Chapter 6
Intermediate SQL

Introduction

Excellent progress so far with getting engaged with SQL! This chapter builds on the METRO_TRANSIT database we coded previously and continues with applying standard SQL commands within this context. Later, we will add more data in look-up tables and establish realistic-looking transactions to populate the database with 100,000's of rows of data. Completing this chapter will put you in a position to develop a stronger understanding of SQL and accelerate through more difficult programming structures including stored procedures and user-defined functions.

Again, this chapter begins to apply the theory, and fundamentals recently acquired into the beginnings of processing data and targeted learning through analytics. We will be introduced to system functions, subqueries, and other common commands that fill out your data toolbox. This is a fascinating journey!

Structure

By the end of this chapter, you will have a more nuanced understanding of SQL processes and be able to construct sufficient queries when posed with basic business questions. Topics of obtaining these skills include the following:

- Shortcuts that provide wider impact with less typing
- JOIN statements that unlock queries to connect an unlimited number of tables
- GROUP BY command that allows applying the functions from above across many rows of data as well as the HAVING clause to filter on aggregated results
- Functions that allow for working much more efficiently with a range of common data including string data, numerical values, aggregated data and NULLs

Let us begin this chapter with a reminder of the simplified version data model for the METRO_TRANSIT database we built and coded by hand last chapter:

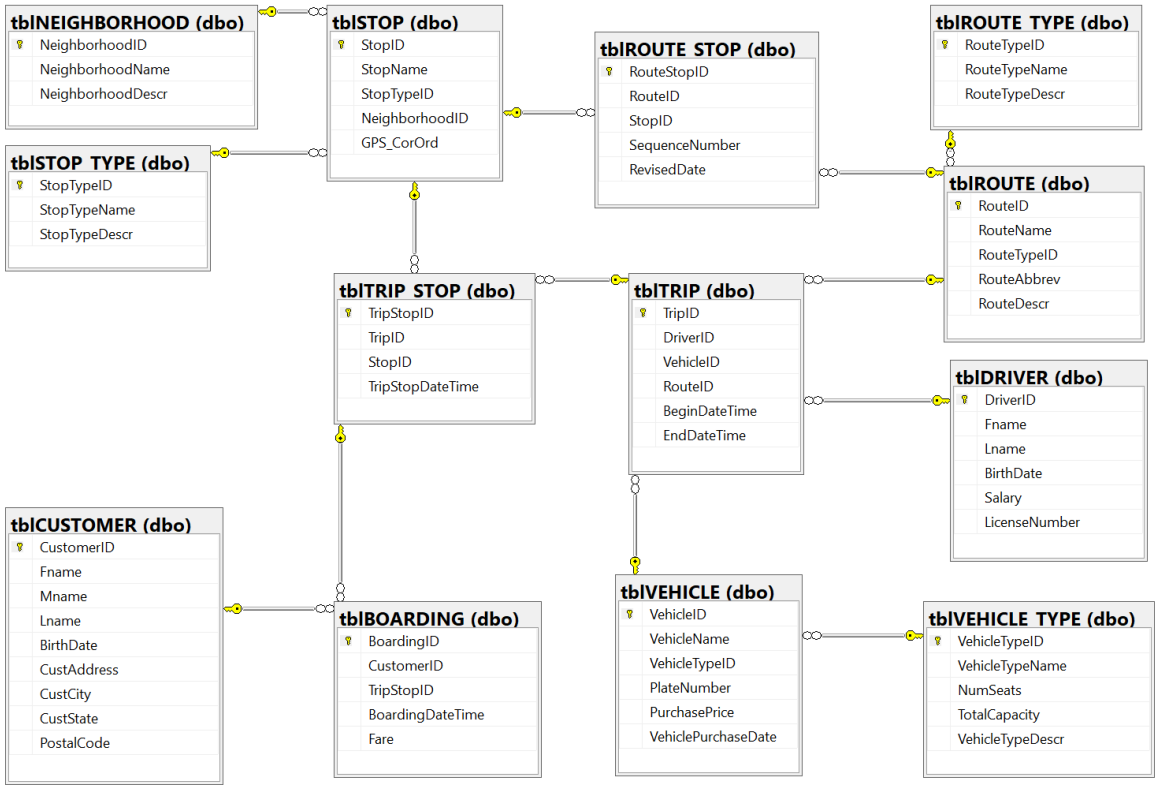


Figure 5.1: Reminder of the simplified version of METRO_TRANSIT

SHORTCUTS

Shortcuts are part of our everyday lives and the need for people to be more efficient; we often seek out shortcuts in many ways, such as getting to work quicker, finding the shortest line at the grocery store, or over-engineering a quicker method to get a weekend chore done at home. Programming languages, just like any spoken language, have slang; often single words that replace a bundle of others that are intended to simplify or shorten the time it takes to communicate a common practice. With SQL, it could be argued the entire language (as a 4GL) is constructed with slang! There are already over 100 reserved keywords that have been established to replace tedious coding details for repetitive tasks.

Think about each command and statement presented so far just in this book, such as CREATE, ALTER, DROP, and SELECT. These are all technical shortcuts. This sleekness is part of the charm, beauty, and brilliance of nearly every programming language and especially SQL. With this in mind, we have created “shortcuts on shortcuts” and added a few more keywords and symbols to further reduce the time and required typing to obtain efficiency. Please review the following common shortcuts:

Shortcut #1: Asterisk (*)

The asterisk (commonly referred to as 'star') is a shortcut to mean 'all columns' in the SELECT line of a query. This shortcut will return every column from all tables included in the query.

```
/*
open a query window and connect to the database designed in previous chapters:
METRO_TRANSIT
*/

SELECT *
FROM tblVEHICLE
GO
```

Many times, analysts begin investigations with simplicity and are effectively searching for any matching rows, often without concern of the minimal or filtered columns. We start with a quick query, see what gets returned, and then often narrow the search in the WHERE clause depending on what we learn from the preceding query. Eventually, we will most likely change the SELECT line to be a precise collection of columns that meet the needs of a recurring query.

Words of caution when using 'SELECT *' are that it can return an overwhelming number of columns that can quickly become difficult to manage, especially when multiple tables are connected in a single query. Many organizations following industry best practices will state that 'SELECT *' is to be avoided, particularly when queries are engaging in millions of rows of data, as an unfiltered query can clog a company's internal network with noncritical data.

Shortcut #2: SELECT..INTO

The SELECT..INTO sequence of SQL code makes a copy of an existing table. This includes the table definition/schema as well as potentially the entire data as well. This is a wonderful shortcut whenever we need a quick copy of a table.

```
/*
open a query window and connect to the database designed in previous chapters:
METRO_TRANSIT
*/

SELECT *
INTO Test_Copy_tblVEHICLE
FROM tblVEHICLE
GO


SELECT CustomerID, Fname, Lname
INTO Test_Copy_tblCUSTOMER
FROM tblCUSTOMER
WHERE BirthDate > 'January 1, 1955'
```

The SELECT INTO statement also allows for the filtering of columns in the SELECT line as well as adding a WHERE clause. Creating a whole copy or just a subset of an existing table is powerful and often takes just a few seconds.

Shortcut #3: INSERT INTO...SELECT

A great shortcut in SQL is when we want to copy data from an existing table into another. The common use cases are when we are generating a working set of data, perhaps during an investigation of an issue or perhaps even maintenance. Our goal may be to quickly isolate a subset of data into a new table for further work. We can do this with SQL by combining a SELECT statement with an INSERT statement. The only caveat is the destination table must already exist, which is different from a shortcut we just saw with SELECT..INTO which creates the destination table on the fly.

For example, let us try to obtain a subset of rows from tblSTOP where the stops are from two specific neighborhoods, such as 'Downtown', 'Waterfront', and 'Fremont'.

First, we must make sure the destination table exists. For the sake of this example, we will create a new entity now, called 'Tourist_Stops'.

```
CREATE TABLE tblTOURIST_STOPS
(TN_ID INTEGER IDENTITY(1,1) primary key,
NeighborhoodID INT not null,
NeighborhoodName varchar(50) not null,
StopID INT not null,
StopName varchar(50) not null,
GPS_CorOrd VARCHAR(50) NULL)
GO
```

Give the query below a try! It is important to be able to assemble copies of data when we are on a mission to conduct a deeper investigation or build one-off reports.

```
INSERT INTO tblTOURIST_STOPS (NeighborhoodID, NeighborhoodName,
StopID, StopName, GPS_CorOrd)
SELECT N.NeighborhoodID, NeighborhoodName, S.StopID, StopName,
GPS_CorOrd
FROM tblNEIGHBORHOOD N
      JOIN tblSTOP S ON N.NeighborhoodID = S.NeighborhoodID
WHERE N.NeighborhoodName IN ('Downtown', 'Waterfront', 'Fremont')
```

Please note, the columns in the SELECT line (the query) must align with the columns in the INSERT INTO line; otherwise the query will fail or perhaps put invalid data into your working table. Also, the WHERE clause above is using IN as a keyword to search for a collection of values.

Shortcut #4: ALIASES

Another common shortcut within programming languages (not just SQL) is substitute names. An alias in general is most often intended to reduce the number of letters needed to type. In fact, many people have abbreviations (also known as 'nicknames') of their formal or legal names. While my legal name is Gregory, I am known to my students as 'Greg', or 'Gus'. My siblings often call me by the first letter of my name 'G'. Nearly everyone has an email alias by which others can quickly contact us.

In SQL, the most common use of aliases is in queries, where we want to replace the long names of columns or tables. In the case of replacing column names, we are converting a cryptic 'short name' to one more appropriate to human readers. An example is as follows:

```
/*
Example query ***without*** any alias using database METRO_TRANSIT
*/

SELECT CustomerID, Fname, Lname
FROM tblCUSTOMER
GO
```

Results | Messages

	CustomerID	Fname	Lname
1	1	Ivey	Hazekamp
2	2	Darcel	Eustache
3	3	Kenyetta	Terron
4	4	Janey	Lundgren

Figure 5.2: Example query with no use of an alias

```
/*
Example query now using 2 aliases in the SELECT line
*/

SELECT CustomerID, Fname AS 'First', Lname AS 'Last'
FROM tblCUSTOMER
```

Results | Messages

	CustomerID	First	Last
1	1	Ivey	Hazekamp
2	2	Darcel	Eustache
3	3	Kenyetta	Terron
4	4	Janey	Lundgren

Figure 5.3: Example query with the use of two aliases ('First' and 'Last')

Please note that the column headers in the results have been replaced by the aliases!

When used effectively, aliases can not only reduce the typing required to execute a query but also eliminate a significant portion of the query code---thereby improving the readability for other users.

Hands-on Practice: query connecting tables ***without*** aliases

```
/*
The following query will return data to answer the question of "Which vehicle types were
driven by any person with the last name 'Lee' after June 1, 2024?"
*/


SELECT tblVEHICLE_TYPE.VehicleTypeID, tblVEHICLE_TYPE.VehicleTypeName,
tblDRIVER.Fname, tblDRIVER.Lname
FROM tblVEHICLE_TYPE, tblVEHICLE, tblTRIP, tblDRIVER
WHERE tblVEHICLE_TYPE.VehicleTypeID = tblVEHICLE.VehicleTypeID
AND tblVEHICLE.VehicleID = tblTRIP.VehicleID
AND tblTRIP.DriverID = tblDRIVER.DriverID
      AND tblDRIVER.Lname = 'Lee'
      AND tblTRIP.BeginDateTime > 'June 1, 2024'
```

Results Messages

	VehicleTypeID	VehicleTypeName	Fname	Lname
1	1	Standard Bus	Bruce	Lee

Figure 5.4: Example of a busy query with no use of any aliases

```
/*
Example query connecting tables WITH aliases
*/
```

The following query will return the same data as above, yet is substituting the names of tables with aliases:

```
SELECT VT.VehicleTypeID, VT.VehicleTypeName, D.Fname, D.Lname
FROM tblVEHICLE_TYPE VT, tblVEHICLE V, tblTRIP T, tblDRIVER D
WHERE VT.VehicleTypeID = V.VehicleTypeID
AND V.VehicleID = T.VehicleID
AND T.DriverID = D.DriverID
      AND D.Lname = 'Lee'
AND T.BeginDateTime > 'June 1, 2024'
```

	VehicleTypeID	VehicleTypeName	Fname	Lname
1	1	Standard Bus	Bruce	Lee

Figure 5.5: Example of a busy query with the use of several aliases

Please note the first mention of each table is completely spelled out, followed by the desired alias. We do not need to include the keyword 'AS' during this process. This use of aliases not only reduces typing but also makes the reading of the code easier for people.

