Chapter 5
BEGINNING SQL

Introduction

The Structured Query Language (shortened to SQL and pronounced either by the letters 'S-Q-L' or as a word 'sequel') is the standard query and programming language that is leveraged to create databases and objects used in the database (such as tables, logins, stored procedures, and user-defined functions among others). With SQL, there is an opportunity to automate most processes that create or consume data, such as point-of-sale purchases as well as publishing reports or updating a dashboard. These are sophisticated, powerful, advanced tasks but are easy enough to learn by most college students or any learner with focus and ambition.

If you have at least a passing understanding of SQL, we won't have to spend time at all on the development history of this language. Just understand that SQL is very stable with very few substantial changes for the past 50 years (how many front-end programming languages have come and gone since the mid-1970s?). As a Fourth Generation Language (4GL), SQL is refined and has many standard keywords that simplify many tasks. This enables users to focus on the immediate analytical objectives without the stress of designing 'how' to accomplish them. Have comfort in knowing SQL is the bedrock of working with data now and for the foreseeable future.

Again, being proficient in leveraging SQL is an exceptionally valuable technical skill for nearly every data-centric position. Many learners will have a bit of discomfort as we are confronted with challenges; learning a new querying and programming language will certainly have this. Keep an open mind as we explore design structures of storing data for machines and not the hierarchies that humans have organized with paper-based systems for centuries. This may seem like a revolutionary way of thinking.

Structure

By the end of this chapter, readers will:

- Become familiar with a development environment to write and execute original SQL commands and directly engage a copy of METRO_TRANSIT

- Understand the purpose of the Structured Query Language including the difference between Data Definition Language (DDL) and Data Modification Language (DML)
- Apply your understanding of basic SQL to code a copy of a database schema
- Create critical aspects of each table, including several data types, primary keys (PK), foreign keys (FK), and the auto-increment feature for PK values
- Become familiar with over a dozen basic SQL operators, commands, and short-cuts to begin assembling more sophisticated queries.

Working with SQL is a remarkable opportunity; being able to explore data with seemingly endless boundaries may become a life-long journey if not obsession. In our modern economy, essentially every organization needs a bevy of highly capable data professionals to help the organization learn, grow, as well as simply maintain operations.

This journey begins with building a new database for METRO_TRANSIT! We will code a simplified version of this database based on the work we did in chapters 2 and 3 after which we will continue with the steps to populate the look-up tables with real values. Later, we will construct stored procedures to manage new rows that represent the events that are called transactions. After creating a million rows or so of simulated activity, we will be ready to write queries. This arc of building a database provides the backdrop of hands-on learning that is missing for most aspiring data professionals.

We start by separating the language to construct objects from the language to insert and query data in a relational database. These two divisions are Data Definition Language (DDL) and Data Modification (or Manipulation) Language (DML). Let's begin with DDL if only to build a very simple set of tables to establish a practice database.

DDL: CREATE, ALTER, and DROP

Even though it may seem easier to just copy or load a file of prepared code, learning any language is best served if you take the time and do some typing (please trust me!).

We must first create a database to hold the tables that will store data. For this first introduction and practice, we will use the example database for tracking a metro bus or transit system from previous chapters. We will call this new database METRO_TRANSIT. During this process, please notice the comments included within the SQL code to provide anecdotal explanations in the code that is a learning moment.

Download a SQL Development Tool

SQL is intended to be a standardized language but there are slight differences across vendors. There are other very capable database products like Oracle, MySQL, PostgreSQL, IBM DB2, and Microsoft SQL Server with very similar implementations of the SQL syntax. This textbook is centric to Microsoft SQL Server based on my experience using the platform since 1990. Unfortunately, there is not enough time or space for me to make every slight change for all these products. Please do your best to follow along if you end up using a product other than SQL Server.

Tool	Platform	Link
DB Visualizer	Windows, MAC, Linux	https://www.dbvis.com/download/
Data Grip	Windows, MAC, Linux	https://www.jetbrains.com/datagrip/download
SQL Server Management Studio	Windows	https://learn.microsoft.com/en-us/ssms/install/install

TABLE 5.1: List of prominent development tools needed to engage METRO_TRANSIT

To get started with connecting to a SQL Server database, please select one of the development tools listed above based on the type of computer or laptop you are using.

Download files for METRO_TRANSIT
https://github.com/ava-orange-education/ultimate-SQL-for-Data-Science-and-Analytics

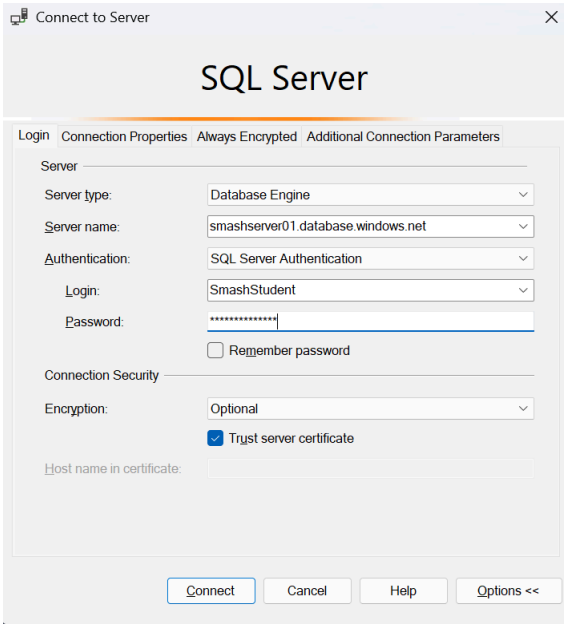


Image 5.1: Screen shot of SQL Server Management Studio Login via browser

In the previous section, you downloaded everything required to establish a connection to a real SQL Server.

Please connect to the server with the following credentials:

- SERVERNAME: SmashServer01.database.windows.net
- AUTHENTICATION: SQL Server Authentication
- LOGIN: SmashStudent
- PASSWORD: SuperSmashSQL!

SQL is the standard database language across the world. There are more positions in the data industry that seek intermediate SQL skills (beyond the basic query statement) than any other technical language. There are literally hundreds of thousands of unfilled/open positions that need skilled database programmers at this exact moment. You will benefit from having a competency in SQL as it helps clarify business objects, their relationships, and a range of typical values; database users become experts of their business data through repetitive engagement and therefore more insightful and valuable to their organization.

Structured Query Language—aka "Sequel"--has been the *de facto* standard database query language since the early 1980s. Essentially, all popular relational database applications have SQL as their core user-based method of engagement. The textbook and online search can provide a wonderful background of the development history of SQL if you are inclined to spend more time learning about that, but we will skip that here and get into the practical application of SQL. If your goals are just observational then it may be better to download the full version of METRO_TRANSIT from the link above.

You will save perhaps 2 or 3 hours of work that will most likely be painful and unenjoyable. If, however, you have a mindset of 'doing the work' of a developer for the sake of knowing a practical, authentic coding experience with SQL, you have come to the right place. Keep working hard by doing the recommended coding of METRO_TRANSIT (an abbreviated or simplified version!) in SQL to be able to run the queries presented in later chapters without having to download extra files.

This text will walk through the process (and pain) of building an abbreviated, sub-set of METRO_TRANSIT to practice Data Definition Language (DDL) by hand. As much as short-cuts simplify peoples' lives so we can spend less time doing work, the only way to cement the learning of building anything---especially code--- is to do the core menial work without any help.

We will follow the 7-step process listed below:

Coding Process Sequence

1) Write the CREATE DATABASE statement (kind of needs to be number one, right?)

2) Write the USE statement to begin defining the newly created database (each semester I have to clean-up several hundred tables that accidentally end-up being created in the sample practice database each class uses for homework because dozens of students skip this step!)

3) Write the CREATE TABLE statements with all columns, proper data types, NULLABILITY, and auto-increment enabled on a surrogate PK. We will see how several keywords will be included to increase functionality and performance.

4) Write ALTER TABLE statements for each foreign key; this last step is separate from the CREATE TABLE statements intentionally as the ALTER TABLE statement allows for the table definitions to be in any order.

5) Save the script for safety as we will be re-running the script several times as we are introduced to development options

6) DROP everything! Just for practice of seeing how the DROP command works (save your typed script!)

7) Re-run the entire script again

Use the following code to create your persona copy of METRO_TRANSIT:

```
/*
Step One: Make the new database
*/

CREATE DATABASE METRO_TRANSIT_put_your_initials_here_plus_3_random_digits
GO
- - Code Example 5.1: SQL code to create a new database


/*
Step Two: Change context of query window to newly created database
*/

USE METRO_TRANSIT_put_your_initials_here_plus_3_random_digits
GO
- - Code Example 5.2: SQL code to change database context

/*
```

The CREATE statement is used to construct a named object with SQL; these objects are not only the database itself, but also each table, column, view, stored procedure, user-defined function, login, job, index, and several others. Pretty much anything that is named within a relational database can be built using the CREATE statement.

Next, let's code the 13 entities that represent a simplified or basic version of the METRO_TRANSIT entity-relationship diagram from chapter 3, *Data Modeling and Normalization*. This abbreviated version is intended to practice writing CREATE and ALTER TABLE statements as well as preliminary INSERT statements without spending the 15 hours it might normally take. These entities include DRIVER, ROUTE_TYPE, ROUTE, VEHICLE_TYPE, VEHICLE, NEIGHBORHOOD, STOP_TYPE, STOP, ROUTE_STOP, TRIP, TRIP_STOP, CUSTOMER, and BOARDING.

