

Contents

Iteration Introduction	3
Adding Attributes to the DBMS Physical ERD	5
Driving Questions	5
Example of Adding Attributes to the Car EERD.....	6
Car EERD With Attributes.....	8
Adding Attributes to Your DBMS Physical ERD	8
TrackMyBuys Attributes	8
Normalizing DBMS Physical ERDs	11
Normalizing Your DBMS Physical ERD.....	11
TrackMyBuys Normalization	11
Minimum Entity Check.....	13
SQL Scripts.....	14
Oracle Scripts	14
SQL Server Scripts	18
Postgres SQL Scripts	21
Tying it Together	24
Creating Tables from a DBMS Physical ERD.....	24
Car EERD With Attributes.....	25
Car CREATE TABLE Statement.....	25
Ferrari CREATE TABLE Statement.....	26
Creating your Tables, Columns, and Constraints.....	26
TrackMyBuys Create Script	26
Indexing Databases	27
Deciding Which Columns Deserve Indexes.....	30
Primary Keys.....	30
Identifying Primary Keys	31
Foreign Keys	31
Adding Foreign Key Indexes	31
Query Driven Index Placement	32
Query by Song Name	34
Query by Album Name.....	35
Query for Albums by Song Length	35

Indexing Summary.....	36
Identifying Columns Needing Indexes for your Database	36
TrackMyBuys Indexing	36
Creating Indexes in SQL.....	38
Creating a Non-Unique Index.....	38
Creating a Unique Index.....	38
Car CREATE TABLE Statement	38
Car Price Index Creation.....	39
Car VIN Index Creation	39
Creating Indexes in your Database	39
TrackMyBuys Index Creations	39
Summary and Reflection	40
TrackMyBuys Summary and Reflection	40
Items to Submit.....	41
Evaluation.....	42

Iteration Introduction

In Iteration 3, you created a DBMS physical ERD that contains primary and foreign keys, which gives you the structure of your database, but does not provide a place to store data. Databases with no data are not useful, so, in this iteration, we add attributes to our DBMS physical ERD. Attributes allow us to store all kinds of useful data. We will then normalize our logical design to reduce or eliminate data redundancy, saving us from many problems during implementation. We then proceed to create our tables and constraints in SQL, and index our tables so that accessing them is performant. Our database structure comes to life in this iteration!

To help you keep a bearing on where you have been and where you are headed, let's again look at an outline of what you created in prior iterations, and what you will be creating in this and future iterations.

Prior Iterations	Iteration 1	<p><i>Project Direction Overview</i> – You provide an overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it.</p> <p><i>Use Cases and Fields</i> – You provide use cases that enumerate steps of how the database will be typically used, also identify significant database fields needed to support the use case.</p> <p><i>Summary and Reflection</i> – You concisely summarize your project and the work you have completed thus far, and additionally record your questions, concerns, and observations, so that you and your facilitator or instructor are aware of them and can communicate about them.</p>
	Iteration 2	<p><i>Structural Database Rules</i> – You define structural database rules which formally specify the entities, relationships, and constraints for your database design.</p> <p><i>Conceptual Entity Relationship Diagram (ERD)</i> – You create an initial ERD, the universally accepted method of modeling and visualizing database designs, to visualize the entities and relationships defined by the structural database rules.</p>
	Iteration 3	<p><i>Specialization-Generalization Relationships</i> – You add one or more specialization-generalization relationships, which allows one entity to specialize an abstract entity, to your structural database rules and ERD.</p> <p><i>Initial DBMS Physical ERD</i> – You create an initial DBMS physical ERD, which is tied to a specific relational database vendor and version, with SQL-based constraints and datatypes.</p>
Current Iteration	Iteration 4	<p><i>Full DBMS Physical ERD</i> – You define the attributes for your database design and add them to your DBMS Physical ERD.</p> <p><i>Normalization</i> – You normalize your DBMS physical ERD to reduce or eliminate data redundancy.</p> <p><i>Tables and Constraints</i> – You create your tables and constraints in SQL.</p> <p><i>Index Placement and Creation</i> – To speed up performance, you identify columns needing indexes for your database, then create them in SQL.</p>
Future Iterations	Iteration 5	<p><i>Reusable, Transaction-Oriented Store Procedures</i> – You create and execute reusable stored procedures that complete the steps of transactions necessary to add data to your database.</p> <p><i>History Table</i> – You create a history table to track changes to values, and develop a trigger to maintain it.</p> <p><i>Questions and Queries</i> – You define questions useful to the organization or application that will use your database, then write queries to address the questions.</p>

Before you proceed, it's a good idea to first revise what you completed in Iteration 3, if you noticed any tweaks or enhancements that will benefit your design. You will be implementing your design in SQL this week, and so changes will become a little more expensive; it's better to make them now rather than later.

Adding Attributes to the DBMS Physical ERD

After all of the relationships in the conceptual ERD have been mapped to a DBMS physical ERD, the next step is to add attributes to the entities. After all, the primary purpose of a database is to provide long-term data storage, so a database that contains only primary and foreign keys, but no other fields, is not useful. The structural database rules, the initial conceptual ERDs, and initial DBMS physical ERDs you have seen thus far have not identified any attributes. The focus thus far has been on the entity and relationship structure, because you generally want to create that structure first before filling in the details. We need to learn more about adding attributes.

Just as with the entity and relationship structure, we rely on analysis of the organization and on how the application will be used to know what attributes we need. However, adding attributes is much less structured than creating the entity and relationship structure. We rely on one somewhat loose principle: *add the attributes that support the data the organization needs based on how the database will be used.* We do not need to create a list of detailed rules which identify the attributes first, and we do not need to add the attributes to the conceptual ERD first.

Because we are working with a DBMS physical ERD, we are now very close to implementing the SQL in our chosen database. Therefore, adding attributes requires involved knowledge of how databases use data, and exactly what the database will store. In contrast, the conceptual ERD only requires us to understand the organization and database usage at a higher level. Adding attributes is an iterative process and requires paying attention to specific details.

When you create a database for an organization, you may be assisted by analysts who help perform the organizational analysis. In that scenario, you and the analysts have many discussions, and the analysts answer many of your questions, to arrive at a suitable list of attributes. If you develop something by yourself or on a small team, such as a mobile application or other small project, you may not have the luxury of analysts; in that case you are both the designer and the analyst. In this course, you are your own team and play the part of answering your own design questions. You will need to do your own research, and make intelligent decisions.

Driving Questions

Here are some questions you can ask yourself to decide what attributes you need.

Question	Reasoning
What fields do other similar applications and databases store?	Taking a look at similar applications and databases can give you many ideas for attributes for your own database. For example, if you are creating a database that tracks people's workouts, then take a look at existing websites and mobile applications that help track people's workouts. There will be many fields you can identify from doing so.
What fields are obvious for my entities?	There are many obvious attributes for many entities. For example, if you have any kind of person entity, you will likely

	need name attributes such as FirstName and LastName. If you have any kind of address entity, you will likely need the standard fields such as Street1, City, and so on. Many entities need obvious fields.
What fields are unique for my database?	You may have very specific ideas about your database, how it will store just the data you need. You know about these attributes because you know what you intend. For example, imagine someone is creating a database to track recent stock performance by hour the purpose of selling in just the right hour. Such a database will have a field to track the hour of the day, which is somewhat unique to that database, since many databases don't track hours.
What would be presented on a user interface that uses my database?	Since end users do not directly interact with the database but an application, what kinds of screens would you envision for such an application? Identifying what fields would be on those screens will help you identify needed attributes for your database.
What fields do the use cases require?	You have already defined use cases and the significant fields they require, so you can also look there to determine what attributes your database needs.

As you can see from the questions above, adding attributes centers around the driving principle of adding the attributes that support the data the organization needs based on how the database will be used, but the process is not an exact science.

Example of Adding Attributes to the Car EERD

Let's take a look at adding attributes to Car EERD we use earlier. Recall that our structural database rule for this is "A car is a Ferrari, Aston Martin, Lamborghini, or none of these." Because we haven't any additional details of what we are trying to store beyond the structural database rule and ERDs, let's define that now. Let's assume that we are creating a database for a car reseller that specializes in selling used, high-end cars. Let's further assume that this is only a subset of the database, a subset that stores just the car information but not the other important entities such Customer, Purchase, and the like. Armed with this information, I can now run through the questions.

Question	Reasoning
What fields do other similar applications and databases store?	Based upon my experience with websites such as http://kbb.com , http://cars.com , and many others, there are many obvious fields we need to store. I visited these sites to refresh my memory. We need to store a VIN number and a price for each car, of course. We need to store a make and a model. We need to store car mileage because customers will want to know. We need to store color as well. These are the basic attributes any reseller would need. If I could meet with the reseller the database is for, I would likely come up with more fields, but for now I'll stick with these basic ones.
What fields are obvious for my entities?	
What fields are unique for my	I looked up information on the Ferrari, Aston Martin, and

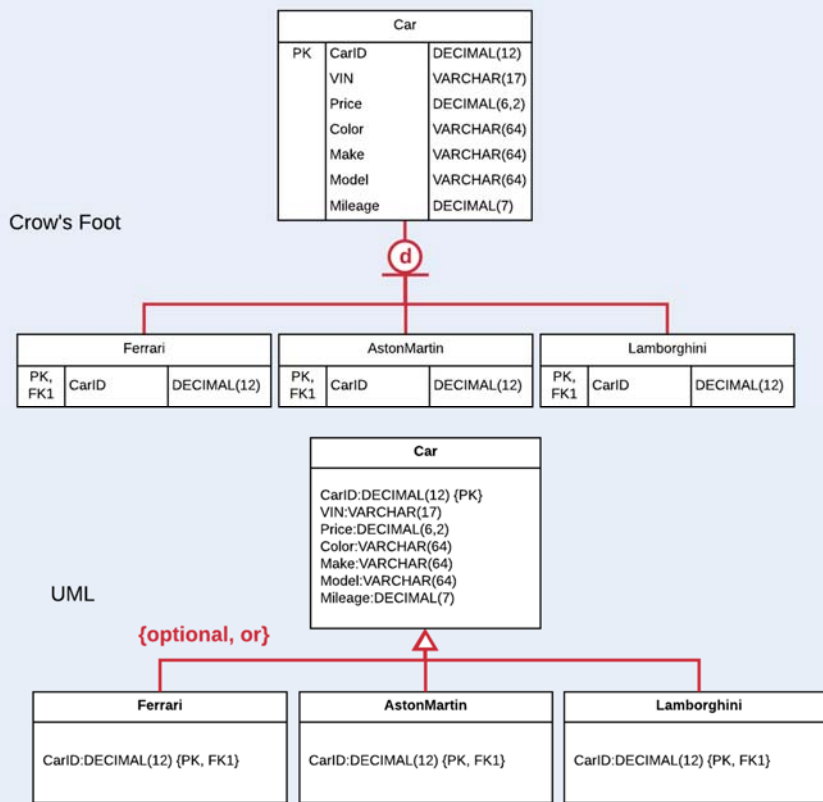
database?	Lamborghini lines, and while they each have unique items and properties in the real-world that customers care about, I could not identify useful fields to store about each of them in the database. Again, if I could meet with the reseller, I may come up with attributes to add here. I will stick with the more general fields already defined for the Car entity.
What would be presented on a user interface that uses my database?	I did not identify any additional fields from this question beyond what I have already included.

I've identified attributes I'd like to add, all of them to the Car entity, but there's still one more important step. I need to carefully select the datatype for each attribute, so that it will support storage of the data that will be in each field. Selecting the correct datatypes can take some research and time. I outline in a table below my datatype selections and reasoning for each attribute.

Attribute	Datatype	Reasoning
VIN	VARCHAR(17)	A quick web lookup reveals that VIN numbers are 17 characters in length. Also, VIN numbers are a combination of numbers and uppercase characters. So I use a VARCHAR(17) to support this attribute.
Price	DECIMAL(6,2)	Even these expensive cars will not cost more than \$999,999, so, I allow for 6 digits. Due to the prices of these cars, it is unlikely the reseller would like to use decimal points in their prices such as \$99,999.99; however, I cannot say for sure, so I allow for the standard two points after the decimal.
Color	VARCHAR(64)	I use VARCHAR(64) for color just in case there is a really long color name I am not aware of. Since VARCHAR does not take up storage for unused characters, this will not harm the database. I can see that the colors will be duplicated in this attribute, so this would be normalized out of the table during normalization, but in this part of the design it is in the table as is.
Make	VARCHAR(64)	The same as with Color, I use VARCHAR(64) in case there is a long make name.
Model	VARCHAR(64)	I use VARCHAR(64) here in case there is a long model name.
Mileage	DECIMAL(7)	While it's unlikely any car being sold will have over 999,999 miles, I allow for 7 digits just in case.

Now that I've identified the attributes and their datatypes, I can now enhance the Car EERD by adding the attributes into the DBMS physical ERD. The updated ERD is diagrammed below.

Car EERD With Attributes



The primary and foreign keys I mapped previously are still present in the diagram. The additional attributes for Car are added in addition to the existing keys. You can see this diagram taking shape now. Further, you can envision how we could create SQL statements to create the four tables in the diagram, along with all of their columns and keys. Exciting!

Adding Attributes to Your DBMS Physical ERD

At this point, you know enough to add the attributes important to your database to your physical ERD. Do so now, making sure to provide reasoning why you selected the attributes and datatypes. Below is a process I go through to add attributes to the TrackMyBuys DBMS physical ERD.

TrackMyBuys Attributes

I am now going through the process of adding attributes and their datatypes table-by-table. My choices and reasoning are in the table below.