Also notice please that we connect multiple tables along the primary key and foreign key relationships. This allows for the return of the matching data that is filtered exclusively to the values we seek (in this case, just the vehicles driven by all drivers with the last name of 'Lee'). This alignment of connecting across PK/FK is called referential integrity although we will also say 'transactional integrity'.

Reminder: Transactional table versus Look-up table

The above query connects several tables, beginning with tblVEHICHLE_TYPE, and includes tblVEHICLE, tblTRIP and tblDRIVER. The table tblTRIP records the individual travels of a bus on a route that picks up dozens of passengers over an hour or so. In this database, it may only have 100 rows of new data each week. Again, one row in tblTRIP represents a driver taking a bus through a scheduled route and picking up passengers before turning around and perhaps conducting a second trip. When compared to other databases found in large organizations, 100 rows each week is miniscule; however, in the METRO_TRANSIT database we designed, it is labelled 'highly' transactional. A second table in METRO_TRANSIT that can be labelled 'highly transactional' is tblTRIP_STOP. All this really means is that the bulk of the data in the entire database (the transactional details, represented by foreign keys) will be held in these tables.

When a query includes a highly transactional table (like our query above), potentially millions of rows of data will be considered. Depending on how we frame our question, we may get results that are answering our question literally and include details from a transactional table. Let me explain with an example from METRO_TRANSIT.

We must be aware whenever a query we write includes or 'crosses through' a transactional table as it has awakened a bee's nest of details. Now, many queries may need these details, such as whenever we want to determine revenue from all customers or total trips by vehicle. If, however, we want to know which customers have boarded a specific route on a particular day, we would need to be aware the results would include duplicate names if the customer happened to meet the search criteria in two separate instances. This is a very common mistake for younger and beginning analysts! Just be thoughtful each time writing a query; is it crossing a transactional table? If the answer is yes, the next question should be, "do I want duplicate names?" that represent each matching occurrence. There are occasions where we need both answers. Just be aware, intentional, and deliberate in writing each query. Let us see how to discern between these two queries when crossing a transactional table by using the keyword DISTINCT.

DISTINCT

The following query is from the METRO_TRANSIT database and passes through the transactional table tblTRIP. The fact that the query crosses a transactional table means there will be detailed ID values for each driver, vehicle, and route. In an average week of business for METRO_TRANSIT, there may be dozens of rows for each driver, vehicle, and route as each participates in delivering service to the public.

Type the following code to answer the question 'Which drivers have participated in a trip on any route that included stops in the neighborhood of Fremont during June of 2025'. Hint: we do NOT want duplicate names. HINT: This query will benefit from the DISTINCT keyword to only return the names of drivers who meet the search criteria as opposed to *each time* they met the search criteria.

```
SELECT DISTINCT D.DriverID, D.Fname, D.Lname
FROM tblDRIVER D, tblTRIP T, tblTRIP_STOP TS, tblSTOP S,
tblNEIGHBORHOOD N
WHERE T.DriverID = D.DriverID
AND T.TripID = TS.TripID
AND TS.StopID = S.StopID
AND S.NeighborhoodID = N.NeighborhoodID
      AND N.NeighborhoodName = 'Fremont'
      AND T.BeginDateTime >'June 1, 2025'
AND T.BeginDateTime < 'July 1, 2025'
```

Results Messages

	DriverID	Fname	Lname
1	4	Meryl	Streep

Figure 5.6: Example of a busy query with the use of DISTINCT keyword

Just for comparison, delete the word 'DISTINCT' from the select line and run the query a second time; the results you receive should produce duplicate rows because the same driver has matched the search criteria 2 times:

Results Messages

	DriverID	Fname	Lname
1	4	Meryl	Streep
2	4	Meryl	Streep

Figure 5.7: Example of a query without DISTINCT keyword showing duplicate rows

We will learn in the very next section how to do query-writing across multiple tables using the keyword JOIN, which might be easier to construct and has some performance benefits.

JOIN statement

The JOIN statement is where SQL picks up speed and flexes power by broadening the reach of accessing data across an entire database. We have already learned to write queries that span multiple tables by simply including additional tables in the FROM clause. The method we have learned already is considered 'old-school' and while flexible by allowing connections across any number of columns, the method is also a potential performance drag if the query is poorly written.

The JOIN statement is often preferred as it requires comparisons (the actual 'joining' of two tables) to be between primary key and foreign relationships. This may require a few extra minutes writing a more methodical query, it will avoid wasting time on mistakes such as comparing a column with phone numbers against a column with zip codes. Another example from METRO_TRANSIT, is searching for a driver's last name or birthdate against a vehicle license plate number. Even if there happens to be a 'miraculous match' does it mean there is any business relationship or transactional relevance? I suppose there *might* be a situation where incorrect data needs to be located and cleaned up but for the most part, we will rarely need to join on unrelated columns.

By using the PK/FK relationships built into the database design, the JOIN statement can take advantage of default indexes and often complete a search through 100,000's of rows in a second or two.

There are a handful of JOIN statements in SQL, including INNER, OUTER, LEFT, RIGHT, SELF, and SEMI. While this may seem like a lot of options, most joins will be the default of LEFT INNER JOIN. Let's see these in action with some examples using Venn Diagrams as well as coding ourselves in the METRO_TRANSIT database!

INNER JOIN

An INNER JOIN is by far the most common type of JOIN in SQL. In fact, when we just type JOIN it is the default action. This default JOIN returns the matching rows between 2 or more tables. In the Venn Diagram below, the intersection or 'overlap' between the blue circle and the yellow circle is colored green.

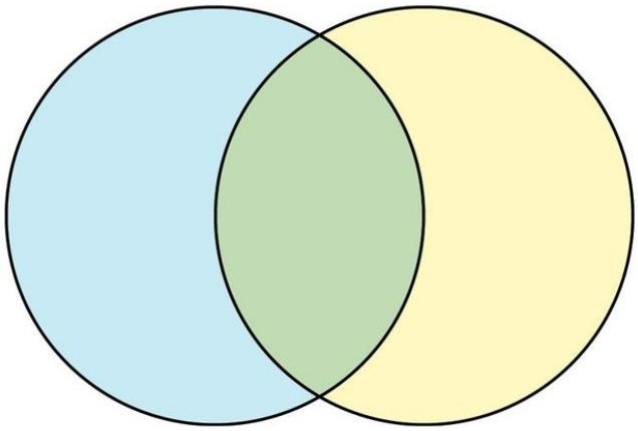


Figure 5.8: An example Venn Diagram showing the intersection of two circles in green.

Again, there is no difference between an INNER JOIN and just 'JOIN'. Many times, analysts seek the intersection between two or more tables and gravitate to this JOIN. An example of seeking the intersection between two tables from the METRO_TRANSIT database is "which routes are of type express?".

```
SELECT R.RouteID, R.RouteName, RT.RouteTypeName
FROM tblROUTE_TYPE RT
      JOIN tblROUTE R ON RT.RouteTypeID = R.RouteTypeID
WHERE RT.RouteTypeName = 'Express'
```

If you are writing queries against the abbreviated METRO_TRANSIT database we created in the previous chapter, there are 2 rows returned as follows:

Results Messages

	RouteID	RouteName	RouteTypeName
1	1	Sodo-Downtown Express	Express
2	3	Fremont-Downtown Express	Express

Figure 5.9: Example of a query with the use of JOIN keyword

Multiple JOIN statements in a single query

Let us see an example query that JOINs several tables from the METRO_TRANSIT database. The following query from the abbreviated METRO_TRANSIT diagram seeks to return the names of drivers, vehicles, vehicle types, as well as the BeginDateTime of all trips through the neighborhood of 'downtown'. There will be six JOIN statements:

```
SELECT VT.VehicleTypeName, VehicleName, Fname, Lname, T.BeginDateTime
FROM tblVEHICLE_TYPE VT
      JOIN tblVEHICLE V ON VT.VehicleTypeID = V.VehicleTypeID
      JOIN tblTRIP T ON V.VehicleID = T.VehicleID
      JOIN tblDRIVER D ON T.DriverID = D.DriverID
      JOIN tblTRIP_STOP TS ON T.TripID = TS.TripID
```

```
      JOIN tblSTOP S ON TS.StopID = S.StopID
      JOIN tblNEIGHBORHOOD N ON S.NeighborhoodID = N.NeighborhoodID
WHERE NeighborhoodName = 'Downtown'
```

	VehicleTypeName	VehicleName	Fname	Lname	BeginDateTime
1	Standard Bus	PA7	Bruce	Lee	2025-04-24 06:57:00.000
2	Double-Decker Bus	GS204	Meryl	Streep	2025-04-23 07:23:00.000

Figure 5.10: Example query showing results from multiple JOIN statements

As seen in the query for Figure 5.10, there is essentially no limit to the number of entities that can be joined in a single SQL query. The questions may be the following:

1) 'How do I include a table that is not directly connected to the previous table?'

2) 'Does it matter where I begin writing a query with multiple join statements?'

Great questions!

First, once a table has been 'joined', it belongs in the ever-expanding base of tables the query will draw data from. Looking at the query in Figure 5.10, the join sequence is direct from tblVEHICLE_TYPE, through tblVEHICLE, tblTRIP, and reaching tblDRIVER. Next, the query needs to 'jump' back to tblTRIP to join tblTRIP_STOP and continue towards tblNEIGHBORHOOD. This is fine!

Second, it does not matter where your query begins with multiple join statements (either left-to-right in the diagram, right-to-left, top-down, or bottom-up). That said, there is an element of elegance when *reading* a query that includes multiple join statements. The best rule for most queries is to start at an 'edge' of the diagram and join tables in the shortest stream that makes sense.

The sequence of joins in the query for Figure 5.10 is very easy to follow in the diagram as it represents a linear or sequential connection stream from the lower right of the ERD to the upper left (with a brief jump to get driver data):

