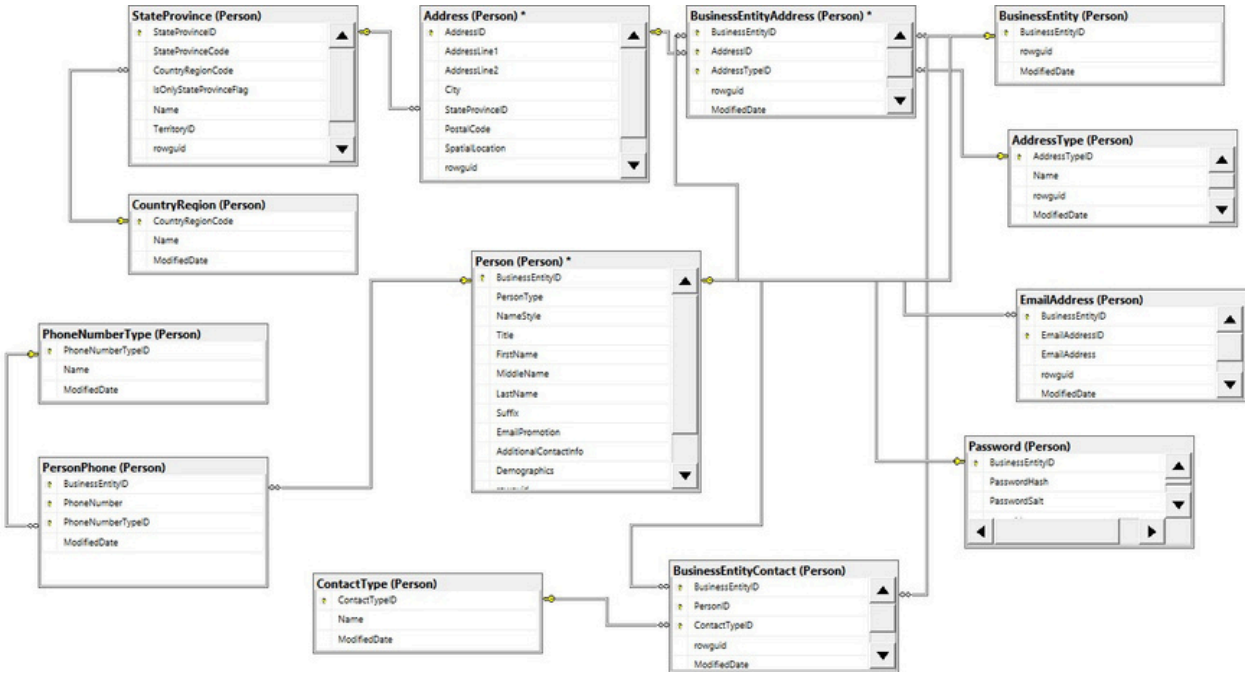


2. Then type a query such as the one in the screenshot below and press Execute. Results should appear in the lower window. If you have multiple scripts in the same window, then select the script you want to run, and press Execute.



ER Diagrams

Person



Sales

Human Resources