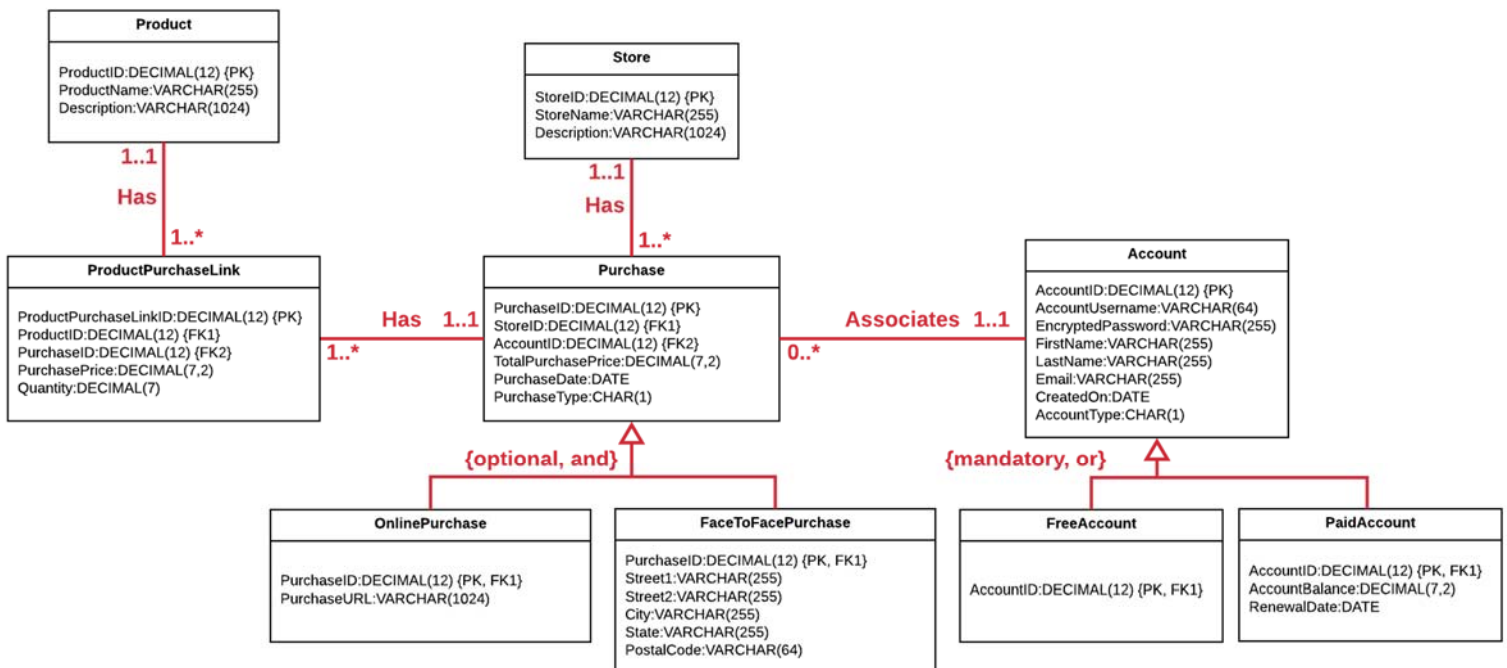
Table	Attribute	Datatype	Reasoning
Product	ProductName	VARCHAR(255)	Every product has a name which acts like the identifier for the product when people are looking it up in TrackMyBuys. I allow for up to

			255 characters just in case of something extraordinary.
Product	ProductDescription	VARCHAR(1024)	Every product may have a description. People may want to describe the product more than just with the name. I allow for 1,024 characters so that people can type in something long if they need to.
ProductPurchaseLink	PurchasePrice	DECIMAL(7,2)	When people buy a product, they buy it at a price. Though the price can change over time, there is a specific price they buy it at, which may include discounts. Although it's unlikely a price will ever be recorded in the millions, I allow for up to 7 digits. I also allow for the standard two decimal points.
ProductPurchaseLink	Quantity	DECIMAL(7)	When people buy a product, they can buy one or more of them. I allow for up to 7 digits as a safe upperbound.
Purchase	TotalPurchasePrice	DECIMAL(7,2)	When people make a purchase, the purchase can include one or more products. This attribute captures to total price including all products purchased. I allow for up to 7 digits as a safe upperbound, and the standard two decimal points.
Purchase	PurchaseDate	DATE	A purchase happens on a specific date.
Purchase	PurchaseType	CHAR(1)	In a prior iteration, I indicated that there can be at least two types of purchases, online and face-to-face. This attribute is the subtype discriminator to indicate which it is.
Account	AccountUsername	VARCHAR(64)	Every account has a username associated with it, which will be used to login to TrackMyBuys. I allow usernames to be up to 64 characters.
Account	EncryptedPassword	VARCHAR(255)	Every account has a password. It will be stored in encrypted text format in the database. 255 characters should be a safe limit to store encrypted text.
Account	FirstName	VARCHAR(255)	This is the first name of the account

			holder, up to 255 characters of the name.
Account	LastName	VARCHAR(255)	This is the last name of the account holder, up to 255 characters of the last name.
Account	Email	VARCHAR(255)	This is the email address of the account holder. 255 characters should be a safe upperbound.
Account	AccountType	CHAR(1)	As identified in a prior iteration, there are two types of accounts – free and paid. This attribute is the subtype discriminator indicating which it is.
PaidAccount	AccountBalance	DECIMAL(7,2)	This is the unpaid balance, if any, for the paid account. I allow for up to 7 digits, though it will likely never get near this high.
PaidAccount	RenewalDate	DATE	This is the date on which the account needs to be renewed with a new payment.

I feel I have captured all of the necessary fundamental attributes for TrackMyBuys in the table above. I see that I could be more detailed with storing account information balances and payment information. But given the use cases and structural database rules I've developed thus far, these attributes suffice.

Here is my ERD with the attributes included.



Each of the attributes have been added to their respective entities in the ERD. The previously added primary and foreign keys have also been retained. One item worth noting is that I did not identify any attributes necessary for the FreeAccount entity at this time. I may identify some as the application is further developed.

While there is room for more detail, I feel that this is a solid DBMS physical ERD for the use cases and structural database rules I have added thus far in the design.

Normalizing DBMS Physical ERDs

Once a conceptual ERD is mapped to a DBMS physical ERD, and attributes are added, there is only one more necessary step before implementation in SQL can occur – normalization. We want to normalize our entities before they are implemented to minimize data redundancy in our database. Recall that normalization is a formal process of eliminating data redundancy. Further recall that normalization works by identifying dependencies between columns, and moving keys and the values they determine into their own tables.

The scope and technical complexity on how to perform normalization from scratch is too broad and deep to be a part of this document. Please use the textbook, online lectures, and live classroom sessions to learn what normalization is and the steps to follow to normalize a table. Keep in mind that it is best practice to normalize tables to BCNF when possible. If a table is not normalized to BCNF for specific reasons, we should be aware of those reasons and make a conscious choice to do so.

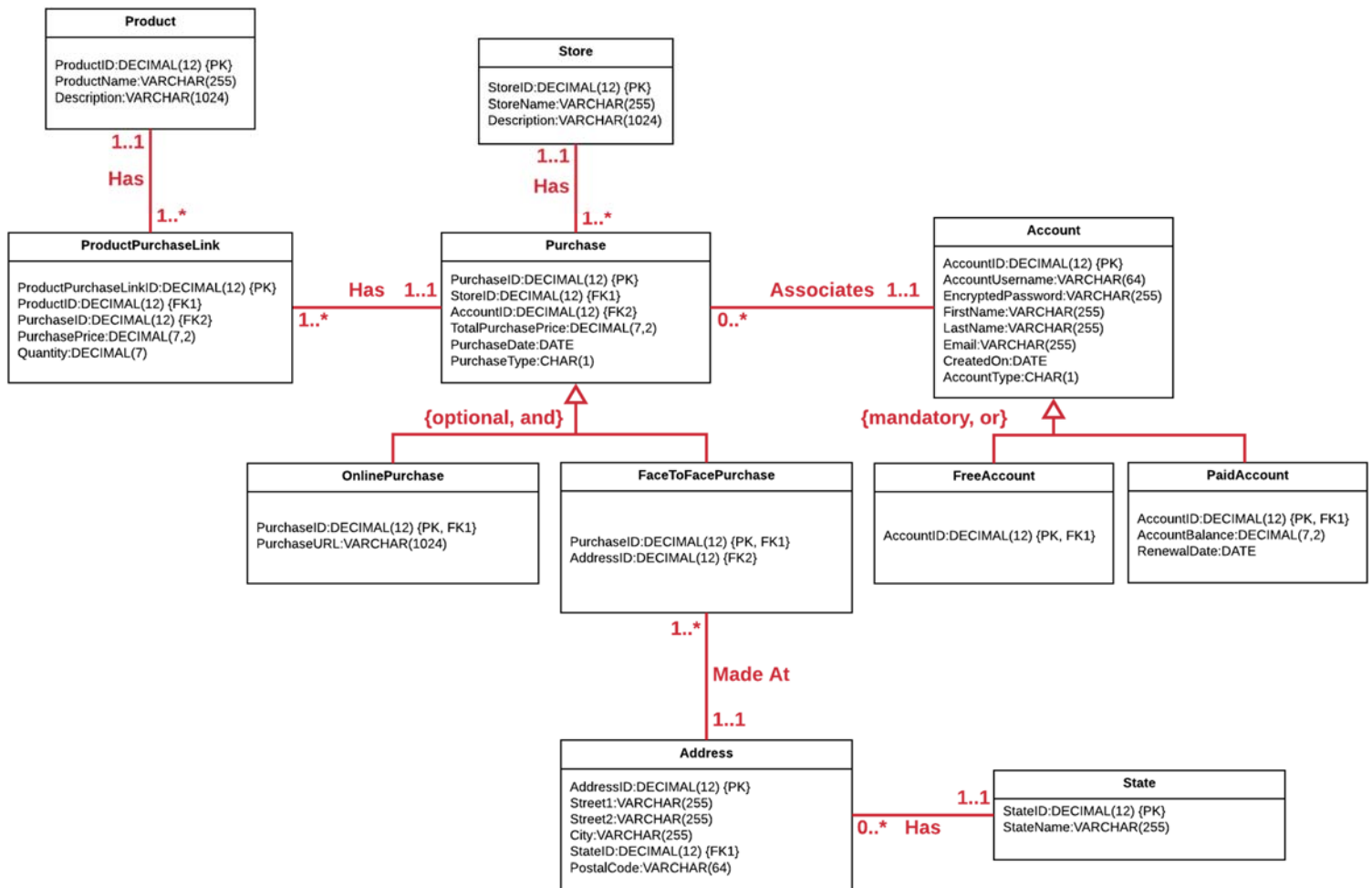
While normalization is performed on the DBMS physical ERD, it affects the conceptual ERD as well as the structural database rules. Why? Because normalization results in additional entities and relationships between those entities. Since it's important to keep the structural database rules, conceptual ERD, and DBMS physical ERD in sync, normalizing the DBMS physical ERD also results in changes to the structural database rules and the conceptual ERD.

Normalizing Your DBMS Physical ERD

You are very close to implementing your database in SQL. All that's left is to normalize your entities in your DBMS physical ERD, and update your conceptual ERD and structural database rules to match. Do so now by normalizing your DBMS physical ERD. Here is an example with TrackMyBuys.

TrackMyBuys Normalization

I notice only one place where there is redundancy in my physical ERD, and that is with the address information in the FaceToFacePurchase entity. If many purchases are made at that same face-to-face store, the address information will repeat. Here is my ERD with the address information normalized.



There are two additional entities after normalization – Address and State. By moving the primary address information into its own entity, I do not need to repeat the address every time a purchase is made. I can reference the address instead. Likewise, rather than repeating the state name every time a purchase is made, the address entity references the state entity instead.

The Address entity is not normalized to BCNF. In fact, I could create more tables that further breakdown street into its number and street name and normalize out street name. I could associate cities and states with postal codes. I could break out city so it does not repeat. I chose not to normalize address to full BCNF because it is not necessary for my database. That would add unnecessary complexity and add no real benefit.

The State entity is normalized to BCNF.