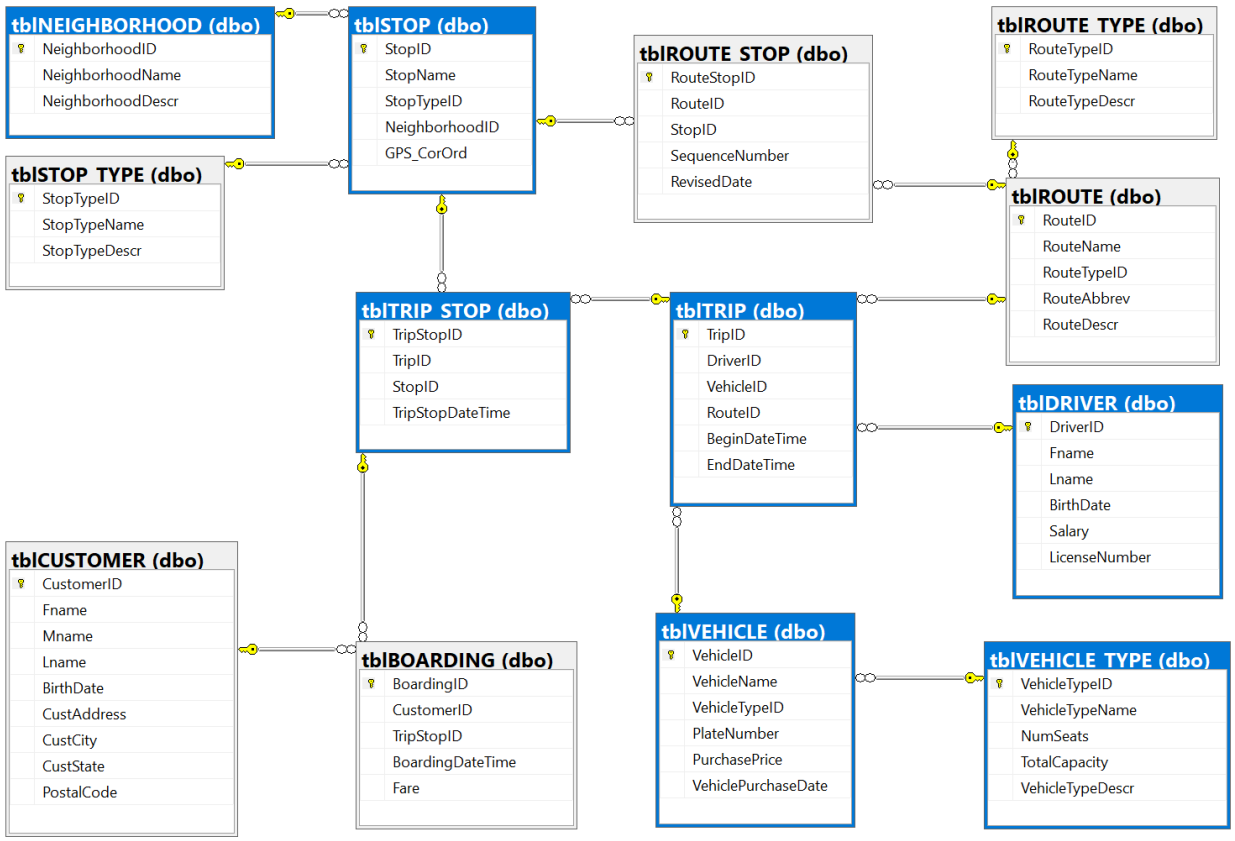


Figure 5.11: Example query showing table connected in multiple JOIN statements

Compare the query that was written for Figure 5.10 with the somewhat scattered version of the same query below:

```
SELECT VT.VehicleTypeName, VehicleName, Fname, Lname, T.BeginDateTime
FROM tblDRIVER D
      JOIN tblTRIP T ON D.DriverID = T.DriverID
      JOIN tblVEHICLE V ON T.VehicleID = V.VehicleID
      JOIN tblTRIP_STOP TS ON T.TripID = TS.TripID
      JOIN tblSTOP S ON TS.StopID = S.StopID
      JOIN tblVEHICLE_TYPE VT ON V.VehicleTypeID = VT.VehicleTypeID
      JOIN tblNEIGHBORHOOD N ON S.NeighborhoodID = N.NeighborhoodID
WHERE NeighborhoodName = 'Downtown'
```

	VehicleTypeName	VehicleName	Fname	Lname	BeginDateTime
1	Standard Bus	PA7	Bruce	Lee	2025-04-24 06:57:00.000
2	Double-Decker Bus	GS204	Meryl	Streep	2025-04-23 07:23:00.000

Figure 5.12: Example of a query and results with a scattered approach to joining tables

Please note the query results in Figure 5.12 are the exact same as the results from the query in Figure 5.10. The only difference between the two should be the time it takes to run as the stream-lined version in query 5.10 should be noticeably faster.

The key takeaway is the order by which tables are joined in a complex query with multiple join statements does not functionally matter. If each join statement connects via proper primary and foreign key relationships, the sequence of join statements does not make a difference in whether the query will parse. As for readability and performance, it is better for the join statements to be in a single direct sequence with as few jumps as possible.

OUTER JOIN

Whereas the INNER JOIN filters only the intersection of two or more tables, an OUTER JOIN returns all rows of each table regardless of whether there are matching values in the other table. The Venn diagram of all values may be rendered as follows:

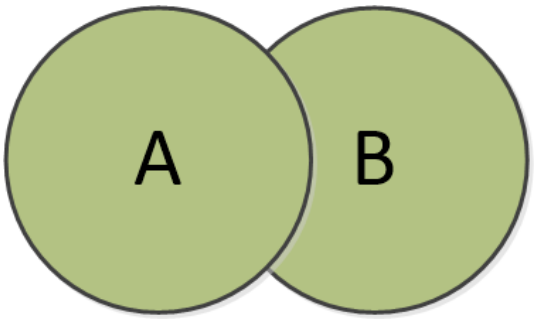


Figure 5.13: A Venn Diagram showing the FULL OUTER JOIN with all rows returned

Please note in Figure 5.13 that both circles (the 'entire collection') are colored in green; this reflects that all values in both entities are returned in a FULL OUTER JOIN.

```
SELECT V.VehicleID, V.VehicleName, T.RouteID, T.BeginDateTime
FROM tblVEHICLE V
      LEFT OUTER JOIN tblTRIP T ON V.VehicleID = T.VehicleID
```

	VehicleID	VehicleName	RouteID	BeginDateTime
1	1	GS203	6	2025-06-22 06:42:00.000
2	2	GS204	6	2025-06-22 06:17:00.000
3	2	GS204	5	2025-06-23 07:23:00.000
4	2	GS204	4	2025-06-29 08:07:00.000
5	3	PA7	6	2025-06-22 06:57:00.000
6	4	GS208	NULL	NULL

Figure 5.14: Example query showing results from LEFT OUTER JOIN with all rows returned, including NULL values from non-matching rows

The results from the query in Figure 5.14 show all rows from tblVEHICLE and all rows from tblTRIP, including row 6 in the results set that have NULL (highlighted in blue). The reason there are NULL values is because VEHICLE with VehicleID of 4 and VehicleName 'GS208', has not yet been assigned a TRIP.

Also, please notice there are three rows of values that match 2 under the column of VehicleID, and three matches for value of 'GS204' under column VehicleName. This reflects that this vehicle has been assigned 3 separate trips. These should not be considered 'duplicates' as the TripID, RouteID, and therefore BeginDateTimes are different.

Let's see essentially the same exact query with the substitution of 'RIGHT' in place of the word LEFT from the query we just saw in Figure 5.14:

```
SELECT V.VehicleID, V.VehicleName, T.RouteID, T.BeginDateTime
FROM tblVEHICLE V
      RIGHT OUTER JOIN tblTRIP T ON V.VehicleID = T.VehicleID
```

	VehicleID	VehicleName	RouteID	BeginDateTime
1	2	GS204	6	2025-06-22 06:17:00.000
2	1	GS203	6	2025-06-22 06:42:00.000
3	2	GS204	5	2025-06-23 07:23:00.000
4	3	PA7	6	2025-06-22 06:57:00.000
5	2	GS204	4	2025-06-29 08:07:00.000

Figure 5.15: Example query showing results from RIGHT OUTER JOIN

The query from Figure 5.15 represents a substitution of the word 'LEFT' with the word 'RIGHT'. With this change, the results are now only 5 rows as the table on the 'RIGHT' in the query (which is tblTRIP) has no NULL values for VehicleID or RouteID.

Experiment with LEFT OUTER JOIN and RIGHT OUTER JOIN in a few queries on your own with the limited data set we have hand-typed from chapter 4; this process will make this concept clearer as we progress through the introduction of other JOIN terms.

LEFT JOIN

A LEFT JOIN is the same thing as a LEFT OUTER JOIN with less typing!

LEFT JOIN returns all rows from the first table and the matching values from the second table; if there is a row in the first table, and no matching values in the second table, then a NULL is returned.

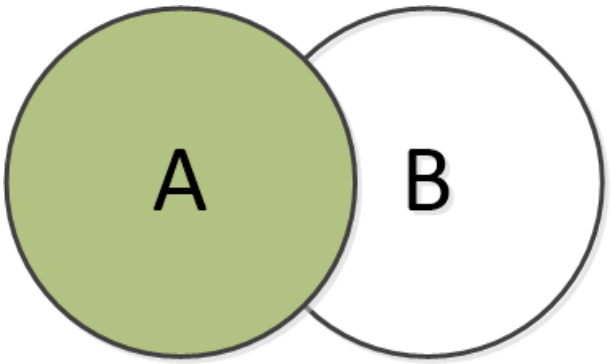


Figure 5.16: A Venn Diagram showing a LEFT JOIN with all rows returned from table A on the left and NULL values for any rows not matching from table B on the right

```
SELECT RT.RouteTypeName, R.RouteName
FROM tblROUTE_TYPE RT
      LEFT JOIN tblROUTE R ON RT.RouteTypeID = R.RouteTypeID
```

Results | Messages

	RouteTypeName	RouteName
1	Express	Sodo-Downtown Express
2	Express	Fremont-Downtown Express
3	Special Event	Fremont-Waterfront-Downtown
4	Regular	Sodo-Downtown Commuter
5	Regular	Fremont-Downtown Commuter
6	Regular	Capitol Hill-Downtown
7	Inclement Weather/Reduced Service	NULL

Figure 5.17: An example result of LEFT JOIN returning NULL value for non-matching row

Now, most of the time there will be a route somewhere that matches every route type. In fact, the query results we have in the database return a NULL value for RouteTypeName matching 'Inclement Weather/Reduced Service' because no route has yet to be assigned this specific route type. Every other RouteTypeName has at least one matching row in the ROUTE table. Let us add one more row to see another example with an INSERT.

Consider for a moment, that the administration of METRO_TRANSIT is adding a new route type of 'Waterfront Bypass' due to some sort of emergency construction. For the time that the new route type name of 'waterfront bypass' is entered into the system until specific routes have been assigned this associated RouteTypeID, the above query will return a NULL value for 'Waterfront Bypass'.

```
INSERT INTO tblROUTE_TYPE (RouteTypeName, RouteTypeDescr)
VALUES ('Waterfront Bypass', 'Routes that are in place during
construction of new seawall barrier along waterfront')
```

Now, re-run the earlier query with LEFT JOIN

```
SELECT RT.RouteTypeName, R.RouteName
FROM tblROUTE_TYPE RT
      LEFT JOIN tblROUTE R ON RT.RouteTypeID = R.RouteTypeID
```

Results | Messages

	RouteTypeName	RouteName
1	Express	Sodo-Downtown Express
2	Express	Fremont-Downtown Express
3	Special Event	Fremont-Waterfront-Downtown
4	Regular	Sodo-Downtown Commuter
5	Regular	Fremont-Downtown Commuter
6	Regular	Capitol Hill-Downtown
7	Inclement Weather/Reduced Service	NULL
8	Waterfront Bypass	NULL

Figure 5.18: An example result of LEFT JOIN returning NULL value for non-matching rows

We see the addition of one more row in the LEFT table (which is tblROUTE_TYPE) and a corresponding NULL value when joined to tblROUTE. The reason for this is for a moment, no routes have been created to have RouteTypeID that aligns with 'Waterfront Bypass'.

RIGHT JOIN

A RIGHT JOIN is the same thing as a RIGHT OUTER JOIN with less typing.

RIGHT JOIN is effectively the opposite of a LEFT JOIN in that it returns all rows from the second table (Table B below) and the matching values from the first table (Table A below). This means if there is a row in the second table and no matching values in the first table, then a NULL is returned in place of a value for the first table.

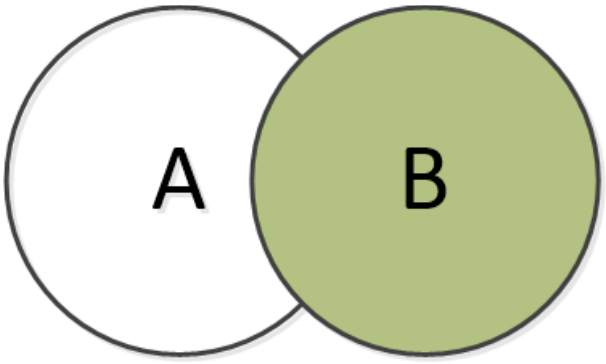


Figure 5.19: A Venn Diagram showing a RIGHT JOIN with all rows returned from table B on the right and NULL values for any rows not matching from table A on the left

Let us explore the RIGHT JOIN with another example query from METRO_TRANSIT. This time we want to return every row from the NEIGHBORHOOD table as well as every row

from STOP. If there happens to be a Neighborhood that currently does not have any stops assigned, then those rows will still be returned however there will be NULL values instead of the name of a stop.