Step Three: Establish tables, columns, and PKs with CREATE TABLE statements

Syntax: some organizations have a preferred naming convention; the prefix of 'tbl' is included for each table here as a standard but it certainly is not required.
*/

```
CREATE TABLE tblDRIVER
(DriverID INTEGER IDENTITY(1,1) PRIMARY KEY,
Fname VARCHAR(25) NOT NULL,
Lname VARCHAR(25) NOT NULL,
BirthDate DATE NOT NULL,
Salary Numeric (8,2) NOT NULL,
LicenseNumber VARCHAR(35) NULL)
GO


CREATE TABLE tblROUTE_TYPE
(RouteTypeID INTEGER IDENTITY(1,1) PRIMARY KEY,
RouteTypeName VARCHAR(50) NOT NULL,
RouteTypeDescr VARCHAR(500) NULL)
GO


CREATE TABLE tblROUTE
(RouteID INTEGER IDENTITY(1,1) PRIMARY KEY,
RouteName VARCHAR(100) NOT NULL,
RouteTypeID INTEGER NOT NULL,
RouteAbbrev VARCHAR(20) NOT NULL,
RouteDescr VARCHAR(500) NULL)
GO


CREATE TABLE tblVEHICLE_TYPE
(VehicleTypeID INTEGER IDENTITY(1,1) PRIMARY KEY,
VehicleTypeName VARCHAR(100) NOT NULL,
NumSeats INTEGER NOT NULL,
TotalCapacity INTEGER NOT NULL,
VehicleTypeDescr VARCHAR(500) NULL)
GO
```

```
CREATE TABLE tblVEHICLE
(VehicleID INTEGER IDENTITY(1,1) PRIMARY KEY,
VehicleName VARCHAR(50) NOT NULL,
VehicleTypeID INTEGER NOT NULL,
PlateNumber VARCHAR(20) NOT NULL,
PurchasePrice Numeric(10,2) NOT NULL,
VehiclePurchaseDate DATE NOT NULL)
GO

CREATE TABLE tblNEIGHBORHOOD
(NeighborhoodID INTEGER IDENTITY(1,1) PRIMARY KEY,
NeighborhoodName VARCHAR(50) NOT NULL,
NeighborhoodDescr VARCHAR(500) NULL)
GO

CREATE TABLE tblSTOP_TYPE
(StopTypeID INTEGER IDENTITY(1,1) PRIMARY KEY,
StopTypeName VARCHAR(50) NOT NULL,
StopTypeDescr VARCHAR(500) NULL)
GO


CREATE TABLE tblSTOP
(StopID INTEGER IDENTITY(1,1) PRIMARY KEY,
StopName VARCHAR(50) NOT NULL,
StopTypeID INTEGER NOT NULL,
NeighborhoodID INTEGER NOT NULL,
GPS_CorOrd VARCHAR(50) NULL)
GO


CREATE TABLE tblROUTE_STOP
(RouteStopID INTEGER IDENTITY(1,1) PRIMARY KEY,
RouteID INTEGER NOT NULL,
StopID INTEGER NOT NULL,
SequenceNumber INTEGER NULL,
RevisedDate DATE NOT NULL)
GO


CREATE TABLE tblTRIP
(TripID INTEGER IDENTITY(1,1) PRIMARY KEY,
DriverID INTEGER NOT NULL,
VehicleID INTEGER NOT NULL,
RouteID INTEGER NOT NULL,
BeginDateTime DATETIME NOT NULL,
EndDateTime DATETIME NULL)
GO


CREATE TABLE tblTRIP_STOP
(TripStopID INTEGER IDENTITY(1,1) PRIMARY KEY,
TripID INTEGER NOT NULL,
StopID INTEGER NOT NULL,
TripStopDateTime DATETIME NOT NULL)
```

```
GO

CREATE TABLE tblCUSTOMER
(CustomerID INTEGER IDENTITY(1,1) PRIMARY KEY,
Fname VARCHAR(25) NOT NULL,
Mname VARCHAR (25) NULL,
Lname VARCHAR(25) Not NULL,
BirthDate DATE NULL,
CustAddress VARCHAR (100) NULL,
CustCity VARCHAR(50) NULL,
CustState VARCHAR(50) NULL,
PostalCode VARCHAR(15) NULL)
GO
```

- - Code Example 5.3: SQL code to establish 12 tables

```
/*
Please note the creation of two foreign key values right in the middle of the
CREATE TABLE statement for tblBOARDING coming up next; this is called 'in-line'
definition and is frequently used when there are only a handful of tables in a
database. The reason is the source table that provides the foreign key values
must already exist, and things get messy and difficult to track if there are
dozens of tables to worry about.
*/


CREATE TABLE tblBOARDING
(BoardingID INTEGER IDENTITY(1,1) PRIMARY KEY,
CustomerID INTEGER FOREIGN KEY REFERENCES tblCUSTOMER (CustomerID) NOT NULL,
TripStopID INTEGER FOREIGN KEY REFERENCES tblTRIP_STOP (TripStopID) NOT NULL,
BoardingDateTime DATETIME NOT NULL,
Fare Numeric(5,2) NOT NULL)
GO


PRINT 'Completed CREATE TABLE statements for all tables in abbreviated
METRO_TRANSIT'
GO
```

- - Code Example 5.4: SQL code to create table tblBOARDING

```
/*
Step Four: Establish foreign keys with ALTER TABLE statements
Please note the foreign keys are coded separately from the CREATE TABLE code.
ALTER TABLE is for schema changes on tables already built.
*/


ALTER TABLE tblROUTE
ADD CONSTRAINT FK_tblROUTE_RouteTypeID
FOREIGN KEY (RouteTypeID)
REFERENCES tblROUTE_TYPE (RouteTypeID)
GO
```

```
ALTER TABLE tblVEHICLE
ADD CONSTRAINT FK_tblVEHICLE_VehicleTypeID
FOREIGN KEY (VehicleTypeID)
REFERENCES tblVEHICLE_TYPE (VehicleTypeID)
GO


ALTER TABLE tblSTOP
ADD CONSTRAINT FK_tblSTOP_StopTypeID
FOREIGN KEY (StopTypeID)
REFERENCES tblSTOP_TYPE (StopTypeID)
GO


ALTER TABLE tblSTOP
ADD CONSTRAINT FK_tblSTOP_NeighborhoodID
FOREIGN KEY (NeighborhoodID)
REFERENCES tblNEIGHBORHOOD (NeighborhoodID)
GO


ALTER TABLE tblROUTE_STOP
ADD CONSTRAINT FK_tblROUTE_STOP_StopID
FOREIGN KEY (StopID)
REFERENCES tblSTOP (StopID)
GO


ALTER TABLE tblROUTE_STOP
ADD CONSTRAINT FK_tblROUTE_STOP_RouteID
FOREIGN KEY (RouteID)
REFERENCES tblROUTE (RouteID)
GO


ALTER TABLE tblTRIP
ADD CONSTRAINT FK_tblTRIP_DriverID
FOREIGN KEY (DriverID)
REFERENCES tblDRIVER (DriverID)
GO


ALTER TABLE tblTRIP
ADD CONSTRAINT FK_tblTRIP_VehicleID
FOREIGN KEY (VehicleID)
REFERENCES tblVEHICLE (VehicleID)
GO


ALTER TABLE tblTRIP
ADD CONSTRAINT FK_tblTRIP_RouteID
FOREIGN KEY (RouteID)
REFERENCES tblROUTE (RouteID)
GO
```

```
ALTER TABLE tblTRIP_STOP
ADD CONSTRAINT FK_tblTRIP_STOP_TripID
FOREIGN KEY (TripID)
REFERENCES tblTRIP (TripID)
GO


ALTER TABLE tblTRIP_STOP
ADD CONSTRAINT FK_tblTRIP_STOP_StopID
FOREIGN KEY (StopID)
REFERENCES tblSTOP (StopID)
GO


PRINT 'Completed ALTER TABLE statements for adding foreign keys'
GO
```

- - Code Example 5.5: SQL code to establish foreign keys

/*
The sequence of steps above establishes a basic, functional database that we can code up in effectively one 60-minute session. As we learn more and are ready for challenges, we will add to the basic script (also called 'schema'). The goal of slowly adding complexity to this beginning is for you to engage the development process as well as typing out the syntax of building each object; remember that conducting the steps as even the most complex database construction will be very similar. Congrats on getting started!

Next, we will see the effects of issuing a DROP statement (save your code!!).

Step Five: Save the script as a file as we will need to re-run it in a moment: Ctrl +S

Also, for safety, both save your script as a .txt file as well as copy the script and send it to yourself through an email. Too often we do an hour or so of coding and then lose our work. Always save a copy of your script!

If you are using Azure Data Studio or SQL Server Management Studio, press the Control key + S key simultaneously to save your script.

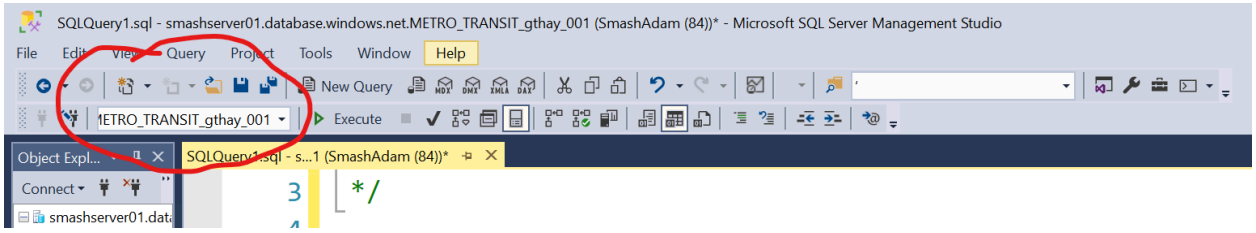
Step Six: DROP database.