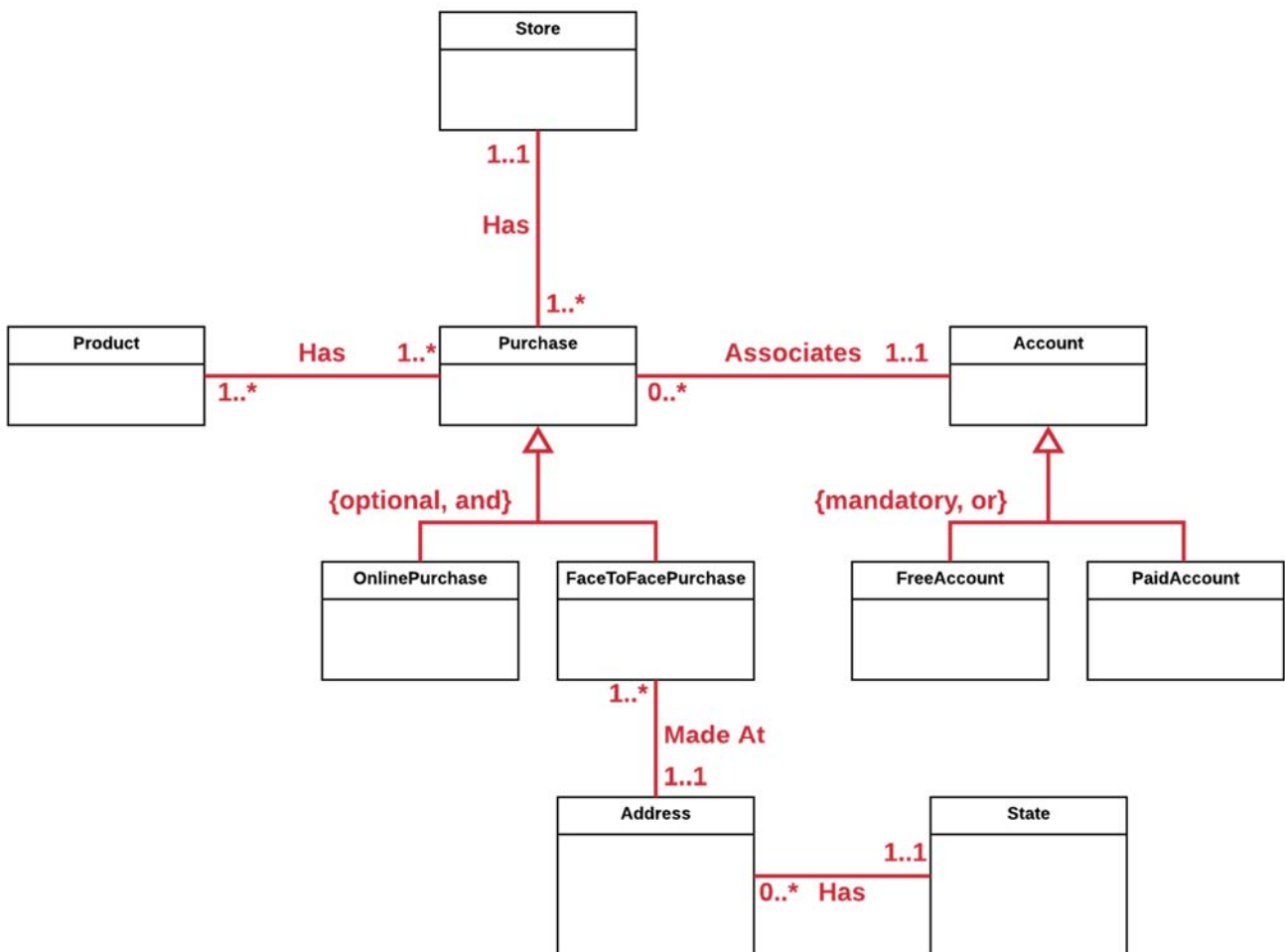
Below are my structural database rules modified to reflect the new entities. The new ones are italicized.

1. Each purchase is associated with an account; each account may be associated with many purchases.
2. Each purchase is made from a store; each store has one to many purchases.

3. Each purchase has one or more products; each product is associated with one to many purchases.
4. An account is a free account or a paid account.
5. A purchase is an online purchase, a face-to-face purchase, both, or none of these.
6. *Each face-to-face purchase is made at an address; each address is associated with one or more face-to-face purchases.*
7. *Each address has a state; each state may be associated with many addresses.*

I added #6 and #7 to reflect the new Address and State entities.

Below is my new conceptual ERD to reflect the new entities.



The Address and State entities are now included in the conceptual ERD, and the conceptual ERD is in sync with the structural database rules and the DBMS physical ERD.

Minimum Entity Check

After normalization, you should have at least 8 entities in your DBMS physical ERD to support the minimal complexity requirements for the term project. If you have less than this, do not worry. Simply add another use case or two, and carry the impact through your design into the structural database rules, conceptual ERD, and DBMS physical ERD. This should get you up to the minimum. Notice that 8 is

just a minimum; you may well have many more than this, which is just fine. The focus is your database meeting the requirements of the organization or application based upon the use cases provided, and not strictly the number of entities. However, a minimum of 8 helps ensure sufficient complexity.

For TrackMyBuys, I have 10 entities after normalization, so I feel comfortable that it meets the minimal complexity requirements for this project.

SQL Scripts

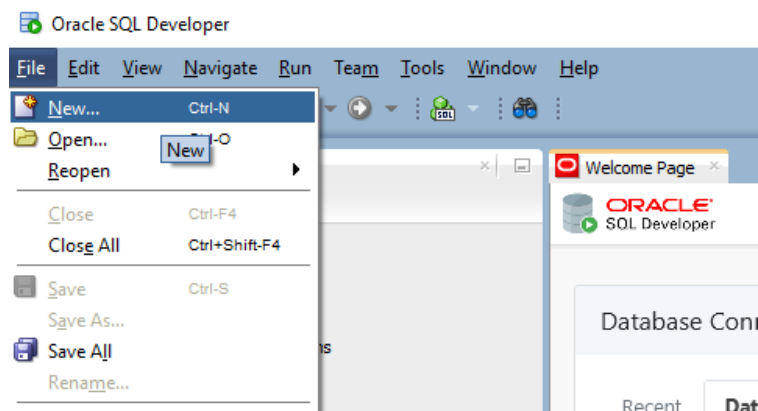
Before you dive into creating SQL code for your project, you need to know about saving your code in SQL scripts. A SQL script is simply a text file with a “.sql” extension at the end of the filename. It’s best practice to save your code in SQL scripts so that you can rerun the code anytime you need. For example, if you create several tables with SQL and save it in a script, then later decide you need to modify any of the table definitions, you could open up the script, make the modifications you need, and re-run the script. If you don’t have a script, you would need to recreate all of the SQL code again to make the modifications. The same goes for queries and stored procedures you create. You want to save those in scripts so you can reuse them as you need.

Saving and using SQL scripts is not difficult. In fact, the experience is very similar to using file-based applications such as Word, Excel, Pages, or Numbers. To save a script, you type your code into the buffer window of your SQL client, then use the client’s save feature to save it as a SQL script. Later, if you want to use the script, use your client’s file open feature to open the script. Once a script is open, you can run the commands in it with the client’s execute options, the options you are already familiar with. The process is simple and familiar.

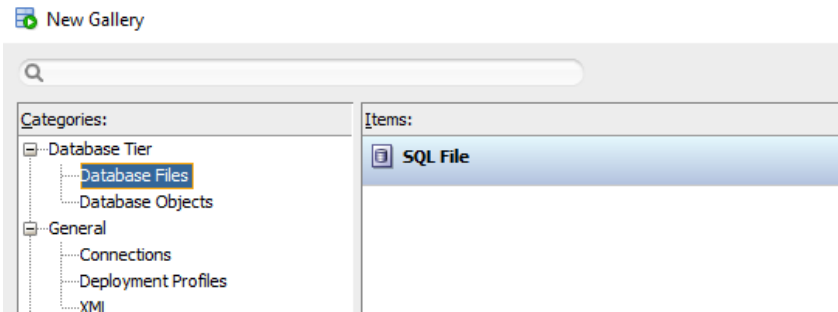
Let’s take a look at saving and opening scripts for the course’s supported databases. As we go through the examples, please keep in mind that as versions of the clients change, the precise names and locations of the menu options may change.

Oracle Scripts

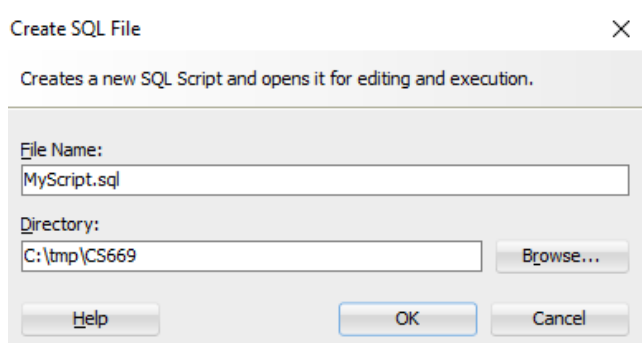
First, we’ll start with Oracle in Oracle SQL Developer. First, open a buffer window by selecting File/New....



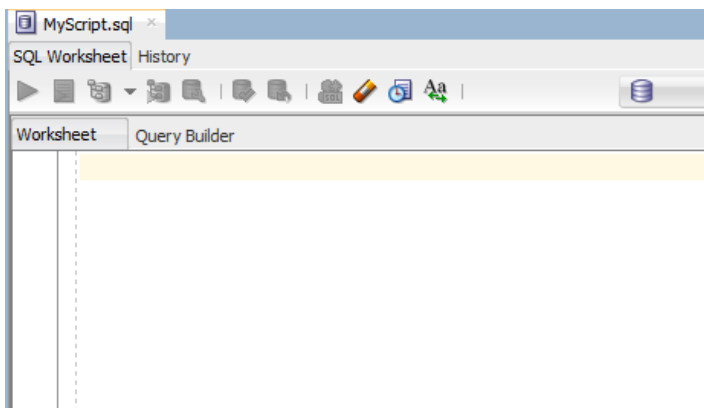
From there, select Database Files/SQL File and click the OK button.



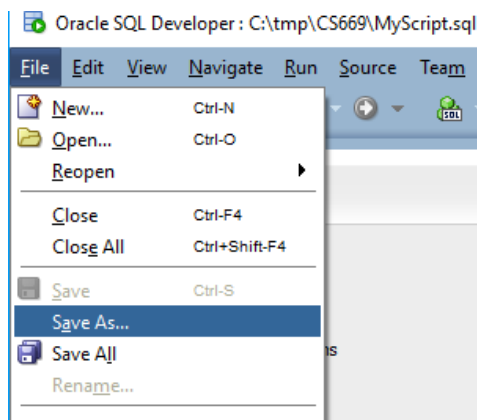
From there, give the file a name and choose a directory where you want to store the file. It's a good idea to store it in a directory you will remember. In the example below, I named my script "MyScript.sql", and put it into a C:\tmp\CS669 directory.



Once you click the OK button, a buffer (named a "SQL worksheet" in Oracle SQL Developer) appears where you can type SQL commands.

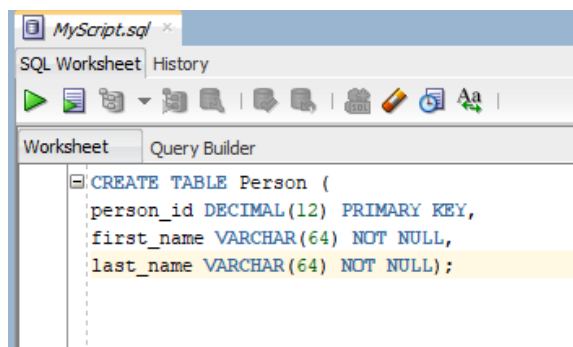


This is just one way to open a buffer window associated with a script. If you already have a buffer open and just want to save it to a script, you can select File/Save As... to save it to a script.

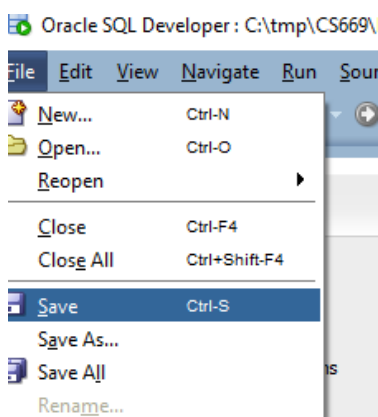


Either way, you've achieved the goal of opening a buffer window and tying that window to a specific SQL script saved on your drive.

Next, type in whatever SQL commands you need. In the example below, I add a simple create table statement for Person table, for illustrative purposes.



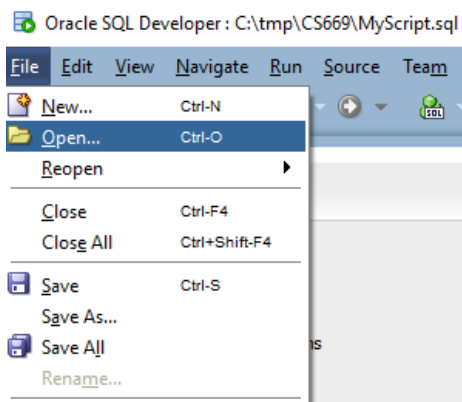
It's a good idea to save regularly so you don't lose work. You can either type Ctrl-S to save, or select File/Save....



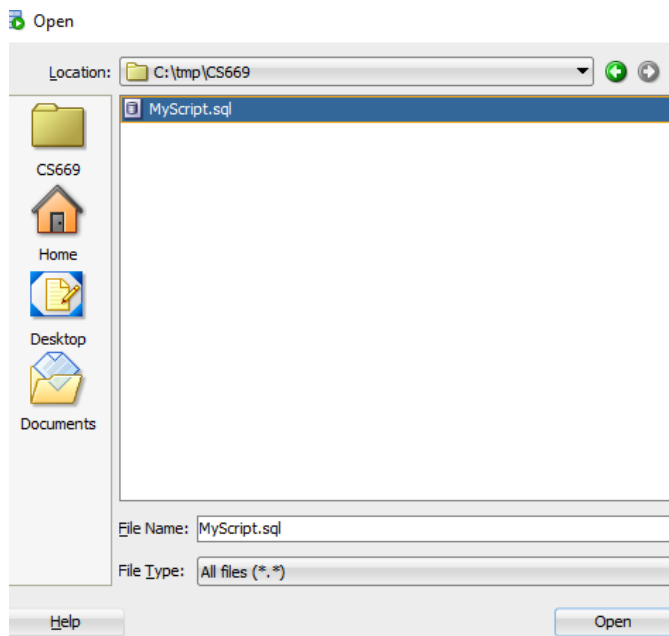
Either way, your latest commands will be saved to the file.

That's it! It's not too complex, and not much different than saving Word or Pages files. You are now saving your code to your drive and can work with it later, and will not lose your code the next time you close your client or restart your computer.

Later on, when you want to work with the script, you just use File/Open... to open up the script again.