Be sure to put tblNEIGHBORHOOD as the second table!

```
SELECT S.StopID, S.StopName, N.NeighborhoodName
FROM tblSTOP S
      RIGHT JOIN tblNEIGHBORHOOD N
            ON S.NeighborhoodID = N.NeighborhoodID
```

Results | Messages

	StopID	StopName	NeighborhoodName
1	6	Broadway Avenue and Cherry Street	Capitol Hill
2	3	Elliott Avenue and Mercer Street	Waterfront
3	4	Sixth Avenue and Battery Street	Downtown
4	5	Fourth Avenue and Seneca Street	Downtown
5	7	First Avenue and Terry Street	SoDo
6	1	Hwy 99-N 54th	Fremont
7	2	Hwy 99-N 36th	Fremont

Figure 5.20: An example query result from a RIGHT JOIN

Now, let's add a new neighborhood to prove how the RIGHT JOIN will behave.

```
INSERT INTO tblNEIGHBORHOOD (NeighborhoodName, NeighborhoodDescr)
VALUES ('Richmond Highlands', 'Smaller neighborhood with a mix of
shops, light industry, and schools West of Highway 99 in Shoreline')
```

Please re-run the previous query with the recently added row of a new neighborhood:

```
SELECT S.StopID, S.StopName, N.NeighborhoodName
FROM tblSTOP S
      RIGHT JOIN tblNEIGHBORHOOD N ON S.NeighborhoodID =
N.NeighborhoodID
```

Results | Messages

	StopID	StopName	NeighborhoodName
1	6	Broadway Avenue and Cherry Street	Capitol Hill
2	3	Elliott Avenue and Mercer Street	Waterfront
3	4	Sixth Avenue and Battery Street	Downtown
4	5	Fourth Avenue and Seneca Street	Downtown
5	7	First Avenue and Terry Street	SoDo
6	1	Hwy 99-N 54th	Fremont
7	2	Hwy 99-N 36th	Fremont
8	NULL	NULL	Richmond Highlands

Figure 5.21: An example query result from a RIGHT JOIN with a NULL value

The results set have returned 1 row with a NULL value for the recently added row in tblNEIGHBORHOOD because no stops have been assigned the NeighborhoodID of 'Richmond Highlands'.

SELF JOIN

A self-join is used when trying to locate data from a table that equals other data from the same table. A very common example of this is an EMPLOYEE table that has a primary key of the EmpID column for each employee but also a second EmpID for tracking the employee's manager. The second EmpID tracking the manager is a foreign key back to the primary key in the same table. At first glance this seems weird.

When we want to write a query to return the list of Employees and their managers, we will need to join two instances of the same table. Nothing needs to change in the process of writing our query as we just make sure to have different aliases for each instance below:

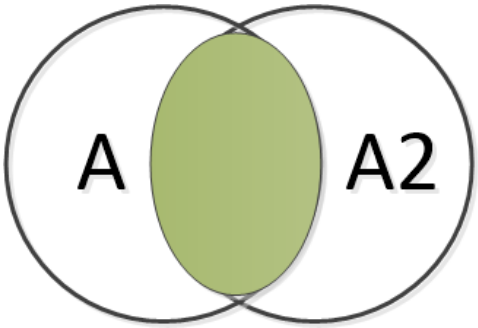


Figure 5.22: A Venn Diagram showing a SELF JOIN with the intersection of rows returned from a query drawing data from two separate instances of one table (same table…joined to itself with different aliases).

To see a self-join example, we will need to make a slight change to the schema of METRO_TRANSIT. Imagine in METRO_TRANSIT that each driver is now required to have another driver (who might be slightly older or more experienced) assigned to them as a mentor. How might we track this assignment in our database? As it is currently designed, we have no column for tracking a mentor. Let's fix this!

```
/*
The following query will add a new column MentorID to the tblDRIVER table as well as
making the primary key (DriverID) a foreign key in the same table!
*/
```

```
ALTER TABLE tblDRIVER
ADD MentorID INT
FOREIGN KEY
REFERENCES tblDRIVER (DriverID)
```

Next, let's update a few rows to reflect potential mentoring relationships.

```
UPDATE tblDRIVER
SET MentorID =
      (SELECT DriverID
      FROM tblDRIVER
      WHERE Fname = 'Bruce' and Lname = 'Lee')
WHERE DriverID =
      (SELECT DriverID
      FROM tblDRIVER
      WHERE Fname = 'Jimi' and Lname = 'Hendrix')
GO

UPDATE tblDRIVER
SET MentorID =
      (SELECT DriverID
      FROM tblDRIVER
      WHERE Fname = 'Jimi' and Lname = 'Hendrix')
WHERE DriverID =
      (SELECT DriverID
      FROM tblDRIVER
      WHERE Fname = 'Meryl' and Lname = 'Streep')
GO
```

Now we have added these values into our database, we can write a self-join to return the values in the context of a report.

```
SELECT D1.Fname AS Driver_First, D1.Lname AS Driver_Last, D2.Fname AS
Mentor_First, D2.Lname AS Mentor_Last
FROM tblDRIVER D1
      JOIN tblDRIVER D2 ON D1.DriverID = D2.MentorID
```

Results | Messages

	Driver_First	Driver_Last	Mentor_First	Mentor_Last
1	Bruce	Lee	Jimi	Hendrix
2	Jimi	Hendrix	Meryl	Streep

Figure 5.23: A query result from a Self-Join with DRIVER joining itself

Please note the query results in Figure 5.23 is drawing data from two copies of tblDRIVER (labeled as D1 and D2). The results show the driver and their respective mentor. Self joins are not very common, but when there is a need to layer queries, knowing how to connect two copies of a single together is invaluable.

CROSS JOIN

A cross join returns a Cartesian product of each row in one table and every single matching combination from another table. While this kind of query can result in an overwhelming number of rows, there are some situations where it may be the only way to answer a question. In the METRO TRANSIT database, it might be important to see each trip that every driver has ever been assigned. Let's use round numbers for simplicity.

If METRO_TRANSIT was used by a medium-sized city, it may have perhaps 500 drivers in the database. If each driver is assigned 100 trips a month, after 5 years (obviously 60 months) there may be 3 million rows of data. Since it would be nearly impossible for a person to read 3 million rows of data, the use of CROSS JOIN is probably limited to capturing data for feeding another application for analyses or perhaps archival or historical purposes.

Let's see this example from METRO_TRANSIT first with a simple JOIN s a comparison:

```
SELECT *
FROM tblDRIVER D
      JOIN tblTRIP T ON D.DriverID = T.DriverID
```

Results Messages

	DriverID	Fname	Lname	BirthDate	Salary	LicenseNumber	MentorID	TripID	DriverID	VehicleID	RouteID	BeginDateTime	EndDateTime
1	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	1	4	2	6	2025-06-22 06:17:00.000	2025-06-22 06:50:00.000
2	1	Jimi	Hendrix	1942-11-27	42590.00	JMHend3487RR	2	2	1	1	6	2025-06-22 06:42:00.000	2025-06-22 07:29:00.000
3	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	3	4	2	5	2025-06-23 07:23:00.000	2025-06-23 08:17:00.000
4	2	Bruce	Lee	1940-11-27	42590.00	BTLee*2198DL	NULL	4	2	3	6	2025-06-22 06:57:00.000	2025-06-22 07:48:00.000
5	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	5	4	2	4	2025-06-29 08:07:00.000	2025-06-29 08:48:00.000

Figure 5.24: A query result from a standard JOIN

A standard JOIN returns 5 rows from the METRO_TRANSIT database as it only returns values that match across primary and foreign key relationships. Now compare to a CROSS JOIN without any restrictions:

```
SELECT *
FROM tblDRIVER D
      CROSS JOIN tblTRIP T
```

Results Messages

	DriverID	Fname	Lname	BirthDate	Salary	LicenseNumber	MentorID	TripID	DriverID	VehicleID	RouteID	BeginDateTime	EndDateTime
1	1	Jimi	Hendrix	1942-11-27	42590.00	JMHend3487RR	2	1	4	2	6	2025-06-22 06:17:00.000	2025-06-22 06:50:00.000
2	1	Jimi	Hendrix	1942-11-27	42590.00	JMHend3487RR	2	2	1	1	6	2025-06-22 06:42:00.000	2025-06-22 07:29:00.000
3	1	Jimi	Hendrix	1942-11-27	42590.00	JMHend3487RR	2	3	4	2	5	2025-06-23 07:23:00.000	2025-06-23 08:17:00.000
4	1	Jimi	Hendrix	1942-11-27	42590.00	JMHend3487RR	2	4	2	3	6	2025-06-22 06:57:00.000	2025-06-22 07:48:00.000
5	1	Jimi	Hendrix	1942-11-27	42590.00	JMHend3487RR	2	5	4	2	4	2025-06-29 08:07:00.000	2025-06-29 08:48:00.000
6	2	Bruce	Lee	1940-11-27	42590.00	BTLee*2198DL	NULL	1	4	2	6	2025-06-22 06:17:00.000	2025-06-22 06:50:00.000
7	2	Bruce	Lee	1940-11-27	42590.00	BTLee*2198DL	NULL	2	1	1	6	2025-06-22 06:42:00.000	2025-06-22 07:29:00.000
8	2	Bruce	Lee	1940-11-27	42590.00	BTLee*2198DL	NULL	3	4	2	5	2025-06-23 07:23:00.000	2025-06-23 08:17:00.000
9	2	Bruce	Lee	1940-11-27	42590.00	BTLee*2198DL	NULL	4	2	3	6	2025-06-22 06:57:00.000	2025-06-22 07:48:00.000
10	2	Bruce	Lee	1940-11-27	42590.00	BTLee*2198DL	NULL	5	4	2	4	2025-06-29 08:07:00.000	2025-06-29 08:48:00.000
11	3	Jim	Morrison	1943-12-08	47625.00	JPMorr7762WS	NULL	1	4	2	6	2025-06-22 06:17:00.000	2025-06-22 06:50:00.000
12	3	Jim	Morrison	1943-12-08	47625.00	JPMorr7762WS	NULL	2	1	1	6	2025-06-22 06:42:00.000	2025-06-22 07:29:00.000
13	3	Jim	Morrison	1943-12-08	47625.00	JPMorr7762WS	NULL	3	4	2	5	2025-06-23 07:23:00.000	2025-06-23 08:17:00.000
14	3	Jim	Morrison	1943-12-08	47625.00	JPMorr7762WS	NULL	4	2	3	6	2025-06-22 06:57:00.000	2025-06-22 07:48:00.000
15	3	Jim	Morrison	1943-12-08	47625.00	JPMorr7762WS	NULL	5	4	2	4	2025-06-29 08:07:00.000	2025-06-29 08:48:00.000
16	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	1	4	2	6	2025-06-22 06:17:00.000	2025-06-22 06:50:00.000
17	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	2	1	1	6	2025-06-22 06:42:00.000	2025-06-22 07:29:00.000
18	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	3	4	2	5	2025-06-23 07:23:00.000	2025-06-23 08:17:00.000
19	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	4	2	3	6	2025-06-22 06:57:00.000	2025-06-22 07:48:00.000
20	4	Meryl	Streep	1949-06-22	46965.00	MDStre8121UJ	1	5	4	2	4	2025-06-29 08:07:00.000	2025-06-29 08:48:00.000

Figure 5.25: A query result from a CROSS JOIN

A CROSS JOIN produces a cartesian product and in this case returns 20 rows. In the image above, also note the NULL values for MentorID for Bruce Lee and Jim Morrison since they were not yet assigned mentors. Bruce did not get a mentor because, well, he did not need one and Jim did not get one because he refused to listen to anyone.

Multiple Conditions in JOIN

One last set of examples with JOIN statements needs to include multiple filters. In the following example, we are adding a JOIN filter to determine if the driver is completing the trip during the month of their birthday. NOTE: MONTH() is a date function and will be explored in greater detail with other Date functions soon!

```
SELECT D.DriverID, Fname, Lname, BirthDate, T.TripID, BeginDateTime
FROM tblDRIVER D
      JOIN tblTRIP T ON D.DriverID = T.DriverID
      AND MONTH(D.BirthDate) = MONTH(T.BeginDateTime)
```

Results Messages

	DriverID	Fname	Lname	BirthDate	TripID	BeginDateTime
1	4	Meryl	Streep	1949-06-22	1	2025-06-22 06:17:00.000
2	4	Meryl	Streep	1949-06-22	3	2025-06-23 07:23:00.000
3	4	Meryl	Streep	1949-06-22	5	2025-06-29 08:07:00.000

Figure 5.26: A query result with multiple JOIN conditions

Please note that June matches both the month that Meryl Streep was born as well as when the trips were operated (satisfying the conditions of the query). The above query could quite easily be written with the second condition in a separate WHERE clause, but this method may perform faster, depending on the number of rows in the table.