This command will destroy everything we have created so far, so make sure you have saved your script as a file or at least have copied the code out to a text editor and perhaps even emailed it back to yourself! We will re-run the script in just a moment to re-establish what we have built so far after seeing the effects of the last DDL command.
*/

```
USE DATABASE MASTER
GO
```

- - Code Example 5.6: SQL code to set context of database window

```
/*
```
The USE statement may not work depending on the version of software you have. If this statement errors, change context to MASTER database with a mouse by changing the active database in the upper ribbon. See the screenshot below with the location of database context circled in red:



Screenshot of SQL Server Management Studio establishing context of which database code will run inside

```
*/
```

```
DROP DATABASE METRO_TRANSIT_put_your_initials_here_plus_3_random_digits
GO
```

- - Code Example 5.7: Dangerous SQL code to drop a database

The above steps change the context of the query window to the central database (master) of the database management system and allows us to eliminate the just-completed database (don't worry, you have saved your script and can get it back in a few seconds).

Now that we have dropped the database, refresh the window of your database management system. Spend a brief minute and look around the environment; the METRO_TRANSIT database is nowhere to be found! Again, the DROP command is used to destroy any object we have created; during the development process we commonly need to 'start over' with slight alterations, such as adding a column previously forgotten, or was perhaps defined with incorrect data type.

Instead of spending tens of minutes using the ALTER TABLE statement, it is often significantly faster to just DROP the whole database and run the coded script again; in less than a minute we are right back in a desired state.

If you did not save your script (or send it to your personal email), well, you will need to re-type the code from steps 1 through 5. It's ok to growl!

Welcome to the software development process; every seasoned developer has scars from doing the exact same thing in their career.

Step Seven: Re-run the entire database script.

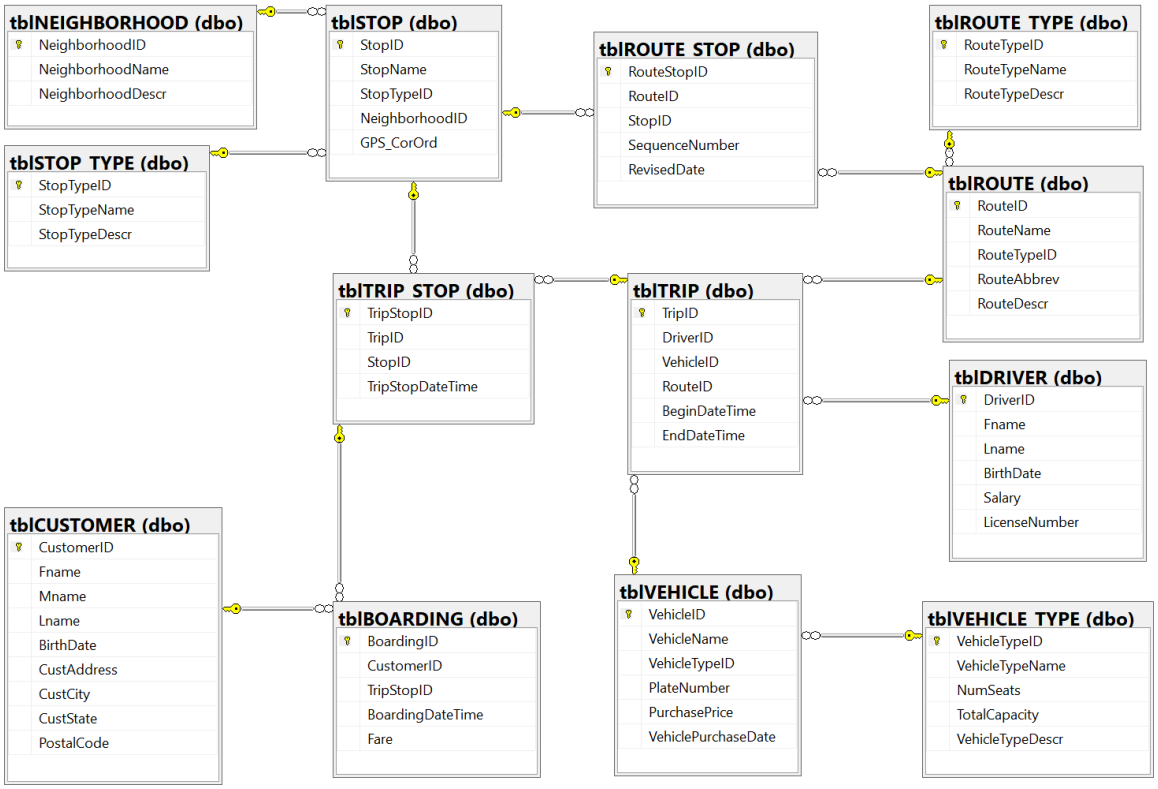


Figure 5.1: Simplified Entity-Relationship Diagram of METRO_TRANSIT

The above ERD in Figure 5.1 represents the rendering of the code from Example Code 5.1 through Example Code 5.5. This code represents a slightly simplified version of METRO_TRANSIT and is intended to provide the following advantages:

1) Walk readers step-by-step through a schema build process in roughly 1 hour. The fully normalized schema of METRO_TRANSIT is too elaborate for the purposes of showing new developers how to code up a database schema. The more elaborate schema is available for download to evaluate:

https://github.com/ava-orange-education/ultimate-SQL-for-Data-Science-and-Analytics

2) Keep offline readers engaged! If a reader is not able to connect to the internet or download the example data files, they can still follow along by building their own copy of the METRO_TRANSIT database, albeit in a simplified form.

DML: INSERT, UPDATE, DELETE

Now that we have the structure of a basic database, we are ready to add data using Data Modification Language, or DML. While the most-common commands and statements under DML are INSERT, UPDATE, and DELETE, there are dozens of additional key words

As we begin to populate our database, pay attention to the sequence of which tables get data inserted, first; these tables are almost always the 'look-up' tables that provide foreign key values to the more transactional tables in the database.

Most relational databases will have perhaps 90% of the entire contents of data concentrated in only 2 or 3 tables (seems strange I know). Some databases will have a look-up table with only 4 or so values, while one of its transactional tables may have tens of millions of rows. This design is fast and efficient in not only recording new data accurately in the database but also fast enough to get queries returned from the largest of tables. This is intentional and considered the brilliance of relational theory.

In the METRO_TRANSIT database, the tables that will store the bulk of the data are tblTRIP_STOP and tblBOARDING. These are the transactional tables as they contain the details every time a vehicle drives past a stop on a scheduled trip even if no one boards the bus! Again, a transactional table often contains multiple foreign keys that allow a connection back to the look-up tables. Recognize please that values contained under the columns in these transactional tables have no 'words'; meaning, if we were to break into a database and read just the values from a transactional table, it should make zero sense to a person as the numbers have no context.

All associated context is in the look-up tables. I frequently need to remind first-year students that we store data in relational databases for machines and not people as we will write queries that connect tables together when we need to read names associated with values stored in transactional tables.

Look-Up Tables must be populated first

Think about it. If 90% or more of the data in a relational database is held in the handful of transactional tables and these transactional tables reference the values in the surrounding look-up tables for the detailed context via foreign keys, then is should be apparent that the foreign key values must already exist ***before*** they can be referenced. We will populate the look-up tables in the METRO_TRANSIT database now (aside from including names of several famous people from my early life, these values are entirely fictitious and fabricated solely for illustrative purposes. Trust that none of these data are personally identifiable).

Step One: identify look-up tables (hint: they do not have any foreign keys!)

Also please remember we do not populate the primary key values because the auto-increment feature has been enabled (that is what IDENTITY(1,1) does in the CREATE TABLE statement). This is important to follow!

Please notice the column headers are named specifically in the INSERT statement; this is considered an industry best-practice as skipping the naming of them as a shortcut here may unintentionally put values in the wrong columns (like first and last names getting flipped). This also provides flexibility when working with various external sources of data and we are tasked with importing large amounts of data (we can change how we read the data as opposed to making the external source change how the data is written).

Also notice that several values can be included in a single INSERT statement. While this may seem a bit manual (it is), we need to experience the burdensome typing if only to recognize the impact of 'better code' when we get to creating stored procedures. For now, appreciate that there are several ways to get data brought into a relational database; it will be important for our future careers to know more than a few methods!