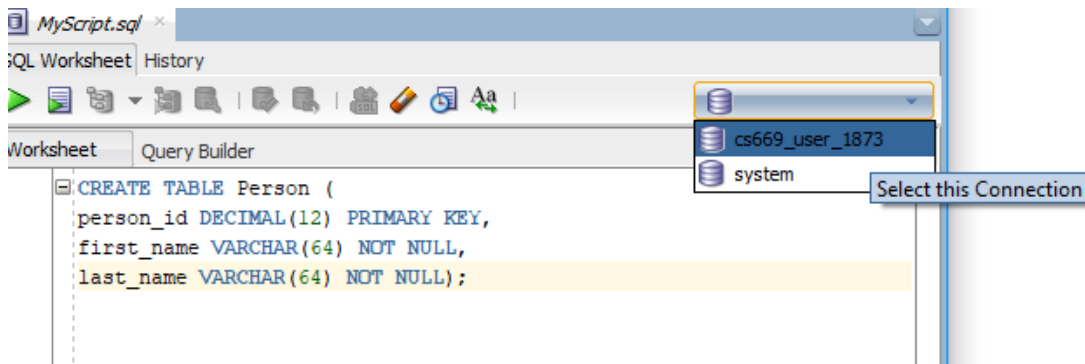


Select the file you want, and click the Open button.



And you'll see your file with your SQL commands in it again.

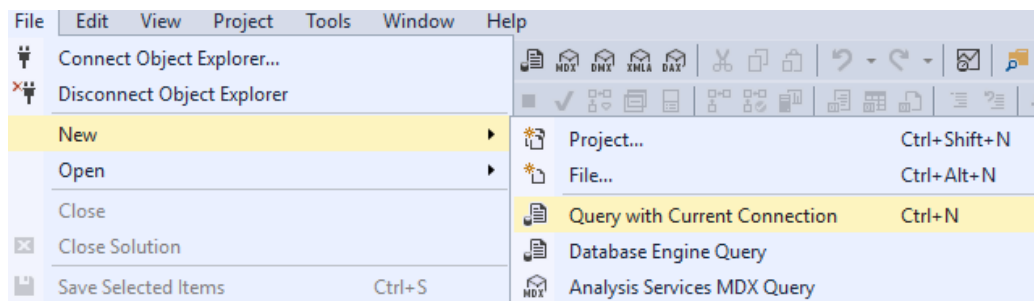
One other thing you need to know is that in order to execute your code, you need to tell Oracle SQL Developer which connection to use for it. On the right, there is a list of connections. Choose the one you want before executing your code.



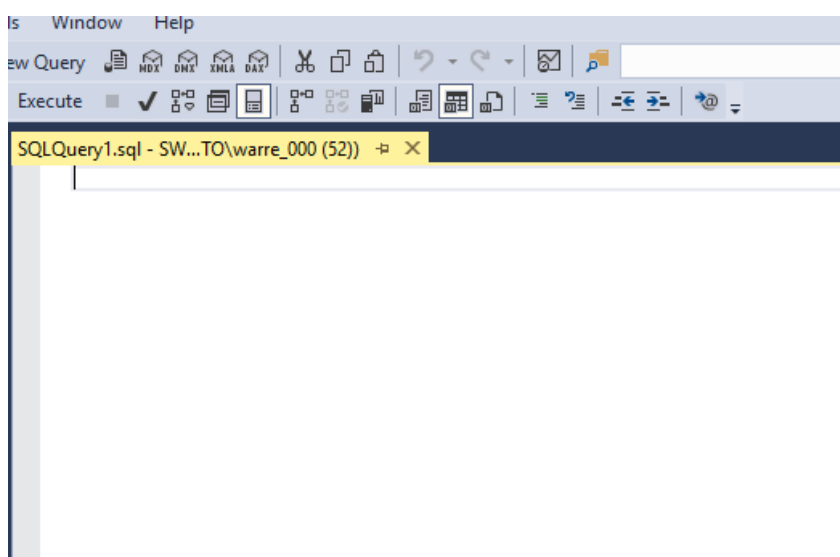
You can keep the same script open, and change connections if you need, before executing the commands therein.

SQL Server Scripts

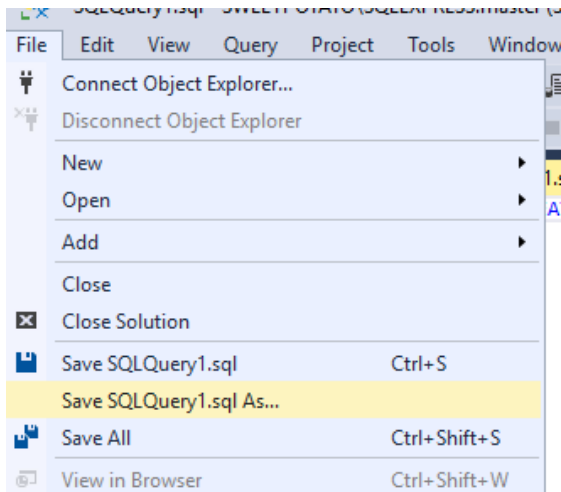
To open a buffer window with SQL Server Management Studio (SSMS), select File/New/Query with Current Connection.



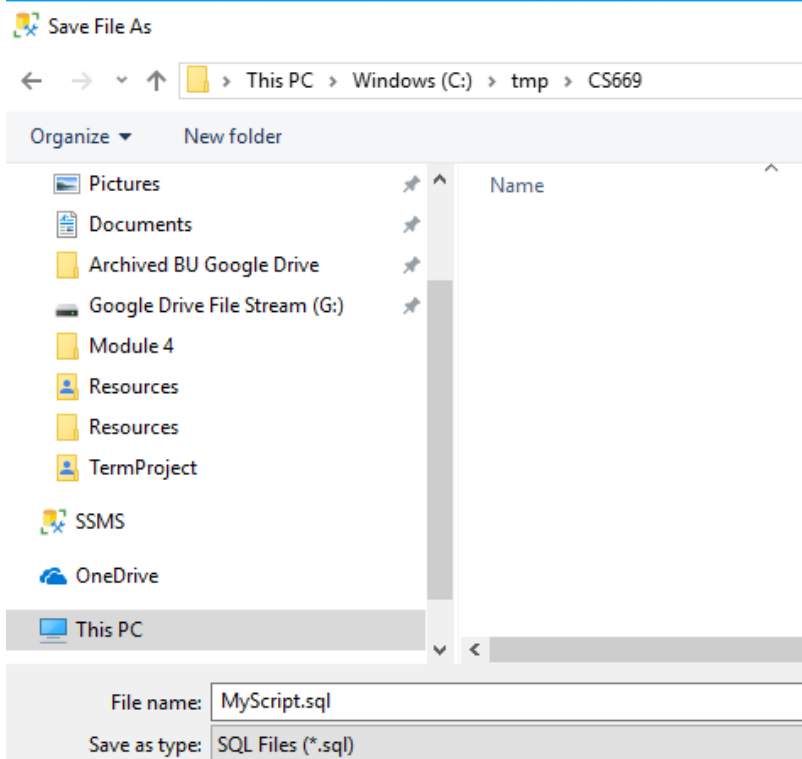
Note that when you start SSMS, it asks you to connect to a database, so you should already have a connection to your database. This command will create a new buffer window associated with that connection. You will see the buffer window such as the below.



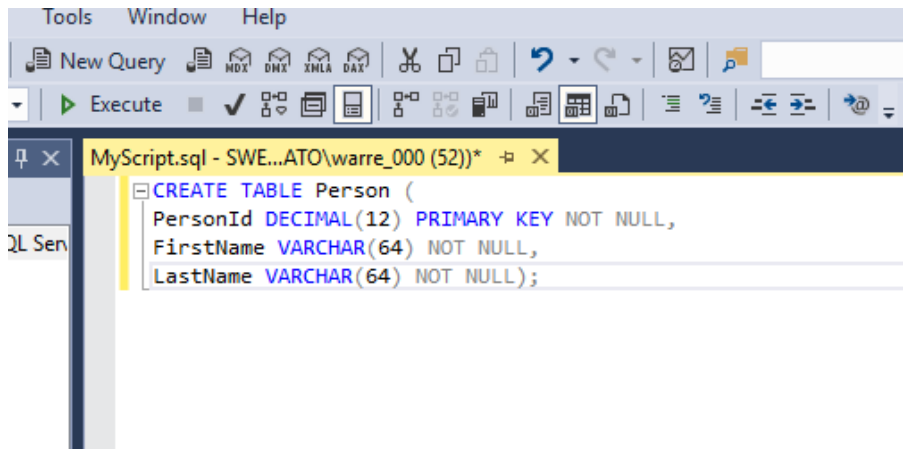
Next, you need to save the buffer to a file. To do so, click File/Save XYZ.sql As..., where XYZ is the default name of the script chosen by SSMS.



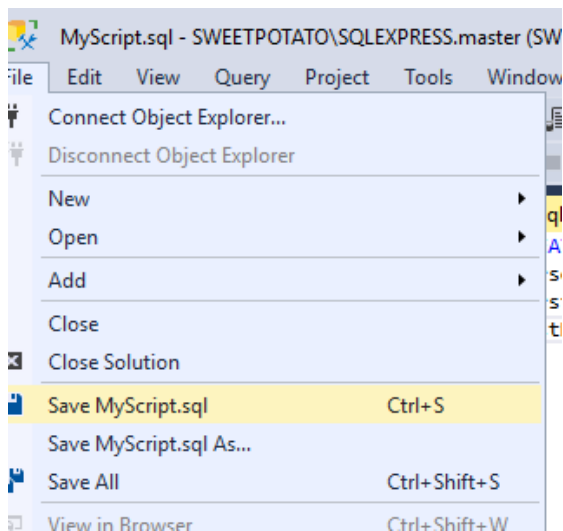
Then save it into a directory with the filename of your choosing. I chose to save it under C:\tmp\CS669\MyScript.sql, shown below.



At this point, the buffer is now associated with the file. Type in whatever SQL commands you need. In the example below, I add a simple create table statement for Person table, for illustrative purposes.

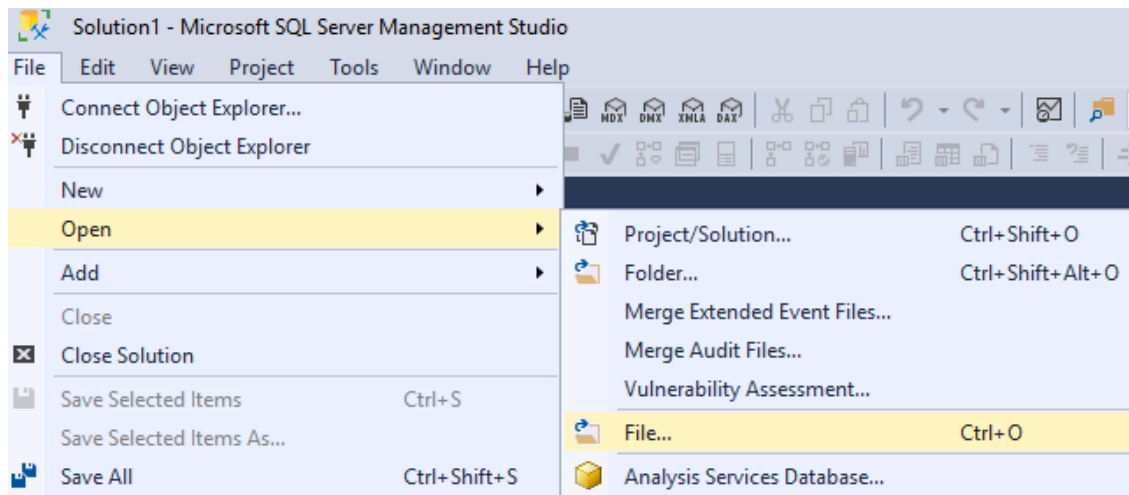


It's a good idea to save regularly so you don't lose work. You can either type Ctrl-S to save, or select File/Save MyScript.sql....

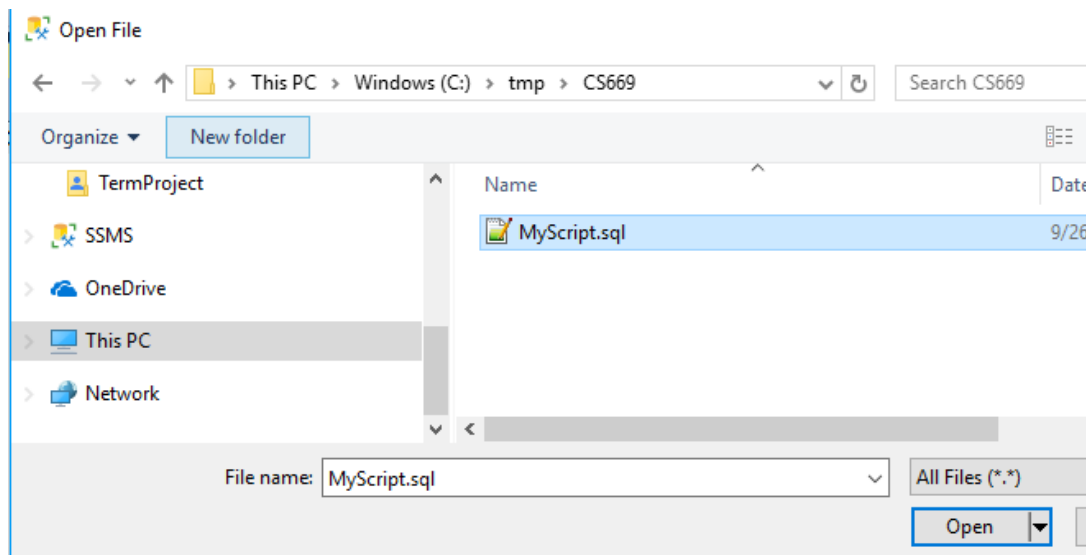


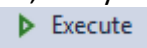
That's it! It's not too complex, and not much different than saving Word or Pages files. You are now saving your code to your drive and can work with it later, and will not lose your code the next time you close your client or restart your computer.

Later on, you may need to open up your script to continue working with it. To do so, click File/Open/File....



You will then need to navigate to your saved script and click the Open button.



After that, the script will be open again, and you can edit and execute the commands as needed. To execute the commands, just use the  **Execute** button like you have already been doing in the labs.