FUNCTIONS

Functions are beautiful! Functions are intended to save time and reduce errors on repetitive tasks, such as getting the current time of day or perhaps total up the bill for an order. A function is simply a pre-written block of code that is named, saved, and ready to run whenever called. There are several different types of functions, including system functions, numeric, string, NULL, aggregate, date, and ranking functions as well as those we can design and code ourselves (called user-defined functions). Most database systems will include more than 100 functions built-in when the system is installed. As mentioned, functions may be the most beautiful item in all programming, not just SQL. We save time in three instances when using functions which can be categorized as *before*, *during*, and *after* runtime (runtime is when the function is called or used in processing).

Efficiencies gained ***before*** runtime include not having to write the code again or reinventing a process. While all code must be written the first time, after the testing is completed and the code is released, we try to put that baby to bed and walk away quietly. This is often the first (and only) answer I receive from students when asked, "how do we gain efficiency from pre-written code?" Efficiency in not having to re-write code may only represent 10% of all efficiencies gained.

Efficiencies gained ***during*** runtime are perhaps a bit obscure. When any code is submitted to be processed, it must first be compiled into a digital form that machines and processors can understand (appropriately called 'machine language'). This is called being 'compiled' and may take only a fraction of a second. Additionally, each piece of compiled code includes what is called an 'execution Plan' which acts like a navigator in a car and basically tells the query how to get to the destination where the data lives inside the database. Again, the savings per query may be only a second or two for simple tasks but may be significantly more for more complex code. If we build functions, these blocks of code are compiled the first time they are used and then rarely if ever need to be re-compiled again. This may not seem like a big deal but imagine a process (such logging-in to email) that occurs billions of times a day and then we can see that every fraction of a second matters.

Efficiencies gained ***after*** runtime are theoretical and only able to be estimated. These estimates include events and errors that *NEVER HAPPENED*. Think about it; How many people did not die because they never started smoking as teenagers? We can postulate an approximate number based on statistics, but we will never know in absolute terms. It is theoretical. The same is true when trying to measure the efficiencies gained from using pre-written code like functions. How many disasters did not happen because the code was pre-written, tested, validated and included proper error-handling? We will never know in absolute terms either. The best organized technology companies have very strict processes and rules regarding the use of repetitive code, no short cuts. The idea is to spend the extra time building something correctly the first time to avoid the pain and confusion of having to fix the same issue many more times in the future.

Let's explore the different kinds of functions included with most SQL systems, including the following:

- String
- Numeric
- NULL
- Aggregate
- Date

String Functions

Many (if not most) queries need a little bit of minor tweaks when trying to match the nuance of a business question. We have dozens of keywords that help adjust not only the asking of the question driving the query but also formatting the results set of data returned to be better able to be read and understood by people.

Please review the following incomplete list of most common string functions:

Name	Purpose
CONCAT	Returns a string that is the result of combining multiple strings together end-to-end
CAST	Changes the data type of a column during query execution
LEFT	Returns the characters from far LEFT of string (counting left to right)
LEN	Returns the number of characters in a string, ignoring blanks at the end
LOWER	Returns a string with all characters in LOWERCASE
LTRIM	Returns the characters remaining from a string after trimming leading blanks
REPLACE	Replaces all occurrences of specified string with a second specified replacement
RIGHT	Returns the characters from far RIGHT of string (counting right to left)
RTRIM	Returns the string after trimming trailing blanks
STR	Returns character data converted from numeric data
SUBSTRING	Chops out string when provided a beginning point and desired number of characters
TRIM	Removes specified characters from beginning and ending of string (often blank spaces)
UPPER	Returns a string with all characters in UPPERCASE

Figure 5.27: A list of common string functions

The above string functions are a subset of a larger list of string functions. These are considered basic and essential to most data scenarios for those being introduced to relational databases and SQL. A more exhaustive list beyond the introductory focus of this book can be found online at https://database.guide/sql-server-string-functions-full-list/.

CONCAT()

There are many times characters and results from queries need to be combined or concatenated for easier reading. This con be accomplished without the use of a function as follows using simple plus sign and quoted blank spaces:

```
SELECT 'Jimi' + ' ' + 'Hendrix' + ' ' + 'was a great musician' AS Homage_to_Jimi
```

	Homage_to_Jimi
1	Jimi Hendrix was a great musician

Figure 5.28: An example query combining characters without a function

CONCAT() happens to be a string function to make the process of combining strings together more programmatic and directly tied to a query result. See the following:

```
SELECT CONCAT('Jimi', ' ', 'Hendrix', ' ','was a great musician') AS
Homage_to_Jimi_2
```

	Homage_to_Jimi_2
1	Jimi Hendrix was a great musician

Figure 5.29: An example query combining characters with CONCAT()

Next, please see CONCAT() returning data from columns in tblDRIVER:

```
SELECT CONCAT(fname,' ', Lname, ' ', 'was a great musician') AS
Homage_to_Jimi_3
FROM tblDRIVER
      WHERE fname = 'Jimi'
```

	Homage_to_Jimi_3
1	Jimi Hendrix was a great musician

Figure 5.30: An example query combining characters pulled from a table with CONCAT()

LEFT() and RIGHT()

LEFT() is a function that can cut a specified number of characters from a string, reading from left to right. RIGHT() is going to be exactly opposite (backwards?) by cutting a specified number of characters from a string reading from right to left.

```
SELECT LEFT(CONCAT(fname, Lname), 6) AS Left_6_Characters
FROM tblDRIVER
```

	Left_6_Characters
1	JimiHe
2	BruceL
3	JimMor
4	MerylS

Figure 5.31: An example query with LEFT() and CONCAT()

```
SELECT RIGHT(CONCAT(fname, Lname), 6) AS Right_6_Characters
FROM tblDRIVER
```

	Right_6_Characters
1	endrix
2	uceLee
3	rrison
4	Streep

Figure 5.32: An example query with RIGHT() and CONCAT()

The results from Figure 5.32 may not be obviously useful at this point; When will anyone need to know the last 6 characters of a combined first and last name? I want readers to be aware of the capabilities of the most common functions so when the process of cleaning-up data obtained from a web scrape or a third-party vendor gets dropped on us we will be better prepared than I was!

LEN()

LEN() is a function that allows a user to determine the number of characters (or LENGTH) of a string. This knowledge can then help us as we programmatically extract data from the string with other functions such as LEFT() and RIGHT() among others we shall see shortly.

```
SELECT Fname, Lname, LEN(Lname) AS Number_Characters_Lname
FROM tblDRIVER
      WHERE fname = 'Bruce'
```

	Fname	Lname	Number_Characters_Lname
1	Bruce	Lee	3

Figure 5.33: An example query with LEN() to determine number of characters

The query in Figure 5.33 produces a result of 3 for the value of the aliased column (Number_Characters_Lname) when provided the fname value of 'Bruce'. This makes sense as the only driver with the first name of 'Bruce' is Bruce Lee. The number of characters in 'Lee' is in fact 3.

LOWER() and UPPER()

The next two functions are going to be a valuable part when presenting data to others as part of a formal dashboard or other reporting requirements. Frequently, data results will need to be displayed in a pattern of consistently upper or lower case.

```
SELECT LOWER(CONCAT(Fname,' ', Lname)) AS lower_Fname_Lname
FROM tblDRIVER
```

	lower_Fname_Lname
1	jimi hendrix
2	bruce lee
3	jim morrison
4	meryl streep

Figure 5.34: An example query with LOWER() to affect readability

```
SELECT UPPER(CONCAT(Fname,' ', Lname)) AS UPPER_Fname_Lname
FROM tblDRIVER
```

	UPPER_Fname_Lname
1	JIMI HENDRIX
2	BRUCE LEE
3	JIM MORRISON
4	MERYL STREEP

Figure 5.35: An example query with UPPER() to affect readability

TRIM(), LTRIM(), and RTRIM()

While scraping data from various locations on the web or because of extracting data from multiple resources, we may need to deal with unnecessary blank spaces either at the beginning or end of values. The string functions TRIM(), LTRIM(), and RTRIM() can cut out these blank spaces!

TRIM() will remove blanks both in front as well as at the end. See as follows with a string with an initial length of 30 characters (including the 2 blanks at the beginning and 2 more at the end):

```
SELECT TRIM('  2 blanks in front and back  ') AS TRIM_Gets_Front_and_Back
```

	TRIM_Gets_Front_and_Back
1	2 blanks in front and back

Figure 5.36: An example query with TRIM()

Do not simply take my word that TRIM() removed the empty spaces in the example found in Figure 5.36. Please see the following query that uses LEN() to measure the number of characters remaining after TRIM() runs:

```
SELECT LEN(TRIM('  2 blanks in front and back  ')) AS
TRIM_Gets_Front_and_Back
```

	Length_After_TRIM
1	26

Figure 5.37: An example query with TRIM() removing blank spaces in front and back

Next, let's see the use of both LTRIM() and RTRIM() with the same string and validation queries to drop the blank spaces one at a time

```
SELECT LTRIM('  2 blanks in front and back  ') AS LTRIM_Gets_Front
SELECT LEN(LTRIM('  2 blanks in front and back  ')) AS LTRIM_Gets_Front
```

	LTRIM_Gets_Front
1	2 blanks in front and back

	LTRIM_Gets_Front
1	28

Figure 5.38: An example query with LTRIM() removing blank spaces in front

Next, let us see a similar example using RTRIM() to drop blank characters from the end of a string and then validate the results with another query using LEN().

```
SELECT RTRIM('  2 blanks in front and back  ') AS RTRIM_Gets_Only_End

SELECT LEN(RTRIM('  2 blanks in front and back  ')) AS
Number_Of_Characters_from_30
```

	RTRIM_Gets_Only_End
1	2 blanks in front and back

	Number_Of_Characters_from_30
1	28

Figure 5.39: An example query with RTRIM() removing blank spaces in back

There may be a need for dropping leading characters that are not blank spaces. From METRO_TRANSIT, see the example in Figure 5.40 where the first two characters (which are '19') are dropped from each birthdate value in the results set.

```
SELECT Fname, Lname, BirthDate, LTRIM(BirthDate, 19) AS NonBlank_LTRIM
```

```
FROM tblDRIVER
```

	Fname	Lname	BirthDate	NonBlank_LTRIM
1	Jimi	Hendrix	1942-11-27	42-11-27
2	Bruce	Lee	1940-11-27	40-11-27
3	Jim	Morrison	1943-12-08	43-12-08
4	Meryl	Streep	1949-06-22	49-06-22

Figure 5.40: An example query with LTRIM() removing specific characters

Consider product data from a supplier or industry partner that we are trying to import to conduct analyses. Maybe all we really want are the names and prices of the products they sell to our organization, yet this external data has 'over-loaded' values with proprietary productid, or product category names mixed in the name values. LTRIM() might be the perfect function to drop leading characters that all have the same pattern.

REPLACE()

REPLACE() is a string function that substitutes one new string that overwrites another. It has the following syntax with 3 inputs:

```
REPLACE (full_string, substring_to_be_replaced, new_substring)
```

An example from METRO_TRANSIT includes re-writing the description for some values in tblNEIGHBORHOOD. Consider wanting to replace 'restaurant' with 'retail and award-winning restaurants' for the sake of a single query:

```
SELECT NeighborhoodDescr, REPLACE(NeighborhoodDescr,
'restaurants','retail and award-winning restaurants') AS
Updated_Changes
FROM tblNEIGHBORHOOD
```

	NeighborhoodDescr	Updated_Changes
1	Vibrant, colorful, and walkable neighborhood with many shops and services	Vibrant, colorful, and walkable neighborhood with many shops and services
2	Naturally scenic, tourist-focused neighborhood with many food options	Naturally scenic, tourist-focused neighborhood with many food options
3	Urban, business-focused with many restaurants and transit connections to entire region	Urban, business-focused with many retail and award-winning restaurants and transit connections to entire region
4	Gritty and industrial neighborhood	Gritty and industrial neighborhood
5	Eccentric and organic feel with shops and restaurants that fit a range of budgets	Eccentric and organic feel with shops and retail and award-winning restaurants that fit a range of budgets

Figure 5.41: An example query with REPLACE() swapping a string with a second string

The values returned in Figure 5.41 include only 2 rows that were modified with the REPLACE() function (rows 3 and 5). The unaffected values are those that did not have the substring 'restaurants' already.