Code example: INSERT Statement

```
INSERT INTO tblDRIVER (Fname, Lname, BirthDate, Salary, LicenseNumber)
VALUES
('Jimi', 'Hendrix', 'November 27, 1942', 42590, 'JMHend3487RR'),
('Bruce', 'Lee', 'November 27, 1940', 42590, 'BTLee*2198DL'),
('Jim', 'Morrison', 'December 8, 1943', 47625, 'JPMorr7762WS'),
('Meryl', 'Streep', 'June 22, 1949', 46965, 'MDStre8121UJ')
GO

PRINT 'Completed INSERT INTO statements for tblDRIVER'
```

- - Code Example 5.8: SQL code to insert sample data into table tblDRIVER

```
/*
String data types have single quotations while numbers (INTEGER, FLOAT, NUMERIC, DECIMAL,
MONEY) do not. Date values are considered string data types.

Note: Jimi Hendrix and Bruce Lee really do share a birthday (and an eternal resting place in Seattle!)
*/

INSERT INTO tblROUTE_TYPE (RouteTypeName, RouteTypeDescr)
VALUES
('Express', 'An express route is often designed to have limited stops during the morning and evening
commute hours'),
```

('Special Event', 'A special event route is often scheduled for high-attendance events at sporting venues or downtown'),
('Regular', 'A regular route is any route not identified as express or special event'),
('Inclement Weather/Reduced Service', 'Any route affected by weather')
GO

- - Code Example 5.9: SQL code to insert more sample data into table tblROUTE_TYPE

```
/*
Most description columns (those ending in 'DESCR') are often defined as NULL; this means that values are not mandatory and may be left empty. Because of this, we will very rarely write queries to filter across a range of values held in any description column. This means no important data should be placed in a description field and its intent is to provide clarification for users on a case-by-case basis only if necessary.
*/

PRINT 'Completed INSERT INTO statements for tblROUTE_TYPE'

INSERT INTO tblROUTE (RouteName, RouteTypeID, RouteAbbrev, RouteDescr)

VALUES
('Sodo-Downtown Express', (SELECT RouteTypeID FROM tblROUTE_TYPE WHERE RouteTypeName = 'Express'), '42-E', 'Express route from South of Downtown to main stops downtown'),

('Sodo-Downtown', (SELECT RouteTypeID FROM tblROUTE_TYPE WHERE RouteTypeName = 'Regular'), '42', 'Commuter route from South of Downtown to main stops downtown'),

('Fremont-Downtown Express', (SELECT RouteTypeID FROM tblROUTE_TYPE WHERE RouteTypeName = 'Express'), '78-E', 'Express route from Fremont to main stops downtown'),

('Fremont-Downtown', (SELECT RouteTypeID FROM tblROUTE_TYPE WHERE RouteTypeName = 'Regular'), '78', 'Commuter route from Fremont to main stops downtown'),

('Fremont-Waterfront-Downtown', (SELECT RouteTypeID FROM tblROUTE_TYPE WHERE RouteTypeName = 'Special Event'), '78-S', 'Special route from Fremont along waterfront to main stops downtown during summer weekend concert series'),

('Capitol Hill-Downtown', (SELECT RouteTypeID FROM tblROUTE_TYPE WHERE RouteTypeName = 'Regular'), '32', 'Regular route from Capitol Hill to main stops downtown')
GO

PRINT 'Completed INSERT INTO statements for tblROUTE'

INSERT INTO tblVEHICLE_TYPE (VehicleTypeName, NumSeats, TotalCapacity, VehicleTypeDescr)

VALUES
```

('Standard Bus', 54, 76, 'Standard bus is primary use for residential neighborhoods during non-commute hours'),

('Double-Decker Bus', 96, 144, 'Double Decker bus is primary coach on state highways between major population centers as well as for special events'),

('Standard Light Rail Car', 58, 88, 'Standard Light Rail car is staple for fixed-track, high-volume routes that connect suburban and urban centers with the airport often in alignments with 4 connected cars'),

('Heavy Rail Single Passenger Train', 40, 50, 'Heavy Rail Single cars are placed on routes that are greater than 85 KM from major urban centers; these often are aligned with 12 connected cars')
GO

PRINT 'Completed INSERT INTO statements for tblVEHICLE_TYPE'

- - Code Example 5.10: SQL code to insert sample data into table tblVEHICLE_TYPE

/*
Next set of INSERT statements will include a SELECT statement inside of the INSERT statement; this is called a 'sub-query' and is both slightly confusing but also extremely valuable in leveraging 'what we do know' to find what we do not know (usually the value of a required foreign key). We will spend more time on this in a minute; for now, just recognize the placement and use of the subquery and hopefully appreciate that it is saving us time from having to manually look the values up!
*/

INSERT INTO tblVEHICLE (VehicleName, VehicleTypeID, PlateNumber, PurchasePrice, VehiclePurchaseDate)
VALUES

('GS203', (SELECT VehicleTypeID FROM tblVEHICLE_TYPE WHERE VehicleTypeName = 'Double-Decker Bus'), 'DRE-657J',887533, 'March 3, 2019'),

('GS204', (SELECT VehicleTypeID FROM tblVEHICLE_TYPE WHERE VehicleTypeName = 'Double-Decker Bus'), 'GTT-7JHS',887533, 'March 3, 2019'),

('PA7', (SELECT VehicleTypeID FROM tblVEHICLE_TYPE WHERE VehicleTypeName = 'Standard Bus'), 'XYW-23KJ', 566988,'July 13, 2023'),

('GS208', (SELECT VehicleTypeID FROM tblVEHICLE_TYPE WHERE VehicleTypeName = 'Standard Bus'), 'TG8-99L2',604995,'August 9, 2024')
GO

PRINT 'Completed INSERT INTO statements for tblVEHICLE'

INSERT INTO tblNEIGHBORHOOD (NeighborhoodName, NeighborhoodDescr)
VALUES

('Capitol Hill', 'Vibrant, colorful, and walkable neighborhood with many shops and services'),

('Waterfront', 'Naturally scenic, tourist-focused neighborhood with many food options'),

('Downtown', 'Urban, business-focused with many restaurants and transit connections to entire region'),

('SoDo', 'Gritty and industrial neighborhood'),

('Fremont', 'Eccentric and organic feel with shops and restaurants that fit a range of budgets')
GO

PRINT 'Completed INSERT INTO statements for tblNEIGHBORHOOD'

INSERT INTO tblSTOP_TYPE (StopTypeName, StopTypeDescr)
VALUES
('Uncovered', 'Standard stop that has a 3-person bench and garbage can'),

('Covered', 'Upgraded 3-sided stop'),

('Tunnel Station', 'High capacity stop that has full service and security'),

('Elevated Rail Station', 'High capacity stop that serves only rail')
GO

PRINT 'Completed INSERT INTO statements for tblSTOP_TYPE'

INSERT INTO tblSTOP (StopName, StopTypeID, NeighborhoodID, GPS_CorOrd)
VALUES
('Elliott Avenue and Mercer Street', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'UnCovered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'Waterfront'),'47.6567° N, 122.2470° W'),

('Hwy 99-N 36th', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'Covered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'Fremont'),'47.6567° N, 122.3474° W'),

('Fourth Avenue South and Lander Street', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'UnCovered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'SoDo'),'47.0117° N, 122.2430° W'),

('Sixth Avenue and Battery Street', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'UnCovered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'Downtown'),'47.6417° N, 122.3276° W'),

('Hwy 99-N 54th', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'Covered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'Fremont'),'47.6697° N, 122.3964° W'),

('Fourth Avenue and Seneca Street', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'UnCovered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'Downtown'),'47.6398° N, 122.3371° W'),

('Broadway Avenue and Cherry Street', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'Covered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'Capitol Hill'),'47.6317° N, 122.3383° W'),

('First Avenue and Terry Street', (SELECT StopTypeID FROM tblSTOP_TYPE WHERE StopTypeName = 'Covered'), (SELECT NeighborhoodID FROM tblNEIGHBORHOOD WHERE NeighborhoodName = 'SoDo'),'47.6212° N, 122.3243° W')
GO

PRINT 'Completed INSERT INTO statements for tblSTOP'
GO

INSERT INTO tblROUTE_STOP (RouteID, StopID, SequenceNumber, RevisedDate)
VALUES
((SELECT RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Waterfront-Downtown'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Sixth Avenue and Battery Street'), 4, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Waterfront-Downtown'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Hwy 99-N 36th'), 2, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Waterfront-Downtown'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Elliott Avenue and Mercer Street'), 3, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Waterfront-Downtown'), (SELECT StopID FROM tblSTOP WHERE StopName = 'First Avenue and Terry Street'), 1, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName ='Sodo-Downtown Express'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Fourth Avenue South and Lander Street'), 1, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName ='Sodo-Downtown Express'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Sixth Avenue and Battery Street'), 2, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName = 'Capitol Hill-Downtown'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Broadway Avenue and Cherry Street'), 1, 'May 1, 2024'),

((SELECT RouteID FROM tblROUTE WHERE RouteName = 'Capitol Hill-Downtown'), (SELECT StopID FROM tblSTOP WHERE StopName = 'Sixth Avenue and Battery Street'), 2, 'May 1, 2024')
GO

PRINT 'Completed INSERT INTO statements for tblROUTE_STOP'