Postgres SQL Scripts

The first step in creating a SQL script in the pgAdmin tool is to connect to your database, because queries and scripts cannot be created otherwise. Open up pgAdmin and expand the PostgreSQL tree item first. When you do so, it will prompt you to enter your password that you chose during installation. Enter the password and confirm to continue.

Connect to Server

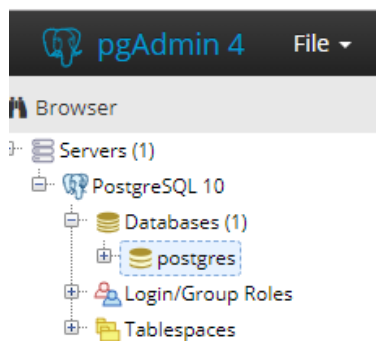
Please enter the password for the user 'postgres' to connect the server -
"PostgreSQL 10"

Password

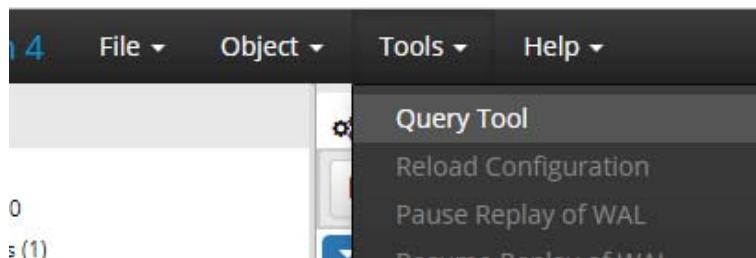
☐ Save Password

OK Cancel

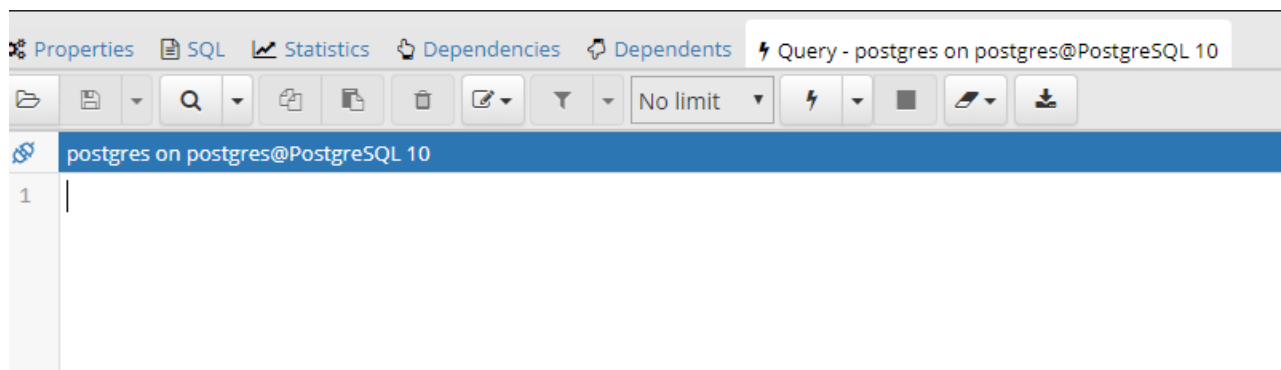
Once connected, you need to expand Databases and select the database you want to create the script for.



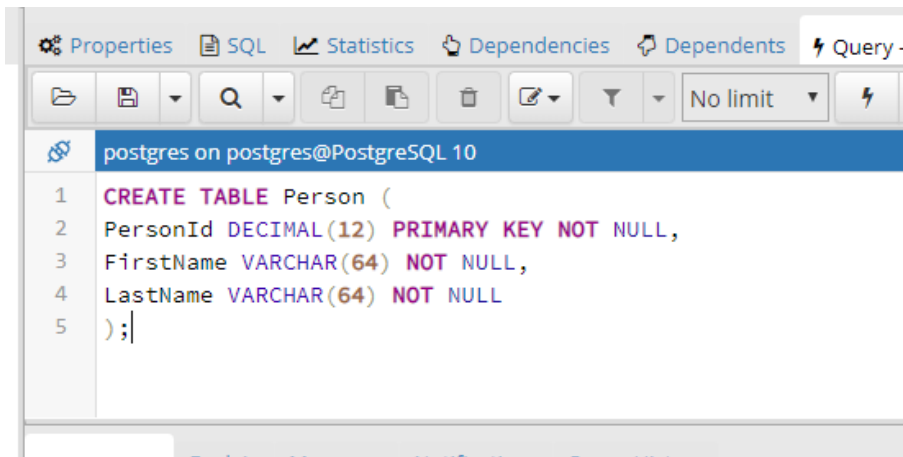
Once selected, the Tools/Query Tool option becomes enabled. Select it.




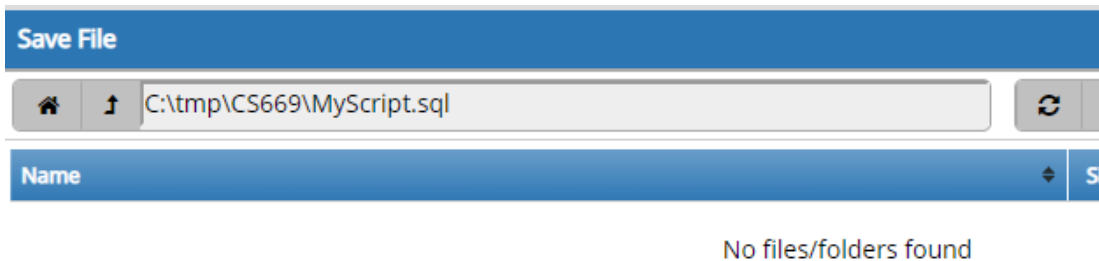
A buffer window appears where you may work with your SQL.




With pgAdmin, you first need to type something into the buffer before you can save it to a script. For illustrative purposes, I type in a command to create a simple Person table.




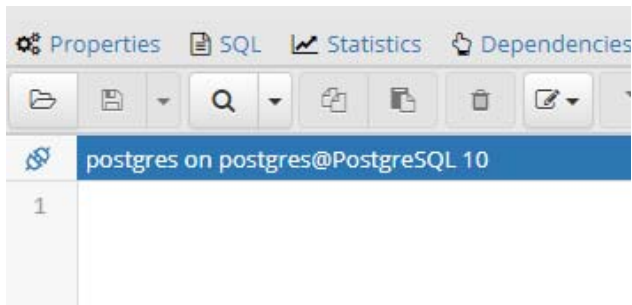
To save the buffer as a script, click the  icon. Then select the location and filename for the script. I opted for C:\tmp\CS669\MyScript.sql, as shown below.



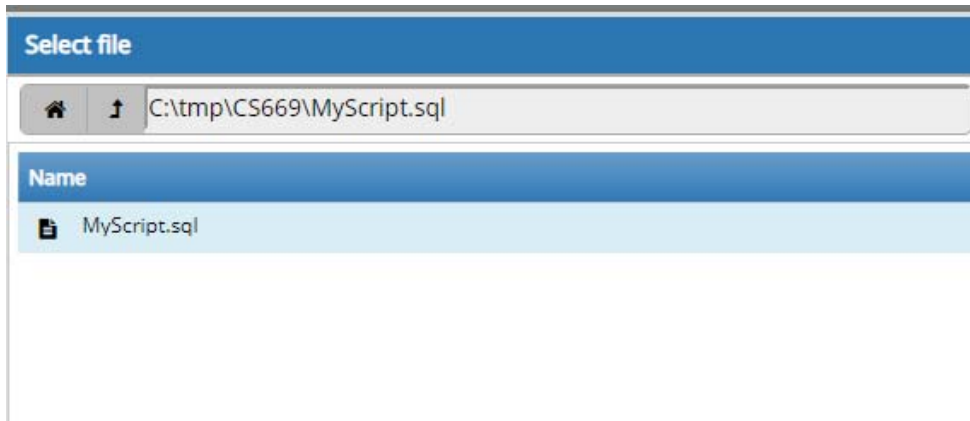
Make sure the click the Save button, and then your buffer will now be associated with this file. pgAdmin will tell you it successfully saved the file, and will also change the title of the buffer to your filename.

That's it! It's not too complex, and not much different than saving Word or Pages files. You are now saving your code to your drive and can work with it later, and will not lose your code the next time you close your client or restart your computer. Make sure to save your work often so you don't lose anything. To do so, just click the  icon whenever you need to save changes.

Later on, you may need to open up your script to continue working with it. To do so, open the query tool again with File/Query Tool, then click the  icon to open a saved file. It's the leftmost icon in the toolbar.



Once you do so, a file dialog will appear, and you choose the file you previously saved.



When you click the Select button, the file will open once again for you to continue editing and executing.

Tying it Together

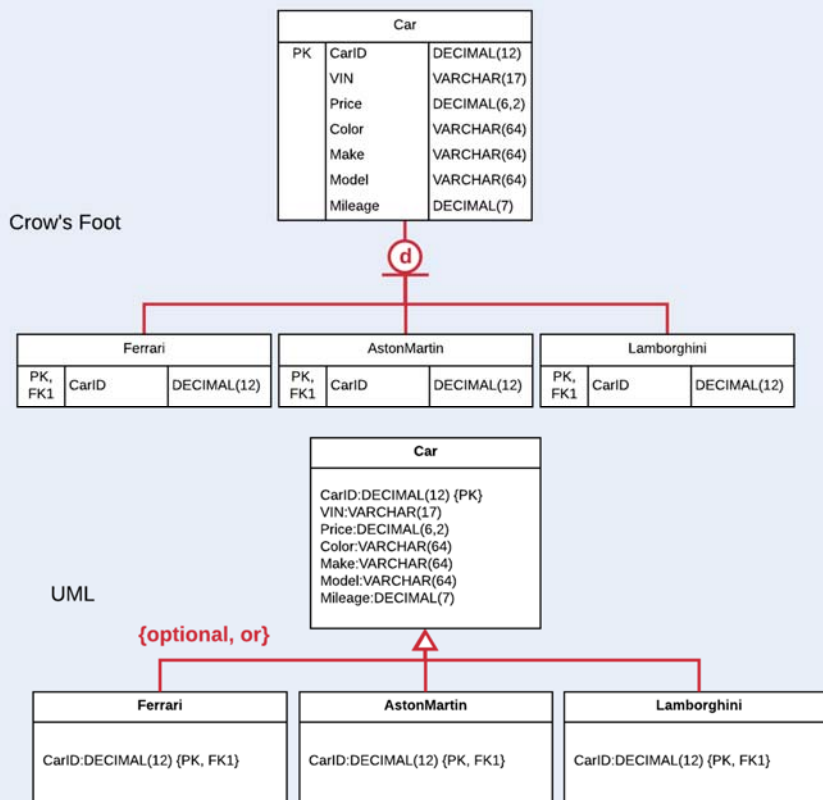
So how does all of this tie into your term project? Simply put, as you work on your SQL code, you want to save it off into a script so you can always edit and re-run them as needed. Imagine how frustrated you would be if you were to lose large amounts of SQL code, and you had to rewrite it all. You will have a SQL script containing your work, with a comment that briefly identifies and explains each section.

Creating Tables from a DBMS Physical ERD

As you may have predicted, it is straightforward to create the tables needed from a DBMS physical ERD. The entities in the ERD are created in SQL, one-for-one. Since the ERD already contains the attributes, datatypes, and constraints, creating the tables is a mostly mechanical process.

Let's again look at the car example from a prior section 4. Recall that the DBMS physical ERD looks as follows.

Car EERD With Attributes



To create the Car entity, we would use the following CREATE TABLE statement.

Car CREATE TABLE Statement

```

CREATE TABLE Car (
  CarID    DECIMAL(12) NOT NULL PRIMARY KEY,
  VIN      VARCHAR(17) NOT NULL,
  Price    DECIMAL(8,2) NOT NULL,
  Color    VARCHAR(64) NOT NULL,
  Make     VARCHAR(64) NOT NULL,
  Model    VARCHAR(64) NOT NULL,
  Mileage  DECIMAL(7) NOT NULL);
  
```

We use the entity name “Car” for the table name, and then create columns that are one-to-one matches of the entity’s attributes. We add the primary key constraint to CarID as indicated, and NOT NULL to the attributes we know should not be null (which for this table is all attributes).

If we want to create the Ferrari table for example, it would look as follows.

Ferrari CREATE TABLE Statement

```
CREATE TABLE Ferrari (  
  CarID    DECIMAL(12) NOT NULL PRIMARY KEY,  
  FOREIGN KEY (CarID) REFERENCES Car(CarID));
```

Notice that the CREATE TABLE statement for Ferrari has a one-to-one mapping with its entity definition in the ERD, including the primary and foreign key constraints. There is only one attribute in the entity – CarID -- so only one attribute in the CREATE TABLE statement.

How are the disjointness and completeness constraints implemented? The simple answer is that both disjointness and completeness do not affect the database structure, and are not implemented as structure. The business logic for the application would maintain both constraints. For example, the application that uses this Car database would not allow a person to designate a car as both a Ferrari and a Lamborghini; the relationship is disjoint since one car cannot be both. And the database structure itself, such as the primary keys and foreign keys and number of entities, does not change whether a relationship is disjoint, overlapping, partially complete, or totally complete.

As you can see, creating tables that correspond to entities is quite mechanical and straightforward. The hard work is found in designing the database with structural database rules, and with conceptual and DBMS physical ERDs. Implementing SQL from the DBMS physical ERD is relatively easier.

Creating your Tables, Columns, and Constraints

Your task now is to create your term project SQL script, and add all table, column, and constraint creations designed in your DBMS physical ERD. Use the one-to-one method previously illustrated. Note that to make the script re-runnable, it is best practice to add DROP TABLE statements to the beginning of the script which drops all of the tables, then to follow that with the CREATE TABLE statements (and ALTER TABLE statements if you are using them). In this way, you can run the script over and over to drop all tables and re-create them, should you need to modify something. Make sure to comment this create section as well. Keep in mind that you will need to drop the tables in an order that honors the foreign key constraints.