Please note, the data stored in the column NeighborhoodDescr in the table tblNEIGHBORHOOD are NOT permanently changed; the REPLACE() function only returns changes for a single query. We would need to execute an UPDATE command to make permanent changes to the database.

STR()

STR() is a function that takes a number value and makes it a string. This conversion helps integrate the data with other string data, perhaps a longer document where the numbered data is being presented in its final form and no longer is going to be involved in math.

The STR() function has several inputs including the following:

- Number (required)
- Length (optional)
- Decimals (optional)

The STR() function is flexible but be cautious when using; this function will round to the nearest whole number if the decimals option is not specified.

```
SELECT Fname, Lname, BoardingDateTime, Fare, STR(Fare) AS
Amount_Fare_Default_STR
FROM tblBOARDING B
      JOIN tblCUSTOMER C ON B.CustomerID = C.CustomerID
```

	Fname	Lname	BoardingDateTime	Fare	Amount_Fare_Default_STR
1	Ivey	Hazekamp	2025-04-23 07:41:35.000	3.75	4
2	Kenyetta	Terron	2025-04-24 06:47:23.000	3.25	3
3	Ivey	Hazekamp	2025-04-24 06:47:23.000	3.75	4

Figure 5.42: An example query with STR() returning numeric data as string

The last column in Figure 5.42 is a string value drawn from the numeric data type. Even though the value has 2 decimal places (as seen in the second to last column 'Fare'), the STR() default rounds to the nearest whole number.

The following optional inputs will provide greater accuracy with STR():

```
SELECT Fname, Lname, BoardingDateTime, Fare, STR(Fare, 8, 2) AS
Amount_Fare_STR_Decimals
FROM tblBOARDING B
      JOIN tblCUSTOMER C ON B.CustomerID = C.CustomerID
```

	Fname	Lname	BoardingDateTime	Fare	Amount_Fare_STR_Decimals
1	Ivey	Hazekamp	2025-04-23 07:41:35.000	3.75	3.75
2	Kenyetta	Terron	2025-04-24 06:47:23.000	3.25	3.25
3	Ivey	Hazekamp	2025-04-24 06:47:23.000	3.75	3.75

Figure 5.43: An example query with STR() including input options

SUBSTRING()

One of the most powerful and important string functions must be SUBSTRING(). This function enables an analyst to take any string value and extract a portion (hence the name 'sub-string'). This takes several inputs including the entire string, the starting point of the extraction, and the total number of characters to extract.

The syntax of substring() is as follows:

SUBSTRING(string, starting position, length)

```
SELECT SUBSTRING('this is an example text', 6, 11) AS SUBSTRING_EXAMPLE
```

	SUBSTRING_EXAMPLE
1	is an examp

Figure 5.44: An example query with SUBSTRING()

In a typical organization, there is a frequent need to extract data from string values. Consider trying to clean up a block of text from a spreadsheet or large text file where a set length of data is buried within other undesirable text. Having a command like substring() will make the process of data analyses much more convenient.

CAST()

Many times, when working with data we will want to present the data in reports or a dashboard with a different data type from how the data is stored in the database. The function that is best used for this task is CAST() (technically CAST() is not a string function but it is included in this section as it will be required in future sections).

The most common switches in data types are changing numbers and character/string (in both directions), as well as modifying date and time formats. Without affecting the data stored within the database, we can change how the data is returned or presented on the fly during a query execution by using CAST().

The syntax for this function is as follows:

CAST (expression **AS** target_type [(length)])

Consider wanting to change the data type of tblDRIVER.salary from NUMERIC(8,2) to INTEGER for aesthetic reasons like dropping decimal places.

```
SELECT fname, Lname, Salary, CAST(Salary AS INTEGER) AS Salary_INT
FROM tblDRIVER
```

	fname	Lname	Salary	Salary_INT
1	Jimi	Hendrix	42590.00	42590
2	Bruce	Lee	42590.00	42590
3	Jim	Morrison	47625.00	47625
4	Meryl	Streep	46965.00	46965

Figure 5.45: An example query with CAST() changing data type NUMERIC to INTEGER

Other data types are frequently changed during queries, like dates to character.

```
SELECT fname, Lname, BirthDate, CAST(BirthDate AS varchar(10)) AS
BIRTH_VARCHAR
FROM tblDRIVER
```

	fname	Lname	BirthDate	BIRTH_VARCHAR
1	Jimi	Hendrix	1942-11-27	1942-11-27
2	Bruce	Lee	1940-11-27	1940-11-27
3	Jim	Morrison	1943-12-08	1943-12-08
4	Meryl	Streep	1949-06-22	1949-06-22

Figure 5.46: An example query with CAST() changing data type DATE to VARCHAR(10)

Figure 5.46 shows a query result with the column Birthdate as both a date data type as well as cast to varchar(10). These results are indistinguishable! We are better able to integrate character data types with other strings when aiming for a consistent format or readability.

NUMERIC FUNCTIONS

The following functions are some of the more common when working with numerical data:

Name	Purpose
ABS	Determines the absolute value of a number
CEILING	Determines the smallest integer value greater than or equal to a number
FLOOR	Determines the largest integer value less than or equal to a number
MOD	Determines Modulo or Remainder
ISNUMERIC	Tests whether an expression is numeric
POWER	Returns the value of a number raised to the power of another number
RAND	Returns a random number that is between 0 and 1 (16 digits)
ROUND	Rounds a number to a specified number of decimal places
SQRT	Returns the square root of a number
SQUARE	Returns the square of a number

Figure 5.47: A list of 10 basic numeric functions in SQL

Let's take a closer look at the more basic numeric functions from Figure 5.47. As with many languages, there are more numeric functions than are listed here, as the goal of this book is to provide a foundational understanding of SQL and relational databases. As your skills and interest grows with analytics and SQL, there will be opportunities to expand your knowledge beyond the basic introduction of functions found here.

Please feel free to type along with these examples and experiment with your own data to get a feel for how these are used and the results generated.

ABS()

This function is used in evaluations often when seeking to measure distance in variation from an expected value (without regard to positive or negative).

```
SELECT ABS (-43.87) AS Example_ABS
```

	Example_ABS
1	43.87

```
SELECT ABS (54 * -43.87) AS Example2_ABS
```

	Example2_ABS
1	2368.98

Figure 5.48: Example queries using numeric function ABS (Absolute Value)

CEILING and FLOOR

Ceiling() and FLOOR() are great functions to be familiar with as they simplify a result set by returning the whole number that is either larger or smaller than the calculated results. CEILING() will round UP to the nearest whole number while FLOOR() will round DOWN.

```
SELECT CEILING(26.768 * 13.923) AS CEILING_BasicMath

SELECT FLOOR(26.768 * 13.923) AS FLOOR_BasicMath
```

	CEILING_BasicMath
1	373

	FLOOR_BasicMath
1	372

Figure 5.49: Example queries using numeric functions CEILING() and FLOOR()

These functions are consistent with negative results as well. Change the sign on one of the numbers in the equations and review the results:

```
SELECT CEILING(26.768 * -13.923) AS CEILING_BasicMath_Negative

SELECT FLOOR(26.768 * -13.923) AS FLOOR_BasicMath_Negative
```

	CEILING_BasicMath_Negative
1	-372

	FLOOR_BasicMath_Negative
1	-373

Figure 5.50: Example queries using numeric functions CEILING() and FLOOR()

We will see aggregate functions in greater detail a moment, however, please consider that AVG() will calculate the average across many values as follows:

```
SELECT AVG(SALARY) AS AVG_Salary
FROM tblDRIVER
```

	AVG_Salary
1	44942.500000

Figure 5.51: Example query showing results from AVG() aggregate function

Next, please see that CEILING() and FLOOR() can be layered over AVG() to round the results to the nearest whole number:

```
SELECT CEILING(AVG(SALARY)) AS CEILING_AVG_Salary
FROM tblDRIVER
```

	CEILING_AVG_Salary
1	44943

```
SELECT FLOOR(AVG(SALARY)) AS FLOOR_AVG_Salary
FROM tblDRIVER
```

	FLOOR_AVG_Salary
1	44942

Figure 5.52: Example queries showing results from CEILING() and FLOOR() over AVG()

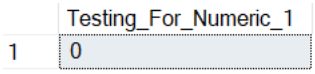
Both CEILING() and FLOOR() are great functions to be familiar with when investigating a set of data. These can help quickly establish the boundaries of the range of values contained in a column with a numeric data type as well as gaining a comprehensive understanding of an entire data ecosystem when applied in conjunction with other functions and formulas.

ISNUMERIC()

Occasionally when working with a vast amount of tabular data, there will be a need to determine whether a column (or a value) contains numeric data. The good news is SQL has a function to determine this! ISNUMERIC() will return a value of '1' (positive) if the input parameter it evaluates is a numeric value. Conversely, ISNUMERIC() will return a value of '0' (negative) if the input parameter it evaluates is NOT a numeric value.

The first example will be against a string as follows:

```
SELECT ISNUMERIC(lname) AS Testing_For_Numeric_1
FROM tblDRIVER
      WHERE Fname = 'Meryl'
```



	Testing_For_Numeric_1
1	0

Figure 5.53: Example query using ISNUMERIC()

There should be no surprise from the resulting value of '0' from Figure 5.53 as the string value 'Streep' is evaluated as the input to the function is not numeric.

Let us try passing other inputs, such as salary from the same table in addition to an obvious math equation:

```
SELECT ISNUMERIC(salary) AS Testing_For_Numeric_2
FROM tblDRIVER
WHERE Fname = 'meryl'
```

	Testing_For_Numeric_2
1	1

Figure 5.54: Example query using ISNUMERIC() against column 'salary'

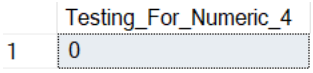
```
SELECT ISNUMERIC(23.67 * 32) AS Testing_For_Numeric_3
```

	Testing_For_Numeric_3
1	1

Figure 5.55: Example query using ISNUMERIC() against a math equation

The third example shown in Figure 5.55 above is against a math equation and returns the value of 1. What should be returned if the exact same equation is placed between quotes?

```
SELECT ISNUMERIC('23.67 * 32') AS Testing_For_Numeric_4
```



	Testing_For_Numeric_4
1	0

Figure 5.56: Example query using ISNUMERIC() against a string of math equation

The fourth example shown in Figure 5.56 above returns a value of 0 because once the quotes are placed around the math equation, it becomes a string.

ISNUMERIC() is quite useful when trying to quickly understand the results of a data set and whether other math functions are going to be effective. We will see how to process thousands (if not millions) of rows automatically later in our learning. Being able to have conditional logic that decides to include a value in math without people being involved will prove to be very important.

MOD() and %

There are situations when working with data that we may need to determine the remaining value after an equation that includes division. This is known as the remainder or modulus. Many database products that comply with standard SQL syntax use a function MOD() to determine the modulus. SQL Server does not use the MOD() function and instead uses the single percentage sign ('%') to perform the same functionality as MOD(). Please see the following syntax that divides 49 by 32 and returns the remaining balance:

```
SELECT 49 % 32 AS Example_Modulus
```

	Example_Modulus
1	17

Figure 5.57: Example query showing results from division using modulus %

Many of us should be able to conduct simple math in Figure 5.57 in our head to see the result after dividing 49 by 32 is 17. A common need for finding the modulus is when we want to determine whether a value is even or odd; when the base number is divided by 2, even numbers will have a remainder/modulus = 0 while any odd number will have the value of 1.

SQUARE(), SQRT() and POWER()

Working with data in many forms occasionally requires us to calculate any number of exponential values when provided a base number. The most common are the square, square root and a raised power (the number of times to multiply the base number to itself). Standard SQL functions are provided to address these concerns, including SQUARE(), SQRT() and POWER(). Consider the following examples:

```
SELECT SQUARE(5) AS Example_SQUARE
```

	Example_SQUARE
1	25

Figure 5.58: Example query showing results from using SQUARE(5)

Figure 5.58 provides an example use of SQUARE() by returning the value 25 (which is the result of 5 * 5 or the base value of 5 squared). The SQUARE() function also works with decimals! See the following example and try other numbers for yourself:

```
SELECT SQUARE(5.44) AS Example_SQUARE_with_Decimals
```

	Example_SQUARE_with_Decimals
1	29.5936

Figure 5.59: Example query showing results of SQUARE() with input number decimals

Another important exponential command that is common when working with data includes square root or SQRT(). This function takes one input parameter (which can include decimals) and then returns the square root as seen below:

```
SELECT SQRT(5) AS Example_SQUARE_ROOT
```

	Example_SQUARE_ROOT_with_decimals
1	2.23606797749979

Figure 5.60: Example query showing results from using SQRT(5)

One last exponential command in SQL that is useful is POWER(). This function takes two input parameters, including the base number as well as the desired exponent:

```
SELECT POWER(3, 4) AS Example_POWER
```

	Example_POWER
1	81

Figure 5.61: Example query showing results from POWER(3,4)

The function POWER() provides significant math ability inside a programming environment while reading and processing potentially millions of rows of data.