INSERT INTO tblCUSTOMER (Fname, Lname, CustAddress, CustCity, CustState, PostalCode, BirthDate)

```
VALUES
('Ivey','Hazekamp','7279 North Lavender Lake Avenue','PETROLEUM','Indiana, IN','46778','1995-12-24'),

('Darcel','Eustache','6087 West Cambridge Beach Sloop','FRUITA','Colorado, CO','81521','1995-06-11'),

('Kenyetta','Terron','6664 NW Arthur Terrace Drive','GRAND RIVER','Ohio, OH','44045','1995-09-27'),

('Janey','Lundgren','14351 N Oak Beach Walk','CENTRALIA','Illinois, IL','62801','1996-01-25')
GO

PRINT 'Completed INSERT INTO statements for tblCUSTOMER'
GO

INSERT INTO tblTRIP (DriverID, VehicleID, RouteID, BeginDateTime, EndDateTime)
VALUES
((SELECT DriverID FROM tblDRIVER WHERE Fname = 'Meryl' AND Lname = 'Streep' AND BirthDate =
'June 22, 1949'), (SELECT VehicleID FROM tblVEHICLE WHERE VehicleName = 'GS204'), (SELECT
RouteID FROM tblROUTE WHERE RouteName = 'Capitol Hill-Downtown'), '2025-04-24 06:17:00',
NULL),

((SELECT DriverID FROM tblDRIVER WHERE Fname = 'Jimi' AND Lname = 'Hendrix' AND BirthDate =
'November 27, 1942'), (SELECT VehicleID FROM tblVEHICLE WHERE VehicleName = 'GS203'),
(SELECT RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Downtown Express'),
'2025-04-24 06:42:55', NULL),

((SELECT DriverID FROM tblDRIVER WHERE Fname = 'Meryl' AND Lname = 'Streep' AND BirthDate =
'June 22, 1949'), (SELECT VehicleID FROM tblVEHICLE WHERE VehicleName = 'GS204'), (SELECT
RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Waterfront-Downtown'), '2025-04-23
07:23:00', NULL),

((SELECT DriverID FROM tblDRIVER WHERE Fname = 'Bruce' AND Lname = 'Lee' AND BirthDate =
'November 27, 1940'), (SELECT VehicleID FROM tblVEHICLE WHERE VehicleName = 'PA7'), (SELECT
RouteID FROM tblROUTE WHERE RouteName = 'Capitol Hill-Downtown'), '2025-04-24 06:57:00',
NULL),

((SELECT DriverID FROM tblDRIVER WHERE Fname = 'Meryl' AND Lname = 'Streep' AND BirthDate =
'June 22, 1949'), (SELECT VehicleID FROM tblVEHICLE WHERE VehicleName = 'GS204'), (SELECT
RouteID FROM tblROUTE WHERE RouteName = 'Fremont-Downtown'), '2025-04-29 08:07:00', NULL)
GO

PRINT 'Completed INSERT INTO tblTRIP'
GO

INSERT INTO tblTRIP_STOP (TripID, StopID, TripStopDateTime)
VALUES
((SELECT TripID FROM tblTRIP T
        JOIN tblDRIVER D ON D.DriverID = T.DriverID
        JOIN tblROUTE R ON T.RouteID = T.RouteID
WHERE D.Fname = 'Bruce' AND D.Lname = 'Lee'
AND R.RouteName = 'Capitol Hill-Downtown'
```

```
AND T.BeginDateTime = '2025-04-24 06:57:00'),
(SELECT S.StopID
FROM tblSTOP S
WHERE StopName = 'Broadway Avenue and Cherry Street'), '2025-04-24 07:06:51'),


--second value for tblTRIP_STOP
((SELECT TripID FROM tblTRIP T
        JOIN tblDRIVER D ON D.DriverID = T.DriverID
        JOIN tblROUTE R ON T.RouteID = T.RouteID
WHERE D.Fname = 'Bruce' AND D.Lname = 'Lee'
AND R.RouteName = 'Capitol Hill-Downtown'
AND T.BeginDateTime = '2025-04-24 06:57:00'),
(SELECT S.StopID FROM tblSTOP S
        WHERE StopName = 'Fourth Avenue and Seneca Street'), '2025-04-24 07:15:23'),


-- third value for tblTRIP_STOP
((SELECT TripID FROM tblTRIP T
        JOIN tblDRIVER D ON D.DriverID = T.DriverID
        JOIN tblROUTE R ON T.RouteID = T.RouteID
WHERE D.Fname = 'Jimi' AND D.Lname = 'Hendrix'
AND R.RouteName = 'Fremont-Downtown Express'
AND T.BeginDateTime = '2025-04-24 06:42:55'),
(SELECT S.StopID
FROM tblSTOP S
WHERE StopName = 'Hwy 99-N 54th'), '2025-04-24 06:43:00'),


-- fourth value for tblTRIP_STOP
((SELECT TripID FROM tblTRIP T
        JOIN tblDRIVER D ON D.DriverID = T.DriverID
        JOIN tblROUTE R ON T.RouteID = T.RouteID
WHERE D.Fname = 'Jimi' AND D.Lname = 'Hendrix'
AND R.RouteName = 'Fremont-Downtown Express'
AND T.BeginDateTime = '2025-04-24 06:42:55'),
(SELECT S.StopID
FROM tblSTOP S
WHERE StopName = 'Hwy 99-N 36th'), '2025-04-24 06:47:23'),


-- fifth value for tblTRIP_STOP
((SELECT TripID FROM tblTRIP T
        JOIN tblDRIVER D ON D.DriverID = T.DriverID
        JOIN tblROUTE R ON T.RouteID = T.RouteID
WHERE D.Fname = 'Jimi' AND D.Lname = 'Hendrix'
AND R.RouteName = 'Fremont-Downtown Express'
AND T.BeginDateTime = '2025-04-24 06:42:55'),
(SELECT S.StopID
FROM tblSTOP S
```

```
WHERE StopName = 'Elliott Avenue and Mercer Street'), '2025-04-24 06:56:34'),


-- sixth value for tblTRIP_STOP
((SELECT TripID FROM tblTRIP T
        JOIN tblDRIVER D ON D.DriverID = T.DriverID
        JOIN tblROUTE R ON T.RouteID = T.RouteID
WHERE D.Fname = 'Meryl' AND D.Lname = 'Streep'
AND R.RouteName = 'Capitol Hill-Downtown'
AND T.BeginDateTime = '2025-04-23 07:23:00'),
(SELECT S.StopID
FROM tblSTOP S
        WHERE StopName = 'Sixth Avenue and Battery Street'), '2025-04-23 07:41:06')
GO

PRINT 'Completed INSERT statements with 6 values for tblTRIP_STOP'
GO
```

- - Code Example 5.11: SQL code inserting sample data into tables tblVEHICLE, tblNEIGHBORHOOD, tblSTOP_TYPE, tblSTOP, tblROUTE_STOP, tblCUSTOMER, tblTRIP, and tblTRIP_STOP

The SELECT statement inside the INSERT statement above again is called a 'subquery' and demonstrates part of the power that a relational database has. We do not have to manually discover the foreign key values in each of the INSERT statements. We simply write the query in place of the expected value and the database engine will do the replacement for us automatically. Pretty cool?

Again, there needs to be some data entry next if you have not been able to download the data files located at the following share:

https://github.com/ava-orange-education/ultimate-SQL-for-Data-Science-and-Analytics

This next section may seem arduous for those entering the data manually! Understand that there will be an easier way soon, and going through this exercise manually reinforces the principles of automation and repetitive code.

As we continue populating the METRO_TRANSIT database, please notice the multiple look-ups required for tblROUTE_STOP (one for each foreign key). While necessary to avoid hard-coding FK values, these look-ups can be tedious!

The order of columns in an INSERT statement can be arbitrary if they are specified; Even though BirthDate, for example, appears in the schema as the 4th column (after Fname, Mname, and Lname), we can stick it at the end of the INSERT statement. In this case, the source from which I generated this sample data was in an order different from this schema.

Second point: Mname is not included in this INSERT statement. First, it was designed with 'NULL' (as opposed to 'NOT NULL') which means values are not mandatory. Second, the source by which I obtained this sample data did not include a middle name.

TRANSACTIONAL TABLES: hold 90% or more of data in entire database

Once the look-up tables have been populated, we are able and ready to begin referencing them for context in the transactional tables. For the METRO_TRANSIT database, the transactional tables are the 3 that have the highest frequency of new data and include tblTRIP, tblTRIP_STOP, and tblBOARDING. Imagine a public transportation system in a typical metropolitan center anywhere in the world; there may be only a dozen or so types of vehicles (as we have typed-in above) over a span of 50 years. In some of the busiest transit systems, there are often over 10 million boardings in a single day! Recognize the difference in volatility in the number of rows affecting a look-up table versus a transactional table; one is effectively static and unchanging (also known as 'read-only') and the other is a firehose of insert activity. Side note: while 10 million rows a day for tblBOARDING may seem like an incredible amount of data, this number is far from being considered massive. Social media sites and email authentication easily surpass 1 billion rows of activity each day.

These next few INSERT statements are for the transactional tables of the METRO_TRANSIT database; since we have not yet learned the structures of better efficiency and automation (hint: stored procedures are coming up in another chapter). These statements will be cumbersome and clumsy but hopefully leave a memory of a lousy method of data entry.

Please appreciate the excessive typing of including subqueries in an insert statement, which will prove to be a very clumsy method of getting these tables populated. More efficient methods will be shown soon.

Code Example: INSERT statement with embedded SELECT statement

```
INSERT INTO tblBOARDING (CustomerID, TripStopID, Fare, BoardingDateTime)
VALUES(
(SELECT CustomerID
FROM tblCUSTOMER
WHERE Fname = 'Ivey' AND Lname = 'Hazekamp' AND BirthDate = '1995-12-24'),

(SELECT TripStopID FROM tblTRIP_STOP TS
      JOIN tblTRIP T ON TS.TripID = T.TripID
      JOIN tblROUTE R ON T.RouteID = T.RouteID
      JOIN tblSTOP S ON TS.StopID = S.StopID
WHERE R.RouteName = 'Capitol Hill-Downtown'
AND S.StopName = 'Sixth Avenue and Battery Street'
```

```
AND BeginDateTime = '2025-04-23 07:23:00'),
3.75, '2025-04-23 07:41:35')
GO

INSERT INTO tblBOARDING (CustomerID, TripStopID, Fare, BoardingDateTime)
VALUES(
(SELECT CustomerID
FROM tblCUSTOMER
WHERE Fname = 'Ivey' AND Lname = 'Hazekamp' AND BirthDate = '1995-12-24'),

(SELECT TripStopID FROM tblTRIP_STOP TS
      JOIN tblTRIP T ON TS.TripID = T.TripID
      JOIN tblROUTE R ON T.RouteID = T.RouteID
      JOIN tblSTOP S ON TS.StopID = S.StopID
WHERE R.RouteName = 'Fremont-Downtown Express'
AND S.StopName = 'Hwy 99-N 36th'
AND BeginDateTime = '2025-04-24 06:42:55.000'),
3.75, '2025-04-24 06:47:23.000')
GO

INSERT INTO tblBOARDING (CustomerID, TripStopID, Fare, BoardingDateTime)
VALUES(
(SELECT CustomerID
FROM tblCUSTOMER
WHERE Fname = 'Kenyetta' AND Lname = 'Terron' AND BirthDate = '1995-09-27'),

(SELECT TripStopID FROM tblTRIP_STOP TS
      JOIN tblTRIP T ON TS.TripID = T.TripID
      JOIN tblROUTE R ON T.RouteID = T.RouteID
      JOIN tblSTOP S ON TS.StopID = S.StopID
WHERE R.RouteName = 'Fremont-Downtown Express'
AND S.StopName = 'Hwy 99-N 36th'
AND BeginDateTime = '2025-04-24 06:42:55'),
3.25,'2025-04-24 06:47:23'
)
GO

PRINT 'Completed INSERT statements with 3 values for tblBOARDING'
GO
```

- - Code Example 5.12: SQL code to insert 3 values into tblBOARDING

SELECT Statement

Finally! We are here with a database and ready to see how to engage with SQL. These examples are intended to provide a solid set of basic skills that can be used immediately in addition to providing an intriguing and hopefully inspirational foundation for growth. In the METRO_TRANSIT database, there are only a handful of rows of data, which for this initial phase, may be small enough for you to read the data manually (called 'eyeballing'). It is not normally worth anyone's time or effort. For now, feel free to manually review the data as it may help build trust and belief that everything is accurate.