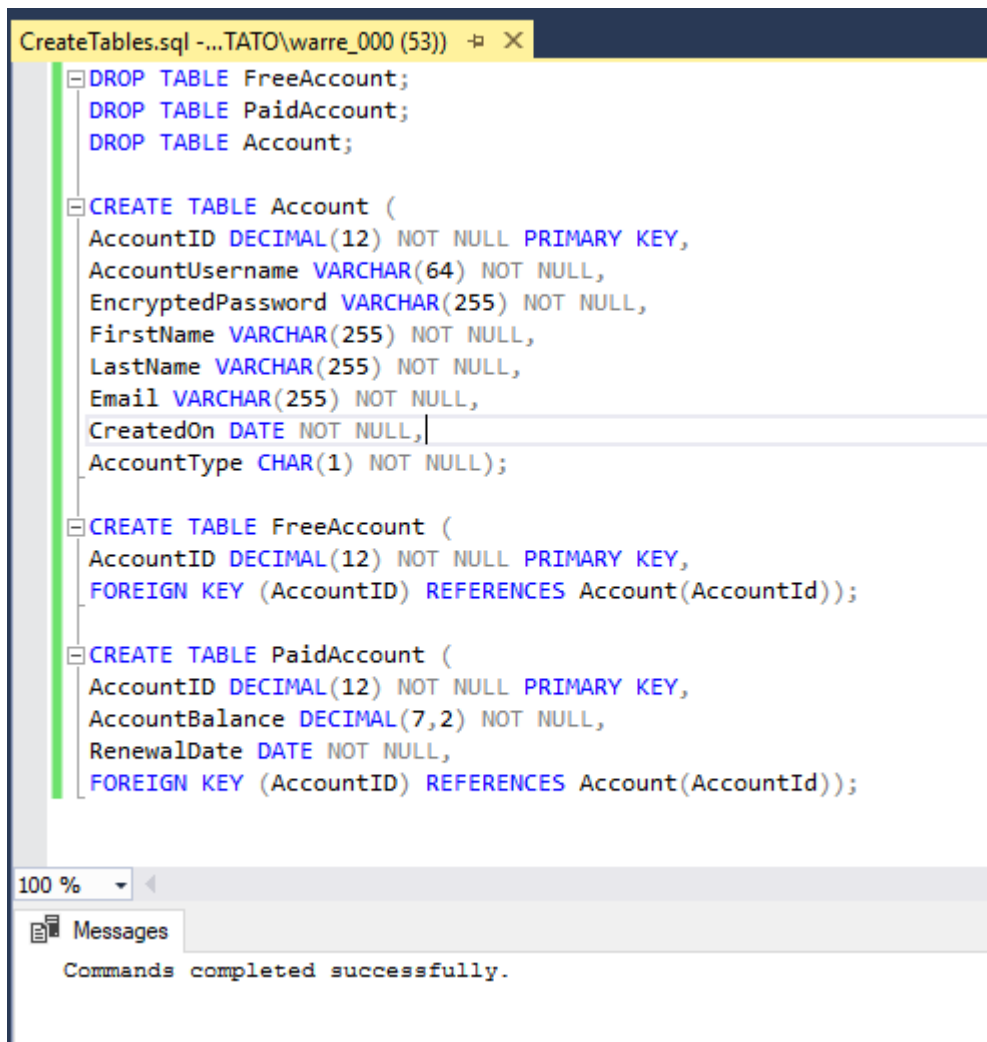
When you submit this and future iterations, attach the SQL script.

Capture a screenshot of the execution of the table creations. There's no need to capture all of the output in this instance. A single screenshot demonstrating the creations executed will suffice.

Below is an excerpt for TrackMyBuys. Note that in the excerpt, I do not exhaustively create all tables, but create enough to provide an example for you to follow. You should create all of your tables.

TrackMyBuys Create Script

In this example I create the Account table along with its subtypes. The screenshot below contains the script along with results of its execution. Note that I am using SQL Server for TrackMyBuys.



```
CreateTables.sql -...TATO\warre_000 (53) X
DROP TABLE FreeAccount;
DROP TABLE PaidAccount;
DROP TABLE Account;

CREATE TABLE Account (
  AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
  AccountUsername VARCHAR(64) NOT NULL,
  EncryptedPassword VARCHAR(255) NOT NULL,
  FirstName VARCHAR(255) NOT NULL,
  LastName VARCHAR(255) NOT NULL,
  Email VARCHAR(255) NOT NULL,
  CreatedOn DATE NOT NULL,
  AccountType CHAR(1) NOT NULL);

CREATE TABLE FreeAccount (
  AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
  FOREIGN KEY (AccountID) REFERENCES Account(AccountId));

CREATE TABLE PaidAccount (
  AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
  AccountBalance DECIMAL(7,2) NOT NULL,
  RenewalDate DATE NOT NULL,
  FOREIGN KEY (AccountID) REFERENCES Account(AccountId));

100 %
Messages
Commands completed successfully.
```

I put my DROP TABLE commands at the top so that the script is re-runnable, then followed with the CREATE TABLE commands. All columns and constraints are included as illustrated in the ERD.

Indexing Databases

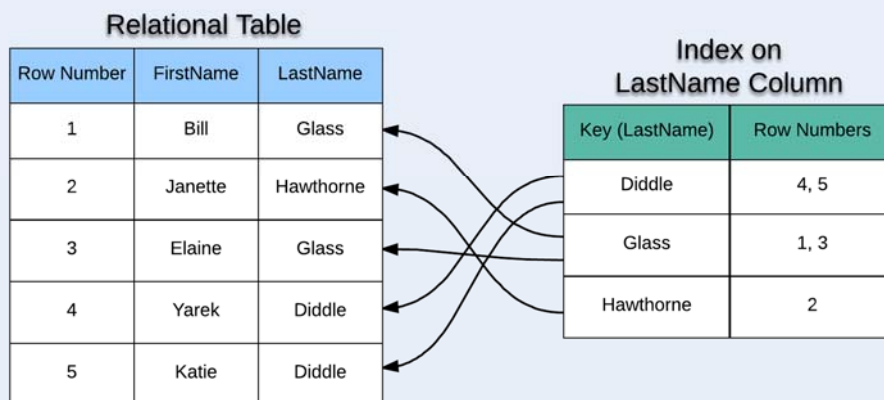
Virtually all relational databases, even well-designed ones, perform terribly without deliberate creation of indexes. This means that as a database designer and developer, you must add index creation to your list of skills. This requires a change in thinking. Logical database design deals with the structurally independent tables and relationships; you deal with these according to the soundness of their design, independent of any particular database implementation. Index placement deals with the way the database will access your data, how long that will take, and how much work the database must perform to do so. Creating indexes requires you to think in terms of implementation.

An index is a physical construct that is used to speed up data retrieval. Adding an index to a table does not add rows, columns, or relationships to the table. From a logical design perspective, indexes are invisible. Indexes are not modeled in logical entity-relationship diagrams, because indexes do not operate at the logical level of abstraction, as do tables and table columns. Nevertheless, indexes are critical component in database performance, and database designers need be skilled with index

placement, because index placement happens mostly during the design phase of a database. Excellent performance is a critical piece of database design.

Each index is assigned to a column or a set of columns in a single table, and is conceptually a lookup mechanism with two fields. The first field contains a distinct list of values that appear in the covered column or columns, and the second field contains a list of rows which have that value in the table. This is illustrated in Example 1 below.

Example 1: An Index at a Conceptual Level



In the example above, there are two data columns in the relational table – FirstName and LastName. You may have noticed the RowNumber column as well. Every relational table has an identifier for each row created automatically by the database. For simplicity in this example, the row identifier is a sequential number. There are two columns for the index. The first column contains the unique list of last names – Diddle, Glass, and Hawthorne. The second column contains the list of rows in the relational table that have the corresponding LastName value. For example, the last name “Diddle” corresponds to rows 4 and 5 in the relational table, the last name “Glass” corresponds to rows 1 and 3, and the last name “Hawthorne” corresponds to row 2. Essentially, the index is referencing the corresponding rows in the relational table.

A DBMS can oftentimes use an index to identify the rows that contain the requested value, rather than scanning the entire table to determine which rows have the value. Using an index is usually more efficient than scanning a table. Indexes are perhaps the most significant mechanism for speeding up data access in a relational database.

While there are various kinds of indexes, one kind is so ubiquitous – the B+ tree index – that oftentimes the use of the generic term “index” is actually referring to the B+ tree index. Indexes are categorized partly by their storage characteristics. B+ tree indexes are stored in a data structure known as the B+ tree, which is a data structure known by computer scientists to minimize the number of reads an application must perform from durable storage. If you’re curious on how it works technically, you will find ample descriptions on the web (it’s beyond the scope of this assignment to go into detail about B+ tree implementation). Bitmap indexes are stored as a series of bits representing the different possible data values. Function-based indexes, which are not categorized by their data storage characteristics,

support the use of functions in SQL, and are actually stored in B+ trees. B+ tree indexes are the default kind of index for many modern relational databases, and it's not uncommon for large production schemas to contain only B+ tree indexes. *Please keep in mind that the generic term "index" use throughout this assignment is referring specifically to a B+ tree index and not another kind of index.*

When learning about indexes, it's important to understand the kind being referred to.

The definition and categorization of indexes is not without complication. One prominent source of confusion is Microsoft's categorization of "clustered" versus "non-clustered" indexes, found throughout documentation for SQL Server, and in the metadata for schemas in SQL Server installations. A "clustered" index is not an index per se; rather, it is a row-ordering specification. For each table, SQL Server stores its rows on disk according to the one and only one clustered index on that table. No separate construct is created for a "clustered" index; rather, SQL Server can locate rows quickly by the row ordering. A "nonclustered" index is actually just an index by definition, a separate construct that speeds up data retrieval to a table. The terms "clustered" and "nonclustered" are not actually index categorizations, but rather provide a way to distinguish row-ordering specifications from actual indexes. An index is not a row-ordering specification.

Another source of confusion for indexes is the column store, a relatively new kind of way to store data in relational databases. For decades, relational databases only supported row-based storage; the data values for each column in a table row are stored together on disk. Indexes thus reference these rows and enable databases to locate the rows more quickly. Column stores group data values for the same column, across multiple rows, together on disk. Columns stores do not reference values in the underlying table, but actually store the values. Column stores can be beneficial when large numbers of rows are aggregated, as is the case with analytical databases. Some DBMS implement column stores with additional enhancements, such as implementing them fully in-memory or compressing the data values. Although column stores are distinct constructs, they are sometimes confused with indexes because column stores also help speed up data retrieval. In documentation for SQL Server, Microsoft refers to column stores within SQL Server as "column-store indexes", propagating that confusion. Column stores and indexes are two fundamentally distinct constructs that are implemented differently. The key distinction between the two is that indexes reference rows in relational tables, and column stores reference nothing; column stores store the data values themselves in a columnar format which can speed up analytic queries. Columns stores and indexes are two different constructs.

Some indexes have a secondary benefit beyond speeding up data retrieval. An index can be implemented to allow the key values to repeat, or can be implemented to disallow repeating keys. If the key values can repeat, it's a non-unique index; otherwise, it's a unique index. Thus, although not the primary purpose, indexes can be used to enforce uniqueness. In fact, many modern relational DBMS enforce uniqueness constraints through automatic creation of unique indexes. That is, the database designer creates uniqueness constraint (a logical construct), and the DBMS automatically creates a unique index (a physical construct), if one isn't already present for the column, in order to enforce the constraint. For this reason, DBMS automatically create indexes for primary keys; a primary key constraint is really a combination of a unique constraint and a not null constraint. Enforcing uniqueness is a secondary benefit of indexes.

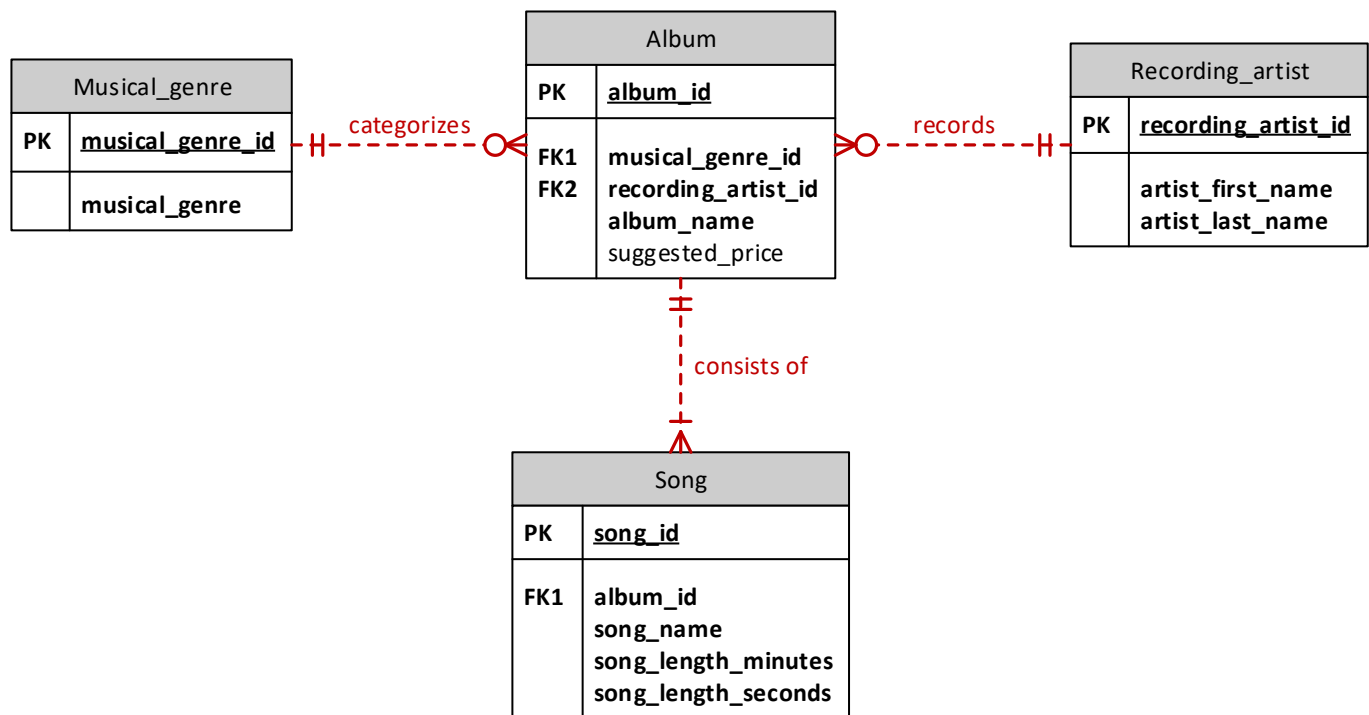
It's not necessary to create uniqueness constraints to define unique indexes; we can explicitly create unique indexes. Essentially, for every index we create, we decide whether an index is unique or non-unique. Unique indexes obtain better performance than non-unique indexes for some queries, because the DBMS query optimizer knows that each key requested from a unique index will at most have one

value, while each key requested from a non-unique index may have many values. Therefore if it is guaranteed that values will not repeat, it is better to use unique indexes. However, adding a unique index on a column that has values that can repeat will cause erroneous transaction abortions every time a repeated value is added to the column. It is important to correctly discern which type of index is needed.