Being able to program the calculation of complex math on data without people having to do the work is incredible for two big reasons:

1) People are slow by factors of millions compared to machines
2) People make mistakes!

RAND()

There are situations when testing code, like stored procedures, where we will be better able to manage repetitive executions by generating a random 16-digit number between 0 and 1. SQL provides a numeric function RAND() to address this need.

While there will be an entire chapter devoted to testing database objects and something called synthetic transactions in an advanced book on data analytics, these topics are beyond the scope of this first book introducing the foundational commands of SQL. Be comfortable in seeing how to generate a random number at this point in your learning and know that we will have more context with additional use cases in the future.

```
SELECT RAND() AS Random_16_digit_Number
```

	Random_16_digit_Number
1	0.0321877495777032

Figure 5.62: Example query showing 16-digit results from RAND() between 0 and 1

As we will see later, there are ways to use RAND() to find random primary key values to automatically call stored procedures and test the effectiveness of an application and infrastructure platform. This eventually saves us time and money. For now, let's just see how the basic use of RAND() works. Try this code on your own as many times; there will never be the same number twice!

The following examples generate random numbers larger than 0:

```
SELECT RAND() * 10000 AS Random_16_digit_Number
```

	Random_16_digit_Number
1	1880.43389938162

Figure 5.63: Example query showing 16-digit results from RAND() greater than 0

The first example in Figure 5.63 generates a random number that is still 16 digits in length that is larger than 0. All that has really occurred here is moving the decimal place.

Next, let's see a cleaner method without any decimal places by using the CAST() function to make the output a whole number:

```
SELECT CAST(RAND() * 10000 AS INT) AS Random_Number_Larger_Than_1
```

	Random_Number_Larger_Than_1
1	8567

Figure 5.64: Example query showing results from RAND() with a 3- or 4-digit number

ROUND()

ROUND() will be the last numeric function found in SQL that we will explore in this section. Simply, this function provides the ability to determine the number of decimals that are rounded on numeric value. Let us see two examples:

```
SELECT ROUND(RAND() * 10000, 3) AS ROUND_Random_Number_3_decimals
```

	ROUND_Random_Number_3_decimals
1	1905.259

Figure 5.65: Example query showing results from ROUND() to 3 decimals

The results in Figure 5.65 would normally return to 16 digits, however, the ROUND() function rounds the results to a specified number of digits.

```
SELECT ROUND(23.9834, 0) AS Example_ROUND
```

	Example_ROUND
1	24.0000

Figure 5.65: Example query showing results from ROUND() to 0 decimals

Please note that ROUND() will in fact round up or down to the specified number of decimals, however, there will still be the total number of decimals in the output as was in the calculation. If there is an interest in dropping the decimals entirely at a certain place, then we will need to use CAST() again.

```
SELECT CAST(ROUND(23.9834, 0) AS INT) AS Example_CAST_ROUND
```

	Example_CAST_ROUND
1	24

Figure 5.66: Example query showing results from ROUND()

NULL Functions

Remember please that NULL is a value that is unknown; it does not equal zero or any other value (not even another NULL).

Standard SQL includes several functions that allow users to affect how the system treats empty, missing, or otherwise unknown values. These are known as NULL functions and include the following:

- ISNULL()
- COALESCE()

- IFNULL()
- NVL()

Of the four NULL functions listed here, only ISNULL() and COALESCE() work with SQL Server and the METRO_TRANSIT example we have constructed. The other two are listed in case you are working with another relational database management system like Oracle or MySQL. Let us see the use of ISNULL() and COALESCE() as they are effectively identical.

```
SELECT ISNULL(NULL, 'Mary') AS Example_ISNULL
```

	Example_ISNULL
1	Mary

```
SELECT COALESCE(NULL, 'had a little lamb') AS Example_COALESCE
```

	Example_COALESCE
1	had a little lamb

Figure 5.67: Example query showing results from ISNULL() and COALESCE()

Please note the output of the NULL functions in Figure 5.67 are interchangeable and provide the same functionality.

One last example of these NULL functions can be seen with revisiting an example from earlier (Figure 5.18) where we were drawing data on tblROUTE_TYPE while demonstrating the use of LEFT JOIN:

```
SELECT RT.RouteTypeName, R.RouteName
FROM tblROUTE_TYPE RT
      LEFT JOIN tblROUTE R ON RT.RouteTypeID = R.RouteTypeID
```

Results Messages

	RouteTypeName	RouteName
1	Express	Sodo-Downtown Express
2	Express	Fremont-Downtown Express
3	Special Event	Fremont-Waterfront-Downtown
4	Regular	Sodo-Downtown Commuter
5	Regular	Fremont-Downtown Commuter
6	Regular	Capitol Hill-Downtown
7	Inclement Weather/Reduced Service	NULL
8	Waterfront Bypass	NULL

Re-writing this query slightly to include different words as opposed to NULL may look like the following:

```
SELECT RT.RouteTypeName, ISNULL(R.RouteName, 'No Route currently
assigned') AS RouteName
FROM tblROUTE_TYPE RT
LEFT JOIN tblROUTE R ON RT.RouteTypeID = R.RouteTypeID
```

	RouteTypeName	RouteName
1	Express	Sodo-Downtown Express
2	Express	Fremont-Downtown Express
3	Special Event	Fremont-Waterfront-Downtown
4	Regular	Sodo-Downtown
5	Regular	Fremont-Downtown
6	Regular	Capitol Hill-Downtown
7	Inclement Weather/Reduced Service	No Route currently assigned
8	Waterfront Bypass	No Route currently assigned

Figure 5.68: Example query showing results from ISNULL() when applied to previous query from Figure 5.18 earlier in chapter

Please note in Figure 5.68 that the column name RouteName has to be replaced with an alias once the function ISNULL() is applied. The values in rows 7 and 8 are originally NULL, but are substituted with the input values 'No Route currently assigned' passed into the function ISNULL().

AGGREGATE FUNCTIONS

Aggregate functions are very powerful and save an incredible amount of time when conducting data analyses. These should become very familiar and trusted in your toolbox of SQL chops. Some of these are self-explanatory, but it makes sense to introduce the context and best use for each as some common mistakes occur with beginning analysts.

For the time being, these functions will be demonstrated without GROUP BY and HAVING clauses. After first understanding how these functions behave, then we can expand their use and include the broader application of GROUP BY.

While there are roughly ten functions that are labeled as aggregate functions, the following are the most common within SQL:

- AVG()
- COUNT()
- MAX()
- MIN()
- SUM()

This section of the textbook will explore the five most common aggregate functions; later, in the advanced SQL chapter, we will include additional aggregate functions related to data analysis as we conduct a deeper dive with additional knowledge.

AVG()

There are many times we desire to determine the average value of some items while analyzing data. For example, we may want to become aware of the average income, number of years of work experience, or age of people. The average rating of customer satisfaction or reliability is important when shopping for cars. When we become aware of the average value, we can then evaluate the relative position of other values as being either 'above or below average'. While important in the world of statistics, average is just one calculation and should be considered suspect.

By itself, any value of average can often be misinterpreted if the sample is insufficient in size, is subject to significant variance or has uninvestigated outliers. That said, AVG() is a critical aggregate function; let's give it a try inside the METRO_TRANSIT database!

```
SELECT AVG(Salary) AS Average_Salary
FROM tblDRIVER
```

	Average_Salary
1	44942.500000

Figure 5.69: A query result from aggregate function AVG()

Please note in the above query returns a value of 44942.500000. Let us update our query quickly to fix it!

```
SELECT CAST(AVG(Salary) AS Numeric(8,2)) AS Average_DriverSalary
FROM tblDRIVER
```

Results Messages

	Average_DriverSalary
1	44942.50

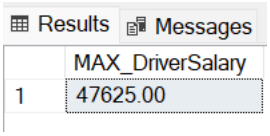
Figure 5.70: A query result from aggregate function AVG() with CAST() and an alias

The results from the query now have a label as well as the number of decimal spaces has been reduced.

MAX()

What are the boundaries of any value? This is important when investigating data in a database for the first time. Instead of returning the average driver salary, let us see the most. We can use the aggregate function MAX() for this.

```
SELECT CAST(MAX(Salary) AS Numeric(8,2)) AS MAX_DriverSalary
FROM tblDRIVER
```



Results Messages

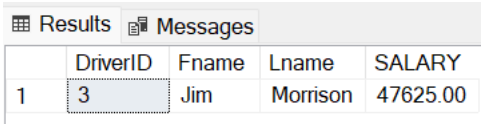
	MAX_DriverSalary
1	47625.00

Figure 5.71: A query result from aggregate function MAX() and CAST()

Once again, be comfortable changing the results with the CAST() function to reduce the number of decimal places as well as using an alias to add a column header.

While we are here, let us make a quick side query and see if we can leverage the MAX() function to find the driver(s) that earn this salary. How might we write this?

```
SELECT DriverID, Fname, Lname, CAST(Salary AS Numeric(8,2)) AS SALARY
FROM tblDRIVER
WHERE Salary = (SELECT MAX(Salary) FROM tblDRIVER)
```



Results Messages

	DriverID	Fname	Lname	SALARY
1	3	Jim	Morrison	47625.00

Figure 5.72: A query result from aggregate function MAX() and CAST() and subquery

Be careful! Many beginning SQL query-writers do not see this question as including a subquery; often beginners want to have MAX(Salary) in the SELECT line and do not get the results they seek. Get comfortable obtaining the value we seek to include in a logical filter through a calculation by writing an entirely separate query and then sticking it inside the other query (more on subqueries is coming up).

MIN()

The MIN() aggregate function is obviously very similar to what we have just seen with MAX(). Let us see a few more examples:

```
SELECT MIN(Fare) AS Smallest_Fare
FROM tblBOARDING
WHERE BoardingDateTime > 'August 5, 2018'
```

	Smallest_Fare
1	3.25

Figure 5.73: A query result from aggregate function MIN()

Aggregate functions can also be used together to do additional calculations. Consider the following query that determines the difference between the highest (MAX()) and lowest (MIN()) fares collected during the month of June 2025:

```
SELECT MAX(fare) AS Highest, MIN(fare) AS Lowest,(SELECT MAX(fare)-
MIN(fare)) AS Fare_Difference
FROM tblBOARDING
WHERE BoardingDateTime > 'June 1, 2025'
      AND BoardingDateTime < 'July 1, 2025'
```

	Highest	Lowest	Fare_Difference
1	3.75	3.25	0.50

Figure 5.74: A query result of multiple aggregate functions MAX(), MIN() with calculation

Aggregate functions MAX() and MIN() can also work on other data types besides numeric, including varchar and dates columns.

```
SELECT MAX(BirthDate) as Youngest_DriverBirthDate
FROM tblDRIVER
```

Results Messages

	Youngest_DriverBirthDate
1	1949-06-22

Figure 5.75: A query result from aggregate function MAX() against a date column

Please note how the largest/maximum value for a birthdate returns the YOUNGEST person! This is because the largest birthdate value according to the database logic is closest to present day, thus yielding the shortest lifespan.

The MIN() and MAX() aggregate functions can also be run against a datatype of varchar, such as tblDRIVER.lname.

```
SELECT MIN(Lname) AS Min_Lname
FROM tblDRIVER
```

Results Messages

	Min_Lname
1	Hendrix

Figure 5.76: A query result from aggregate function MIN() against a character column

By itself, this query result does not provide much learning opportunity. It can be argued that it might be cool to know that MIN()/MAX() functions can be run against character data

types, the questions we are trying to address may be better answered in other ways. Please see the following query that may provide better understanding of relative ages of drivers:

```
SELECT TOP 5 D.DriverID, D.Fname, D.Lname, D.BirthDate
FROM tblDRIVER
ORDER BY BirthDate DESC
```

Results Messages

	DriverID	Fname	Lname	BirthDate
1	4	Meryl	Streep	1949-06-22
2	3	Jim	Morrison	1943-12-08
3	1	Jimi	Hendrix	1942-11-27
4	2	Bruce	Lee	1940-11-27

Figure 5.77: A query result using TOP command and ORDER BY clause

Please note that data on all drivers are returned in a sorted order of youngest to oldest using the ORDER BY clause. This *may* provide a better answer than trying to use either MIN() or MAX() against a birthdate.