The basic and perhaps most frequently used command of SQL is the SELECT statement. This is how we obtain filtered data back to our screen (called the 'result set').

We can start at the very beginning with the introduction of the basic SQL query, known as a SELECT statement. A SELECT statement might be the most basic as well as most common statement within the SQL domain. It almost always consists of at least two keywords (SELECT and FROM) but may also contain several other keywords depending on the complexity and filtering requirements of the query. The list of keywords for a basic SQL query is as follows (always in this sequence):

- The SELECT statement determines the column(s) that values of data are to be retrieved.

- The FROM clause identifies the table(s) that contain the column(s) in the SELECT clause. In all cases, the keyword SELECT precedes the keyword FROM (in a later module, we will introduce subqueries where multiple SELECT statements are embedded within a single query, but for now, this is the rule to follow).

Example query 1: "What are the first and last names of people in the table tblCUSTOMER?"

```
SELECT  Fname, Lname
FROM tblCUSTOMER
```

- - Code Example 5.13: SQL code showing basic query

Next is the WHERE clause, which is not mandatory in a basic SELECT statement. The WHERE clause provides users the option to filter the rows of data returned from the database, effectively "fine-tuning" the result set to reduce unnecessary clutter.

Example query 2: "What are the first and last names of people in the CUSTOMER table who were born after June 9, 1993?"

```
SELECT  Fname, Lname, BirthDate
FROM tblCUSTOMER
WHERE BirthDate > 'June 9, 1993'
```

- - Code Example 5.14: SQL code using WHERE clause

As SQL is very flexible, a user can include more than one single conditional filter in the WHERE clause by simply adding the keyword 'AND' between each condition.

Example query 3: “What are the first and last names of people in the passenger table who were born after June 9, 1993, who also have an address in the state of California?”

```
SELECT  Fname, Lname, BirthDate, CustCity, CustState
FROM tblCUSTOMER
WHERE BirthDate > 'June 9, 1993'
AND CustState = 'California, CA'
```

- - Code Example 5.15: SQL code with multiple conditions in WHERE clause

The pattern to familiarize in our brain and memorize is essentially only 3 words (do not worry, this is just the start): SELECT, FROM, WHERE

- SELECT: List of column(s) to obtain data values
- FROM: List of tables that contain the specified column(s) in SELECT
- WHERE: Optional clause that provides filtering of values based on conditional logic

Most businesses have questions that require data from many tables in a single query; we will explain how to do this in the next chapter. For now, focus on gaining a solid understanding of the structure and the dozen or so basic operators and how they work. Just skip to the challenges at the end of the chapter if you have previous SQL experience.

The next examples of practice queries from SQL require that the query window has the context of the database we have recently designed, coded, and populated with a handful of values. This can be completed by choosing the database name METRO_TRANSIT in the navigation window or typing and executing *USE METRO_TRANSIT*

If you are not connected to the cloud database provided by the publisher, then the queries below are drawing only from the data that was entered by you in the previous pages. Try to download the browser-based tools (either SSMS or Azure Data Studio) to be able to engage the sample data in the proper context!

It may be a burden to ‘hand-type’ more than a page of data, but the ability to engage in real-time with live data is critical for many people learning this language.

Query 1: type the following and run in the query window:

```
SELECT VehicleID, VehicleName, VehiclePurchaseDate
FROM tblVEHICLE
WHERE VehiclePurchaseDate = 'March 3, 2019'
```

Results | Messages

	VehicleID	VehicleName	VehiclePurchaseDate
1	1	GS203	2019-03-03
2	2	GS204	2019-03-03

- - Code Example 5.16: Simple query using '=' with result set

Query 2: type the following and run in the query window:

```
SELECT VehicleID, VehicleName, VehiclePurchaseDate
FROM tblVEHICLE
WHERE VehiclePurchaseDate > 'June 1, 2018'
```

Results | Messages

	VehicleID	VehicleName	VehiclePurchaseDate
1	1	GS203	2019-03-03
2	2	GS204	2019-03-03
3	3	PA7	2023-07-13
4	4	GS208	2024-08-09

- - Code Example 5.17: Simple query using '>' with result set

The queries above are returning data from the table tblVEHICLE as that is the value in the FROM clause. The WHERE clause in each query has an operator that filters which data that are returned. These are literal and absolute filters.

For example, in query 1 only the entries that have an exact match of March 3, 2019, will be returned. Being close by a day or two or perhaps matching on March 3 of a different year does not matter. How a query is written determines absolutely the values that are found.

Likewise, in query 2, if a vehicle has a purchase date of June 1, 2018, it would not be returned because the date is equal to June 1, 2018, as opposed to being greater than (often symbolized as '>') June 1, 2018.

There are a collection of other symbols that represent the logical operators within SQL. SQL has perhaps a dozen standard operators that are common in other programming languages. These are logical symbols used for arithmetic and filtering of values. See the following table (some of these are obvious!):

Operators within SQL

Symbol	Name	Purpose
+	Plus	Addition as well as concatenation with string values
-	Minus	Subtraction
*	Multiplier	Times two values
/	Divisor	Division between two values
%	Modulo	Returns the remainder value after completing division
>	Greater Than	Logical comparison of two values with first being larger
<	Less Than	Logical comparison of two values with first being smaller than the second
=	Equals	Logical comparison of two values where they match
<=	Less than/equals	Logical comparison of two values where they match, or the first value is smaller
>=	Greater than/equals	Logical comparison of two values where they match or the first larger
<>	Not equals	Logical comparison to find un-matching values
!=	Not equals	Logical comparison to find un-matching values

Table 5.2 List of Operators within SQL

Wildcards

SQL has several characters that can be used to represent 'spaces' as opposed to values. These are frequently used when people know part of the value they are searching on or we are trying to quickly narrow a results set. Sometimes, we just do not want to keep typing everything out and know that the wildcards will save us time.

- **%** percent sign: wildcard for 'any number' of characters
- **_** underscore: wildcard for 1 character space only

Wildcards are very common in a WHERE clause as we filter data in a query. Wildcards work against all data types including those that are characters, numeric, and date.

The keyword LIKE immediately precedes the use of wildcards. See the next few examples:

- Write the SQL to return all customers with a last name beginning with letter 'L'

```
SELECT *
FROM tblCUSTOMER
WHERE Lname LIKE 'L%'
```

Results | Messages

	CustomerID	Fname	Mname	Lname	BirthDate	CustAddress	CustCity	CustState	PostalCode
1	4	Janey	NULL	Lundgren	1996-01-25	14351 N Oak Beach Walk	CENTRALIA	Illinois, IL	62801

- - Code Example 5.18: SQL code using keyword LIKE and % wildcard and result set

- Return all customers with the letter 'y' anywhere in their first name:

```
SELECT *
FROM tblCUSTOMER
WHERE Fname LIKE '%y%'
```

Results | Messages

	CustomerID	Fname	Mname	Lname	BirthDate	CustAddress	CustCity	CustState	PostalCode
1	1	Ivey	NULL	Hazekamp	1995-12-24	7279 North Lavender Lake Avenue	PETROLEUM	Indiana, IN	46778
2	3	Kenyetta	NULL	Terron	1995-09-27	6664 NW Arthur Terrace Drive	GRAND RIVER	Ohio, OH	44045
3	4	Janey	NULL	Lundgren	1996-01-25	14351 N Oak Beach Walk	CENTRALIA	Illinois, IL	62801

- - Code Example 5.19: SQL code using keyword LIKE and wildcards before and after search criteria

- Return all customers with the letter 'n' in the 3rd position of their first name:

```
SELECT *
FROM tblCUSTOMER
WHERE Fname LIKE '__n%'
```

Results | Messages

	CustomerID	Fname	Mname	Lname	BirthDate	CustAddress	CustCity	CustState	PostalCode
1	3	Kenyetta	NULL	Terron	1995-09-27	6664 NW Arthur Terrace Drive	GRAND RIVER	Ohio, OH	44045
2	4	Janey	NULL	Lundgren	1996-01-25	14351 N Oak Beach Walk	CENTRALIA	Illinois, IL	62801

- - Code Example 5.20: SQL code using keyword LIKE and two single-position wildcards ('_') to search on fixed 3rd position

- Return all customers with number '4' in the 2nd to last position of their postal code:

```
SELECT *
FROM tblCUSTOMER
WHERE PostalCode LIKE '%4_'
```

Results Messages

	CustomerID	Fname	Mname	Lname	BirthDate	CustAddress	CustCity	CustState	PostalCode
1	3	Kenyetta	NULL	Terron	1995-09-27	6664 NW Arthur Terrace Drive	GRAND RIVER	Ohio, OH	44045

- - Code Example 5.21: SQL code using both wildcards to find second-to-last character

SELECT statement without FROM?