Deciding Which Columns Deserve Indexes

You might reasonably ask the question, “Why not simply add indexes to every column in the schema?” After all, then we would not need to concern ourselves with index placement. The primary reason is that while indexes *speed up reading* from the database, indexes *slow down writing* to the database. Indexes associated with a table slow down writes to that table, because every time data is added to, modified, or deleted from the table, the indexes referencing the data must be modified. Another reason is that indexes increase the size of our database, and that not only affects storage requirements, but also affects database performance since the buffer cache will need to handle the increased size. Yet another reason is that indexes add complexity to database maintenance, especially during structural upgrades. If we need to delete or modify columns for an upgrade, indexes on every column would make the task more complicated. Adding indexes to every column is unnecessary and can cause several issues.

We will now work through a series of examples using the album schema defined below.



Primary Keys

For indexes, you need not consider *all* columns in a table, only most of them. Many modern relational DBMS, including Oracle and SQL Server, automatically add unique indexes to table columns covered by a primary key constraint. We do not need to add indexes to primary keys, since the DBMS will create them automatically for us.

Identifying Primary Keys

So that we know which columns would already have index on them in the album schema, we'll identify the primary key columns. We use the standardized dot notation familiar to database professionals, `TableName.ColumnName`.

Primary Key Column	Description
Musical_genre.musical_genre_id	This is the primary key of the Musical_genre table.
Album.album_id	This is the primary key of the Album table.
Recording_artist.recording_artist_id	This is the primary key of the Recording_artist table.
Song.song_id	This is the primary key of the Song table.

Notice that in this example, the indexes are implicitly unique indexes since primary key values must always be unique.

Foreign Keys

Deciding where to place indexes requires careful thought for some table columns, but there is one kind that requires no decision at all – foreign key columns. All foreign key columns should be indexed without regard to any other requirements or the SQL queries that will use them. Some DBMS, including Oracle, will sometimes escalate a row-level lock to a page-level lock when a SQL join is performed using a foreign key that has no index. The focus of this assignment is not locking, so I will not get into fine details, but suffice it to say that page-level locks are always bad for transactions because they result in deadlocks over which the database developer has no control. Another reason we index all foreign key columns is because foreign keys will almost always be used in the WHERE clauses of SQL queries that perform joins between the referencing tables and the referenced tables. The simple rule is to always index foreign key columns.

Let us look at an example of indexing foreign keys.

Adding Foreign Key Indexes

In this example, we identify all foreign key columns in the album schema. Unlike primary keys, foreign keys are not always unique, so we need to also indicate whether a unique or non-unique index is required. Below is a listing of the foreign key column indexes.

Foreign Key Column	Description
Album.musical_genre_id	This foreign key in the Album table references the Musical_genre table. The index is non-unique since many albums can be in the same genre.
Album.recording_artist_id	This foreign key in the Album table references the Recording_artist table. The index is non-unique since a recording artist can produce many albums.
Song.album_id	This foreign key in the Song table references the Album table. This index is non-unique since there are many songs in an album.

You may have noticed that all of the foreign key indexes in Example 3 are non-unique. In practice, most indexes are non-unique because most columns are not candidate keys.

Query Driven Index Placement

Columns that are considered neither primary nor foreign key columns must be evaluated on a case-by-case basis according to more complex criteria. It starts with a simple rule: *every column that will be referenced in the WHERE clause or any join condition of any query is usually indexed*. The WHERE clause and join conditions in a SQL query contain conditions that specify what rows from the tables will be present in the result set. The query optimizer makes heavy use of these conditions to ensure that the results are retrieved in a timely fashion. For example, if the underlying table has a billion rows, but a condition in the WHERE clause restricts the result set to five rows, a good plan from the query optimizer will only access a small number of rows in the underlying table as opposed to the full billion rows, even if many other tables are joined in the query. We intelligently select columns to index based upon how they are expected to be used in queries, and how we expect the database will execute those queries.

You probably noticed the word “usually” in the rule described above, hinting that the rule is not so simple after all. While we can safely exclude columns that are *not* used in WHERE clauses or join conditions, indexing columns that *are* used in those constructs is usually but not always useful. There are other factors to consider. A simple criterion drives indexing decisions: *add an index to a column if it will speed up SQL queries that use it more than a negligible amount*. If it does not meet this criterion, adding an index is not useful, and would be slightly detrimental since it increases the database size and slightly slows down writes to the table. Essentially, we need to discern whether or not the database will make use of the index. If we create the index and the database does not use it, or if the database does use it but doing so does not increase performance, the index is not beneficial. Simple truth gives way to complexity in regards to indexing.

Table size isn’t a significant factor in logical database design, but it is certainly a significant factor for index placement. Databases usually do not use indexes for small tables. If a table is small enough to fit on just a few blocks, typically no more than a few hundred rows or few thousand rows depending upon the configuration, the database often determines that it’s more efficient to read the entire table into memory than to use an index on that table. After all, using an index requires that database to access one or more blocks used by the index, then additionally access the blocks needed for the table rows. This may be less efficient than reading the entire table into memory and scanning it in memory. Small lookup tables or tables that will never grow beyond a few hundred or a few thousand rows may not need an index. We must consider a table’s size before adding indexes to it.

Large tables do not always benefit from indexes. Another factor databases use to determine whether or not to use an index is the percentage of rows pulled back by queries that access it. Even when a table is quite large, if the queries that access it read in most rows of the table, the database may decide to read in the entire table into memory, ignoring indexes. If most rows of the table will be retrieved, it’s often more efficient for the database to read the entire table into memory and scan it than to access all the blocks for an index and read in most rows of the table. Typically, day-to-day business systems will access a small subset of rows in queries, and analytical systems pull back large subsets of the table to aggregate results (these rules will of course sometimes be broken). So we consider the type of system, and what that system will do with the particular table, to decide whether adding an index is beneficial.

If the number of distinct values in the column is very small, it's usually not beneficial to index the column, even on large tables, even when queries access a small number of rows. For example, imagine a "gender" column with two possible values on a billion-row table. If a single row was being requested, what good would it do the database to narrow down the search to 500,000 rows using the index? Not to mention, an index with a single entry that references 500,000 rows would not be efficiently accessed. An index is beneficial if it can be used to narrow down a search to a small number of rows (relative to the size of the table). If an index only cuts the table in half, or into a few partitions, it's not beneficial. Indexes must subdivide the table into enough partitions to be useful.

If most values for a column are null, it's not usually beneficial to index the column. The reason for this is actually the same as in the prior paragraph. As far as the database is concerned, a large number of null values is just another large partition identified by the index. For example, if a column for a billion-row table has 900,000 nulls, then the database could use the index, but would only narrow down the search to 900,000 rows whenever the value is null; this is not useful! An additional complication is that some DBMS do not add null entries to indexes, and some do, so some DBMS cannot take advantage of the index when a value is null regardless of the number of nulls. The percentage of null values should be taken into account when deciding which columns deserve an index.

In practice, one does not usually peruse the thousands of queries written against a schema to determine which columns to index. We need to decide what to index when a system is being created, before all the queries are written. We can usually spot the columns that are likely to be used in the where clause or in join conditions and provide indexes for those without viewing queries. For example, many systems would support searching on a customer's name, but would not support searching on optional information such as order comments that customers may for an order. It would be reasonable for us to index the first and last name columns, and to avoid indexing an order_comment field. Of course, you need to know the system and how queries will generally use the database fields in order to make this determination. A systematic review of all of a system's queries is not required to place many of the indexes in a database schema.

This is not to say that every index for a database schema is created at design time. Sometimes during development and maintenance of a system, a new query is written that does not perform well, and we must add an index to support the query. We must understand how to correctly place indexes given a specific query. The process of adding indexes can be categorized as adding most indexes upon design of the database, and adding some indexes iteratively over time as new queries demand them. Adding indexes is an iterative process.

You may have discerned that although certain principles are in play, index placement is not always an exact science. I will reiterate the guiding principle once again: *add an index to a column if it will speed up SQL queries that use it more than a negligible amount*. Confirming this may require some before and after testing for some indexes, or researching the system that will use it. If you apply this principle to every index you create, you will be creating useful indexes throughout your database, and improving database performance.

Let us now work through examples where specific queries are observed and indexes are created for those queries.

Query by Song Name

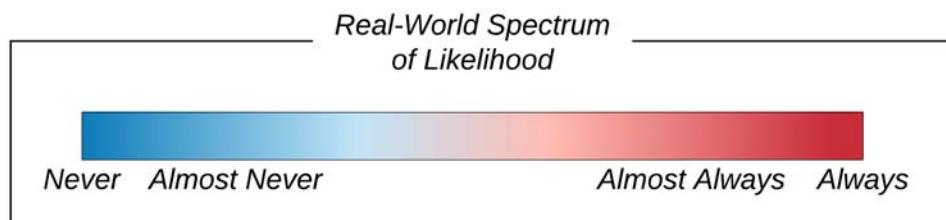
In this example, there is single table query that retrieves the minutes and seconds for the “Moods for Moderns” song in the album schema.

```
SELECT Song.song_length_minutes, Song.song_length_seconds
FROM   Song
WHERE  Song.song_name = 'Moods For Moderns'
```

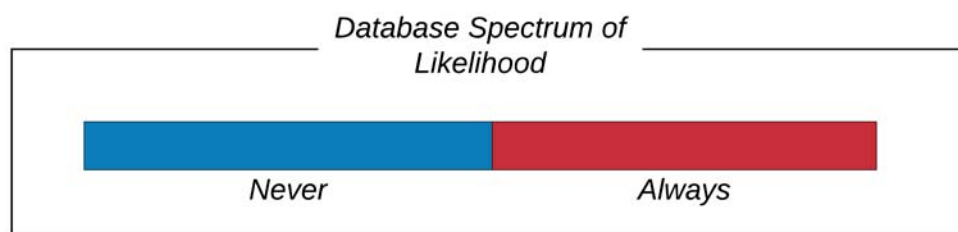
Three columns are accessed in this query – *song_length_minutes*, *song_length_seconds*, and *song_name* – but only *song_name* is in the WHERE clause. Therefore we would index the *song_name* column. The database can use *song_name* to locate the correct rows in the table, and once the rows are located, can retrieve additional columns including *song_length_minutes* and *song_length_seconds* without searching again. Adding indexes for the other two columns would not benefit this query.

We would create this index as a non-unique index, because it is possible that a song name can repeat.

Because song names are almost always unique (artists usually give each new song a unique title), one might lean toward creating a unique index in the above example. In the real-world “almost always” is considered quite close to “always” in spectrum of likelihood of occurrence, demonstrated in the figure below.



However in the database world the two are fundamentally different, because we are dealing with absolutes. This is illustrated in the figure below.



If something can happen even once, then we must accommodate for it. For example, several artists have sung the song “Amazing Grace”, so if we add a unique index to *song_name*, we would prevent these songs from being input into our database. It is important not to apply shades of gray to the choice of indexes, because a unique index requires 100% conformity.

Now for another example.

Query by Album Name

In this example, the query retrieves the artist name for the “Power Play” album.

```
SELECT Recording_artist.artist_first_name, Recording_artist.artist_last_name
FROM   Album
JOIN   Recording_artist
ON     Album.recording_artist_id = Recording_artist.recording_artist_id
WHERE  Album.album_name = 'Power Play'
```

Album.recording_artist_id and *Recording_artist.recording_artist_id* are both used in a join condition so are candidates for indexing. However, the former is a foreign key, and the latter is a primary key; we already marked these columns for indexing in Example 1 and Example 2. This also demonstrates that foreign keys are typically used in join conditions.

Album.album_name appears in the WHERE clause (this how the query limits the results to the “Power Play” album), so we will want to index it. Similar to song names, album names are usually unique, but not always, so we make the index non-unique.

Queries with subqueries can be complex, but subqueries affect indexing in a predictable way. The WHERE clause and join conditions for the outer query, and for all subqueries, collectively determine what rows from the underlying tables in the schema will be retrieved. Hence all columns in the WHERE clause and join conditions for the subqueries are candidates for indexing, in addition to those in the outer query. In terms of indexing, subqueries add more layers, but do not change the strategy.