COUNT()

The aggregate function COUNT() may be the most frequently used aggregate function. Being able to know how many rows match a certain condition is the guts of what writing a query with SQL is all about, especially when conducting a high-level initial investigation.

There are several quick concerns to be aware of when writing a query where we seek to know the number of matching rows using the aggregate function COUNT():

- Do NOT confuse COUNT() with SUM()
- Use the keyword DISTINCT when the query JOINs a transactional table

COUNT() versus SUM()

Many times, in introductory database classes, my students will make a mistake and try to 'count' the results of columns containing money values when the proper aggregate function is SUM(). This mistake is fixed early in an engineer's career (usually right after a failing grade on their first midterm). Also, we can use COUNT() with a specific input parameter either a primary key or a foreign key in the parentheses, as follows:

```
SELECT COUNT(TripID) AS NumberOfTrips
FROM tblTRIP
```

	NumberOfTrips
1	5

Figure 5.78: Query with COUNT() aggregate function with *primary key* as input parameter

```
SELECT COUNT(T.VehicleID) AS NumberOfTrips
FROM tblVEHICLE V
      JOIN tblTRIP T ON V.VehicleID = T.VehicleID
WHERE V.VehicleName = 'GS204'
```

	NumberOfTrips
1	3

Figure 5.79: Query with COUNT() aggregate function with *foreign key* as input parameter

The results in Figure 5.78 and Figure 5.79 are based on different questions. The first returns the result for the question 'how many trips have occurred?' The second query narrows the first question and filters on the foreign key value in tblTRIP to find the number of trips assigned to just a single vehicle, which has the name of 'GS204'. Many times, analysts are seeking the results of a filtered query based on the results of another previous query, narrowing each subsequent query as more is learned.

While having a PK or FK as input parameters is more precise, they can be replaced with the asterisk ('*') without losing functionality (go ahead and try it!).

```
SELECT COUNT(*) AS NumberOfTrips
FROM tblTRIP
```

	NumberOfTrips
1	5

Figure 5.80: Query with COUNT() aggregate function with *asterisk(*)* as input parameter

Please note that the substitution of the asterisk ('*') as an input parameter for the aggregate function queries does not change the results. This is often used during informal investigations where the analyst is conducting a quick perusal of data and not sticking to explicit reporting nomenclature.

SUM()

Whenever we are 'doing math', we need to read the values inside of each cell as opposed to just counting rows that match the filtering criteria we specify in the WHERE clause. Examples may include determining the amount of money a single passenger has spent on boarding fares, the total amount of money spent on vehicles of a certain type, as well as annual salaries for all drivers. First, in Figure 5.81, we can remind ourselves of the simplified METRO_TRANSIT diagram.

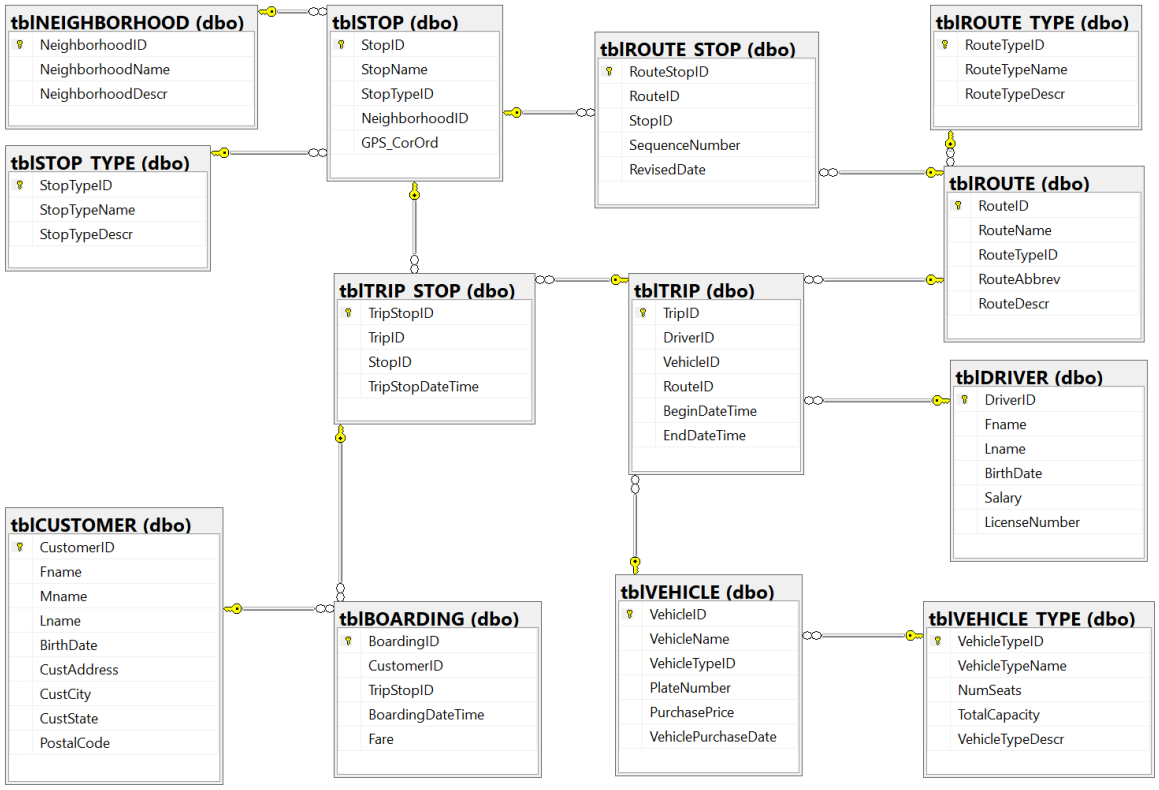


Figure 5.81: Reminder of simplified Entity-Relationship Diagram of METRO_TRANSIT

From the diagram above in Figure 5.81, the following queries are posed as example challenges to incorporate aggregate function of SUM():

1) Write the SQL to determine the total boarding fares from customer Ivey Hazekamp.

```
SELECT SUM(Fare) AS TotalFareAmount
FROM tblCUSTOMER C
      JOIN tblBOARDING B ON C.CustomerID = B.CustomerID
WHERE Fname = 'Ivey'
      AND Lname = 'Hazekamp'
```

	TotalFareAmount
1	7.50

2) Write the SQL code to determine the total amount of money spent on vehicles of type 'Double-Decker Bus'.

```
SELECT SUM(PurchasePrice) AS TotalExpenditure
FROM tblVEHICLE V
      JOIN tblVEHICLE_TYPE VT ON V.VehicleTypeID = VT.VehicleTypeID
WHERE VT.VehicleTypeName = 'Double-Decker Bus'
```

	TotalExpenditure
1	1775066.00

3) Write the SQL code to determine the total annual salaries for all drivers.

```
SELECT SUM(Salary) AS TotalSalaries
FROM tblDRIVER
```

	TotalSalaries
1	179770.00

Figure 5.82: Example queries using SUM() aggregate function

The above queries and results in Figure 5.82 are simple as they are aggregating a column from a single column. We will see in a moment how to expand this function and apply the function against many rows.

Common mistake with SUM()

Occasionally students will confuse COUNT() and SUM() when being introduced to aggregate functions for the first time. This mistake is passing the primary key value as the parameter to be included in the math equation, as we may get comfortable passing in primary key values with COUNT() and think these are interchangeable.

Consider the following 'mistake' query:

```
SELECT SUM(DriverID) AS TotalMistake
FROM tblDRIVER
```

	TotalMistake
1	10

Figure 5.83: Example mistake(!!) query of SUM() function against a primary key value

The math equation of adding unique identifiers together may not seem like a mistake at this moment (especially if these functions are new for you). Consider in slow-motion the act of adding these values together; even though a value has been returned by the database system, does the value (in this case '10') even make sense? This result is from adding up each unique identifier from the primary key column (1 + 2 + 3 + 4):

DriverID	Fname	Lname
1	Jimi	Hendrix
2	Bruce	Lee
3	Jim	Morrison
4	Meryl	Streep

Figure 5.84: Primary key and name values of all drivers showing common mistaken use of SUM() function against a primary key value

This accident has no business value in the same way that adding every driver's phone numbers or home house number together; just because the system returns a value does not alleviate the mistake! Avoid using SUM() against primary key or foreign key values.

Other Aggregate Functions

There are other SQL commands that are considered aggregate functions; however, they have limited use in the scope of this textbook. Please be aware the following aggregate functions exist and explore them in further detail on your own if you see fit:

- BINARY_CHECKSUM()
- CHECKSUM()
- CHECKSUM_AGG()
- GROUPING()
- GROUPING_ID()

Great job getting through the material here in Chapter 6: *Intermediate SQL*! This content is considered difficult for many learners and getting this far is a very strong indicator of your dedication and focus.

Post-Chapter Challenges
Based on your objectives and intentions on learnings from this book, please approach the following challenges as appropriate.
Track 1 (THINK): Data Tourist Seeking Ancillary Awareness
Please spend a total of 10 minutes reviewing the following questions, exploring your thoughts. These questions and your responses cut to the essence of this chapter:

- Which of the shortcuts and functions intended to simplify the coding process resonate with you after finishing the reading? Why?

- Consider the ability you now possess to write sophisticated and complex queries by adding the JOIN statement, aggregate functions, and GROUP BY command. JOIN allows you the ability to draw data from across an unlimited number of entities and do math against millions of rows of data! How powerful is that?

- How might data analytical skills with SQL allow you to be more effective, impactful, or otherwise 'do a better job' for any position you may pursue as a career?

Track 2 (WRITE): Dedicated Student or Recent Graduate
Based on your completion of this chapter, WRITE several paragraphs in response to each question; explore these as if you are being asked a similar question during a job interview!

- Write the INSERT statements to populate another 5 rows of data into the following 'look-up' entities:
    - tblCUSTOMER
    - tblEMPLOYEE
    - tblDRIVER
    - tblROUTE
    - tblVEHICLE

- Write the INSERT statements to populate another 10 rows of data into the following transactional entities:
    - tblTRIP
    - tblTRIP_STOP
    - tblBOARDING

Writing INSERT statements to populate transactional entities can be considered advanced! It is important to see the struggle of managing multiple foreign key values (we will eventually learn how to do this with automated code and 'stored procedures'. For now try to get data in with INSERT statements.

Track 3 (BUILD): Full Speed Learner Seeking Job
This track targets readers of this book who want to develop professional skills working with data to launch a career or obtain a more satisfying job.

- Take the database design that you chose at the conclusion of chapter 5 Beginning SQL and write the INSERT statements to populate all the entities with at least 5 rows of data. Save your script!

- Try to include appropriate data types, proper PK/FK functionality (surrogate keys with auto increment feature).

If you can get the original database established (designed, normalized, coded, and populated) you will have joined the top 5% of developers in the world! Very few professionals have built a database application from scratch; take command of your career!

Conclusion

The basic query structures used to draw data out of a database can be a difficult pattern of words if we try to simply memorize them. There is the basic sequence of 'SELECT, FROM and WHERE' and then dozens of keywords called functions. Approaching this data retrieval language as a method of learning may make the onboarding process a bit easier.

Learning in the modern economy involves being able to analyze data, often found in massive relational databases. The basic language used by millions of companies is SQL; being comfortable writing intermediate queries across dozens of tables is a valuable skill that will allow you to become more independent as an analyst and critical thinker pursuing innovation and opportunities for improving efficiency and optimization across an organization.

You have done a fantastic job so far getting through the first 6 chapters and on-boarding SQL beyond a basic level of competency! Do not slow down just yet as there is a small amount of additional learning required with SQL. The next chapter is #7 Intermediate SQL Part 2, which includes ground-changing skills of GROUP BY, date functions, and subqueries in greater detail. Keep rolling!