While many books and videos found on YouTube will claim that the FROM clause is mandatory in any SELECT statement, the following three examples prove that to be false. Please type in the following and execute in a query window:

```
SELECT 36 * 42
GO
```

Results Messages

	(No column name)
1	1512

- - Code Example 5.22: SQL code example of a SELECT statement without a FROM clause

In Code Example 5.22, the SELECT statement acts like a calculator (it is). We often will need to determine quantities, percentages, discounts, sums and many other results from math (almost all of which are dynamic and volatile). We want to be comfortable with SQL as a trusted math partner that can reliably handle complex scenarios in a rapid-fire high-volume context. Keeping people from doing math is part of the job of SQL.

```
SELECT GETDATE()
GO
```

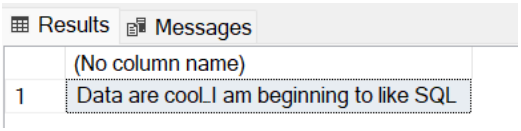
Results Messages

	(No column name)
1	2025-10-05 16:25:28.817

- - Code Example 5.23: SQL code example of SELECT statement calling a date function with no FROM clause

In Code Example 5.23, the SELECT statement is calling a date function. This is a very common task and key part of SQL. We want to programmatically obtain the date and/or time of many events, such as when an order or phone call occurred. If people must be involved, not only does it slow down processing by perhaps a million times over, but we also make mistakes and may 'flip' a day and month to make unintentional misinformation.

```
SELECT 'Data are cool…I am beginning to like SQL'
GO
```



- - Code Example 5.24: SQL code example of SELECT statement without FROM clause

In Code Example 5.24, the SELECT statement returns the string value of what was presented. This will become valuable in later lessons as we capture values based on queries and can then use the values in other processing. This way of thinking and operating (using the power of the database system as opposed to people) will significantly increase our ability to design analytical processing as opposed to doing the manual movement of data.

This is incredibly valuable as it allows for autonomous and independent learning (not only for the organizations we work for but for us individually as well).

There is a challenge for us to learn SQL while having only recently onboarded the fundamentals of database design as these are mutually dependent. A person cannot engage SQL without knowing the basics of database theory (not only familiar with what each table and columns contain but also how and why they are connected). Likewise, strictly learning about relational database theory without the context of the query language SQL is also difficult. In this book, we are learning both at the same time. Please be patient and open to not knowing everything right away; the material will become clearer in a little bit of time.

As we engage SQL beyond a basic SELECT statement, we must understand best practice before we are fully exposed to the conceptual theory. The following is a brief list of SQL traps or "gotchas" that seem to trip up beginners of database coding. Not all these concepts have been fully introduced yet—understand that the theory and coding of SQL are being presented in parallel and relatively soon everything will coalesce.

With that being stated, the following list and brief explanations will give you a "heads-up" on upcoming challenges. Briefly, each topic is shown from the perspective of a novice

(perhaps in the same position as you who may be engaging these topics for the very first time) with an attitude of “what are they and why should I care?”

With the following code examples, simply pay attention to the structure and order of keywords. There will be example code snippets in future sections where you will be able to “test drive” the execution of new concepts as they are introduced.

NULL

Additionally, basic SQL requires understanding of what NULL is; the key concept is NULL is not a number or value. It is the absence of a value (because it was either not provided or the value is unknown). Nothing *equals* NULL---not even another NULL value.

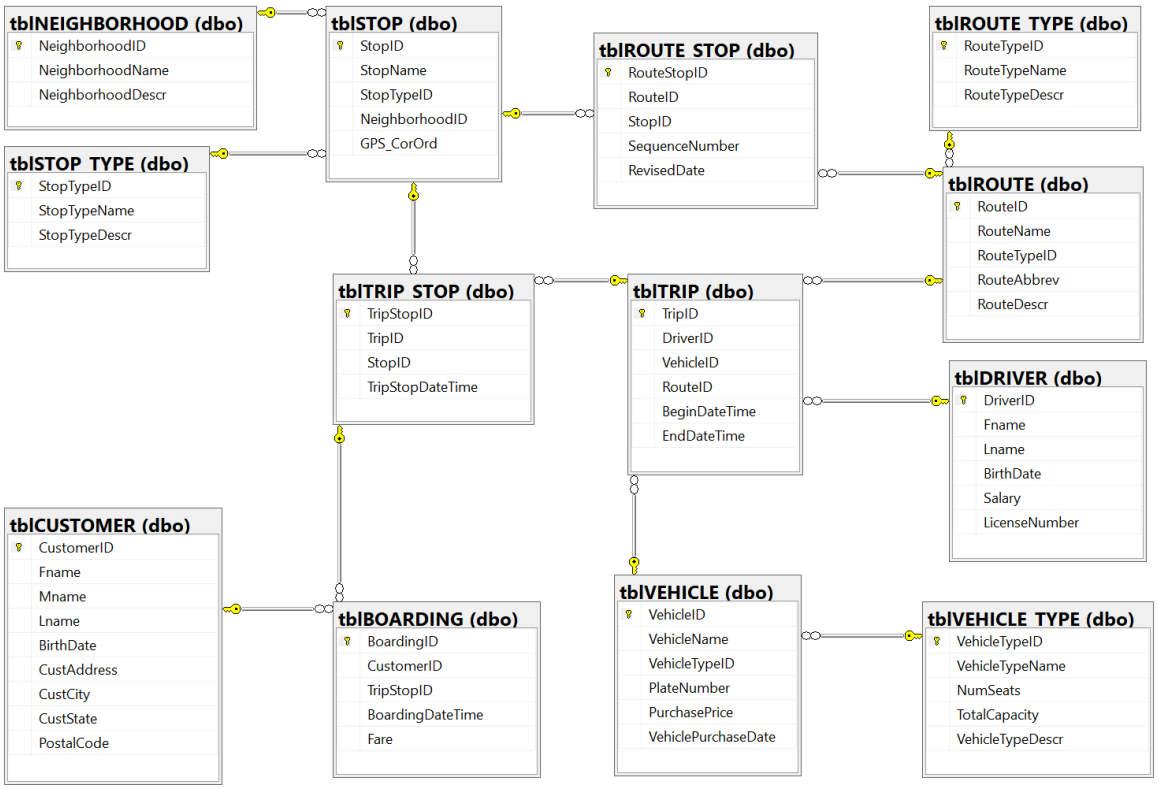
The important part of understanding NULL is how to treat math equations that happen to include a NULL; the short answer is we have choices.

- Choice #1: return NULL for the results (even if just 1 value out of a million is NULL)
- Choice #2: replace NULL values with a known value (like average of everything else)
- Choice #3: drop the NULL values from the calculation

Many scientific calculations will follow choices 1 and 2 while many business calculations (in my experience at least) will follow choices 2 and 3. It all depends on how precise the measurement needs to be and how comfortable a decision-maker is being approximate. Just be aware that NULL does NOT equal anything (especially not zero).

Now that you have been exposed to what SQL is and the very basics of how a simple query is written, you are ready to engage live code against a real database. The next section will provide you with “starter code” as well as prompts to execute the code; pay attention to the results and familiarize yourself with the process of engaging a database. Many complex and intricately layered questions can be answered with SQL as it is both logical and robust. The challenge for beginners is to recognize the patterns associated with setting up answers as well as learning the various keywords and their functionality. While this will not happen overnight, it can be accomplished in a relatively short period. The journey continues!

Now is an opportunity to try out writing queries referencing the data model in Figure 5.1 from METRO_TRANSIT. This may be brand new for you—do not be too concerned if there are many questions! The answers are included but you are encouraged to experiment by changing each query slightly to observe the impact of the data results set.



Review of Figure 5.1: Simplified Entity-Relationship Diagram of METRO_TRANSIT

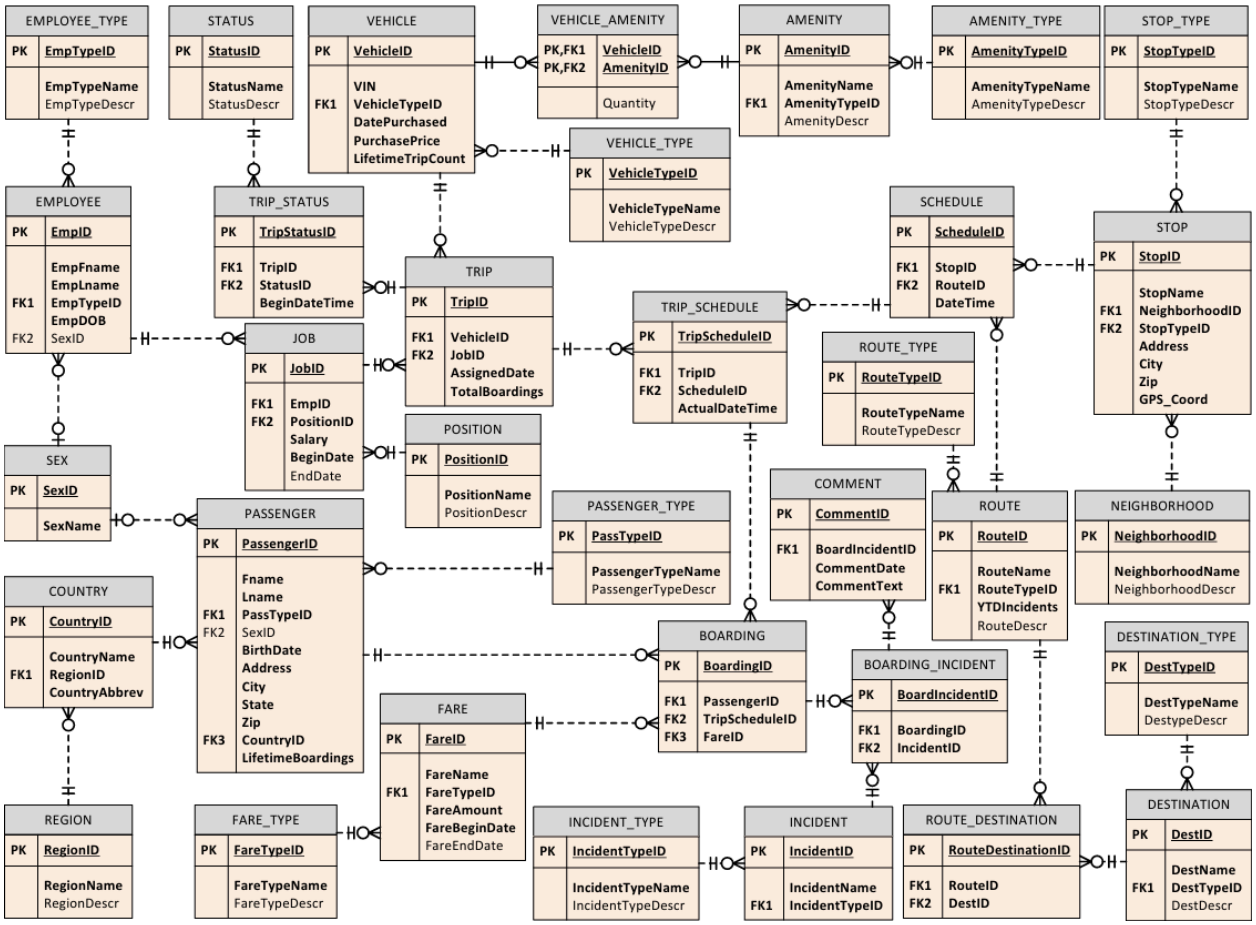


Figure 5.2: Fully Normalized Entity-Relationship Diagram of METRO_TRANSIT

PRACTICE

Time to get your hands dirty! Please remind yourself of the schema for the fully developed METRO_TRANSIT diagram in Figure 5.2 above.

Please visit the following link hosted by this book's publisher to download relevant files for the coding portion of this chapter:

https://github.com/ava-orange-education/ultimate-SQL-for-Data-Science-and-Analytics

Here you will find more than 8 different METRO_TRANSIT databases populated with data drawn from the transit systems serving the prominent cities listed below:

- New York City
- London
- Istanbul

- Dubai
- Seoul
- Delhi
- Shanghai
- Mumbai

The exciting part about this schema is the ability to scrape data from real-world transit systems from around the world to provide a familiar and relevant query-writing experience for people around the world. Ideally, you have also shared an experience on one of these beautiful transit systems!

Regardless of which METRO_TRANSIT database you have chosen to work with, the following 8 queries are based on the generic schema that all iterations of METRO_TRANSIT share. Each database has elements of real data pulled from one of the 10 or so transit systems, including data on routes, vehicles, neighborhoods, stops, and schedules. All data concerning passengers, drivers, boardings, and incidents are fabricated!

The purpose of these data sets is to provide a learning platform that is familiar to your lived experience (hopefully having interesting or at least familiar data chips away at the obstacles of learning something new we all face). Additionally, METRO_TRANSIT has enough data to challenge all learners with a full range of SQL commands while giving a safe place to explore and practice independently without fear of hurting anything.

The first set of questions are centric to the data that was input by hand from this chapter. Questions from future chapters will be centric to specific METRO_TRANSIT databases from various cities around the world.

Chapter 5: SQL Challenges

1) Write the SQL code to determine which customers are from the zip code 80471.

2) Write the SQL code to determine which customers have 'Blvd' in their address.

3) Write the SQL to determine which employees have last names that match the pattern 'F**R' (read 'F in position one and R in position four') and reside in either Florida or Texas.

4) Write the SQL to determine which employees have a first name beginning with the two letters 'Ra' and live in California.

5) Write the SQL code to determine which customers have 'Leaf' in their address and have the two letters 'ba' sequentially together in any part of the city they live in.

6) Write the SQL query to determine which employees were born after November 5, 1996.

7) Write the SQL to determine which vehicles have a purchase price less than $1,250,000 and were purchased after October 1, 2019.

8) Write the SQL to return each STOP that is in the city of Bellevue in Washington state or is in Zip code of '98102'

ANSWERS TO SQL CHALLENGES

Great effort tackling SQL perhaps for the first time! This section will walk you through each question and reinforce SQL basics.

1) Write the SQL code to determine which customers are from the zip code 80471.

```
SELECT *
FROM tblCUSTOMER
WHERE ZIP = '80471'
```

2) Write the SQL code to determine which customers have 'Blvd' in their address.

```
SELECT *
FROM tblCUSTOMER
WHERE Address LIKE '%Blvd%'
```

3) Write the SQL to determine which employees have last names that match the pattern 'F**R' (read 'F in position one and R in position four') and reside in either Florida or Texas.

```
SELECT *
FROM tblEMPLOYEE
WHERE State IN ('Florida, FL', 'Texas, TX')
AND EmpLname LIKE 'F__r%'
```

4) Write the SQL to determine which employees have a first name beginning with the two letters 'Ra' and live in California.

```
SELECT *
FROM tblEMPLOYEE
WHERE State = 'California, CA'
AND EmpFname LIKE 'Ra%'
```

5) Write the SQL code to determine which customers have 'Leaf' in their address and have the two letters 'ba' sequentially together in any part of the city they live in.

```
SELECT *
FROM tblCUSTOMER
WHERE Address LIKE '%leaf%'
AND City LIKE '%ba%'
```

6) Write the SQL query to determine which employees were born after November 5, 1996.

```
SELECT EmpID, EmpFname, EmpLname, EmpDOB
FROM tblEMPLOYEE
WHERE EmpDOB > 'November 5, 1996'
```

7) Write the SQL to determine which vehicles have a purchase price less than $1,250,000 and were purchased after October 1, 2019.

```
SELECT VehicleID, VIN,DatePurchased, PurchasePrice
FROM tblVEHICLE
WHERE DatePurchased > 'October 1, 2019'
AND PurchasePrice < 125000
```

8) Write the SQL to return each STOP that is in the city of Bellevue in Washington state or is in Zip code of ‘98102’

```
SELECT *
FROM tblSTOP
WHERE (City = ‘Bellevue’
AND State = ‘Washington’)
OR ZIP = ‘98102’
```

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 or exploring your reactions. These questions and your responses cut to the essence of this chapter:

- How is the structure of SQL easy (or not) for you to remember the sequence and syntax of particular commands?

- Do you find writing queries or the process of searching through data to answer business questions interesting? Why or why not?

- Try and explain SQL to someone with zero knowledge or experience with coding or databases; what would you highlight as the advantages of SQL? What are the obstacles or disadvantages of using SQL from what you have seen so far?

- What parts of your daily life (job, household, schoolwork, or extended network of family and friends) might be made more manageable **IF** all of the relevant data was available in a well-designed relational database and effectively 'one query away' from learning more about everything?

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!

- Frame the sequence of commands and the overall structure of an example query in a few short paragraphs without referencing your notes or the data model. What are the primary or key commands found in nearly every basic query?

- What are the limitations of the basic SQL query structure that we have seen so far? Knowing that most businesses need sophisticated analyses across a range of data (hint: across many entities!), what is the level of complexity can we currently address with the knowledge of SQL presented so far?

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. Let's begin with a challenge to structure data collection to keep track of a large-scale metropolitan transit system:

- This is a challenge to design and code a small yet original database from scratch! Leverage the design skills we have seen from previous chapters as well as the SQL coding skills to CREATE tables and build an original database.

- Consider some of your personal hobbies, interests, or perhaps a business idea; these make wonderful beginning ideas for a database. Go through the step-by-step process of the Conceptual, Logical, and Physical design phases as demonstrated in this textbook; feel free to implement your code on the same server that queries are being executed on!

Conclusion

SQL is a very powerful language and tool for answering sophisticated and nuanced questions buried within complex systems. We have effectively built a real database on a central server with only a little bit of starter code! Your diligence and determination is commendable; if data science were easy, everyone would have a great job at Google, Microsoft, Amazon, Oracle or NASA. The basics of SQL are behind us!

We have executed simple queries using SELECT, FROM, and WHERE clauses; this is a solid foundation upon which great skills will follow. This may not have been the easiest chapter to follow. The difficulty is compounded when we struggle to frame the query in a manner that a database management system can effectively interpret.

Next is Chapter 6 *Intermediate* SQL. Not only are we building on the progress of beginning SQL here, but we will also explore the nuances of perhaps a dozen commands and functions that build on the power of SQL. We will see how to connect tables together using the JOIN command to significantly increase the sophistication and complexity of the queries we can write. Also, we will be introduced to aggregate and date functions that will leverage GROUP BY and the HAVING clauses.

The best way to complete your learning objectives successfully is to have consistent practice and work sessions to test your comprehension. You have met the challenges so far, keep on charging!