Now for a more complex example of a query that contains a subquery.

Query for Albums by Song Length

This query lists the names of all albums that have songs that are less than four minutes in length.

```
SELECT Album.album_name
FROM   Album
WHERE  Album.album_id
      IN (SELECT Song.album_id
          FROM   Song
          WHERE  Song.song_length_minutes < 4)
```

Because *song_length_minutes* is in the WHERE clause of the subquery, we add an index to that column. We use the same strategy for the subquery as with the outer query. *Album.album_id* is the primary key of album, and so was previously indexed in Example 1.

Even though the less-than sign “<” is used in Example 6 rather than equality, the database can still take advantage of an index on the column. The reason is that the index is sorted, and the database can take advantage of the sorting to efficiently locate all songs less than 4 minutes in length.

Indexing Summary

You learned many principles as to how we determine which columns deserve indexes. To help you remember them, let's summarize them now.

- ✓ Modern databases automatically index primary key columns.
- ✓ Foreign key columns should always be indexed.
- ✓ Columns in WHERE clauses or join conditions are usually indexed.
- ✓ Indexes on tables that remain small are usually not beneficial.
- ✓ If queries always retrieve most rows in a table, indexes on that table are not usually beneficial.
- ✓ Avoid indexing columns with a small number of distinct values or a large percentage of nulls.

Identifying Columns Needing Indexes for your Database

You now know enough to identify columns needing indexes for your database. As noted in the prior section, there can be many indexes discovered both at design time and implementation time. So, we'll limit the amount you need to identify to the below:

1. Identify all primary keys so you know which columns are already indexed.
2. Identify all foreign keys so you know which columns must have an index without further analysis.
3. Identify three query driven indexes, that is, indexes that are neither foreign keys nor primary keys, but deserve an index because of queries that will use them.

Use the standard TableName.ColumnName format when identifying columns for indexing. For example, if a table is named "Person" and a column for indexing is named "LastName", then it would be identified as "Person.LastName". For each index, explain why you selected it, and indicate whether it's a unique or non-unique index, and why.

Below are the columns I identified for TrackMyBuys.

TrackMyBuys Indexing

As far as primary keys which are already indexed, here is the list.

Account.AccountId
FreeAccount.AccountId
PaidAccount.AccountId
Purchase.PurchaseId
OnlinePurchase.PurchaseId
FaceToFacePurchase.PurchaseId
Address.AddressId
State.StateId
Store.StoreId

Product.ProductId
ProductPurchaseLink.ProductPurchaseLinkId

As far as foreign keys, I know all of them need an index. Below is a table identifying each foreign key column, whether or not the index should be unique or not, and why.

Column	Unique?	Description
Purchase.StoreID	Not unique	The foreign key in Purchase referencing Store is not unique because there can be many purchases in the same store.
Purchase.AccountID	Not unique	The foreign key in Purchase referencing Account is not unique because there can be many purchases from the same account.
FaceToFacePurchase.AddressID	Not unique	The foreign key in FaceToFacePurchase referencing Address is not unique because there can be many purchases at the same store location.
Address.StateID	Not unique	The foreign key in Address referencing State is not unique because there can be many addresses in the same state.
ProductPurchaseLink.ProductID	Not unique	The foreign key in ProductPurchaseLink referencing Product is not unique because the same product can be purchased many times.
ProductPurchaseLink.PurchaseID	Not unique	The foreign key in ProductPurchaseLink referencing Purchase is not unique because a single purchase can be linked to many products.

As far as the three query driven indexes, I spotted three fairly easily by predicting what columns will commonly be queried. For example, it's reasonable that there will be many queries that limit by account balances, to see what accounts are over or under a certain. So I select PaidAccount.AccountBalance to be indexed. This would be a non-unique index because many accounts could have the same balance.

It's also reasonable that the date of purchase will be a limiting column in queries, because reports and analysts will commonly want to limit their analysis by date range, such as a particular year, quarter, month, or day. So I select Purchase.PurchaseDate to index. This would be a non-unique index because many purchases can happen on the same day.

Lastly, it's reasonable that the date the account was created will be a limiting column for some queries, such as queries that want to see how many accounts were created in a certain time period. So I select Account.CreatedOn for an index, which would be non-unique index because many accounts could be created on the same day.

Creating Indexes in SQL

Identifying the columns that deserve indexes is an important step, but alone does not improve performance on a database. Of course, the indexes need to be created in SQL for the database to take advantage of them. Thankfully, the same command can be used across Oracle, SQL Server, and Postgres. The command to create a non-unique index (which are most common) is illustrated below.

Creating a Non-Unique Index

```
CREATE INDEX IndexName  
ON TableName (ColumnList);
```

The keywords CREATE INDEX are followed by a name given to an index. It's a good idea to choose a useful name that will help explain what the index is for, which might include part or all of the tablename, part or all the column name, and perhaps even a suffix such as "idx" to indicate it's an index. For example, if an index was placed on the FirstName column in a Person table, the index could be named "PersonFirstNameIdx". The reason we care about the name is that we sometimes see them come up in error messages, and a useful name will help us track down any issues without the need to investigate the schema's metadata.

Next, the ON keyword is followed by the name of the table which will contain the index. Last, a comma-separated list of columns in the table are listed. It is quite common for an index to contain only one column. It is possible, however, to have an index contain multiple columns. Such composite indexes are useful if queries commonly access all of those columns together.

Creating a unique index is very similar, as illustrated below.

Creating a Unique Index

```
CREATE UNIQUE INDEX IndexName  
ON TableName (ColumnList);
```

The syntax is mostly identical to the non-unique version, the exception being the presence of the UNIQUE keyword. When a unique index is created, the database enforces that the column (or columns) in the list have unique values. If a SQL statement makes a change that would violate the uniqueness, the SQL statement is rejected.

Let's take a look again at the Car example that was in a prior section. As a reminder, the SQL that creates the Car table is listed below.

Car CREATE TABLE Statement

```
CREATE TABLE Car (  
  CarID    DECIMAL(12) NOT NULL PRIMARY KEY,  
  VIN      VARCHAR(17) NOT NULL,  
  Price    DECIMAL(8,2) NOT NULL,  
  Color    VARCHAR(64) NOT NULL,  
  Make     VARCHAR(64) NOT NULL,  
  Model    VARCHAR(64) NOT NULL,  
  Mileage  DECIMAL(7) NOT NULL);
```

We can explore adding both a unique and a non-unique index for this table. Let's start with a non-unique index. In truth, all of these attributes – Price, Color, Make, Model, and Mileage – deserve a non-unique index because the car reseller could be interested in asking questions about any one of these fields. How many cars are in stock that are greater than a certain price? How many cars are in stock with a particular color? How many cars in stock have a specific make and model? How many cars in stock have fewer than a specific mileage? These are all reasonable questions for a car reseller.

For illustrative purposes, I will add an index for Price, illustrated below.

Car Price Index Creation

```
CREATE INDEX CarPriceIdx  
ON Car(Price);
```

Notice that I created an index named “CarPriceIdx” to help us know later what it would be used for. Further notice that the index identifies the Price column in the Car table.

The VIN column deserves a unique index, for two reasons. First, every car must have a unique VIN. If any person tries to enter two different cars with the same VIN, it should not be permitted at least at the database level, and even better at the application level. Second, it is quite reasonable that a person would want to lookup a car by its VIN number, and establishing an index on the column will support an efficient lookup. We can create a unique index to accomplish both of these goals. Such an index is illustrated below.

Car VIN Index Creation

```
CREATE UNIQUE INDEX CarVINIdx  
ON Car(VIN);
```

Notice the UNIQUE keyword here makes the index unique. The name “CarVINIdx” helps describe what the index is for. Otherwise this index creation is quite similar to the creation of the Price index.

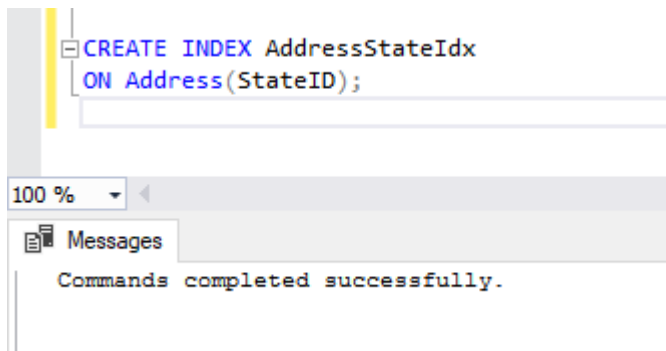
Creating Indexes in your Database

Go ahead and create the indexes for your database that you identified. You'll want to add these commands to your SQL script after your table creations. Comment the section so your instructor or facilitator knows what the SQL code is for. Provide a screenshot of creating the indexes.

Below are a sample of the index creations for TrackMyBuys. Note that only one foreign key index and one query-driven index is demonstrated for TrackMyBuys, but you should include all that are requested in the instructions.

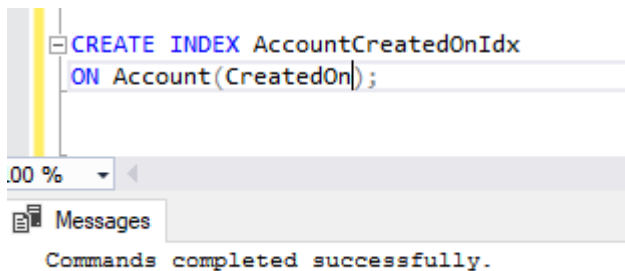
TrackMyBuys Index Creations

Here is a screenshot demonstrating creation of a foreign key index for TrackMyBuys, for the foreign key between Address and State.



I named the index “AddressStateIdx” to help identify what it’s for, and placed the non-unique index on the StateID table in Address.

Here is a screenshot demonstrating creation of a query-driven index, an index on the CreatedOn attribute in Account.



I named the index “AccountCreatedOnIdx” to help identify what it’s for, and put it on the CreatedOn column in Account.

Summary and Reflection

You have created all of your tables based off of a normalized DBMS physical ERD, and indexed your tables. Your database structure is now implemented in SQL, and your tables are primed to be filled with data. Great work! Just in the next iteration, you will be inserting data into these tables and writing useful queries against the data. Update your project summary to reflect your new work.

It’s a common phenomenon that, once you start implementing your design in SQL, both the structures that work well as well as the structures that need improvement become acutely obvious. Write down your questions, concerns, or observations you have about this, and other areas, so that you and your facilitator or instructor are aware of them.

Here is an updated summary as well as some observations I have about my progress on TrackMyBuys for this iteration.

TrackMyBuys Summary and Reflection

My database is for a mobile app named TrackMyBuys which records purchases made across all stores, making it the one stop for any purchase history. Typically, when a person purchases

something, they can only see their purchase history with that same vendor, and TrackMyBuys seeks to provide a single interface for all purchases. The database must support a person entering, searching, and even analyzing their purchases across all stores.

The structural database rules and conceptual ERD for my database design contain the important entities of Store, Product, Purchase, and Account, as well as relationships between them. The design contains a hierarchy of Purchase/FaceToFacePurchase and Purchase/OnlinePurchase to reflect the two primary ways people purchase products. The design also contains a hierarchy of Account/PaidAccount and Account/FreeAccount to reflect the fact that people can signup for a free account or a paid account for TrackMyBuys. The DBMS physical ERD contains the same entities and relationships, uses the best practice of synthetic keys, and contains the important attributes needed by the database to support the application.

The SQL script that creates all tables follows the specification from the DBMS physical ERD exactly. Important indexes have been created to help speed up access to my database and are also available in an index script.

It's thrilling to create the tables and columns my database needs. My design is no longer abstract, but actually implemented in a real, modern database. I can't wait to add data to it and query it!

My database structure seems to map OK to SQL. I suppose I will know for sure next iteration, once I start putting data into the database.

Items to Submit

In summary, for this iteration, you add attributes to your DBMS physical ERD, normalize it, create your tables and constraints in SQL, and add indexes. Your design document will contain the following items, with items new or partially new to this iteration highlighted.

Component	Description
Project Direction Overview	Revise (if necessary) your overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it.
Use Cases and Fields	Revise (if necessary) your use cases that enumerate steps of how the database will be typically used, and the significant database fields needed to support the use case.
Structural Database Rules	Revise (if necessary) your list of structural database rules for all significant entities and relationships.
Conceptual entity-relationship diagram	Revise (if necessary) your conceptual ERD.
Full DBMS Physical ERD	Revise (if necessary) your initial DBMS physical ERD, add attributes to it, and normalize it.
Index Identification and Creations	Identify indexes useful to your database, explain why they help, and provide screenshots of their creations.
Summary and Reflection	Revise the concise summary of your project and the work you have completed thus far, and your questions, concerns, and observations.

Your script will contain the following sections.

Script	Description
Table Creations	Add a section that creates all of your tables and constraints.
Indexes	Add a section that creates your indexes.

Evaluation

Your iteration will be reviewed by your facilitator or instructor with the following criteria and grade breakdown.

Criterion	A	B	C	D	F	Letter Grade
Technical mastery (50%)	Evidence of excellent mastery throughout	Evidence of good mastery throughout	Evidence of basic mastery throughout or good mastery intermittently	Minimal mastery evidenced	Virtually no mastery evidenced	
Depth and thoroughness of coverage (25%)	Excellent depth and coverage of significant topics and issues	Good depth and coverage of significant topics and issues	Basic depth and coverage of significant topics and issues	Minimal depth and coverage of significant topics and issues	Virtually no depth and coverage of significant topics and issues	
Clarity in presentation (25%)	Ideas and designs are exceptionally clear and organized throughout	Ideas and designs are clear and organized throughout	Ideas and designs are somewhat clear and organized throughout	Ideas and designs are mostly obscure and disorganized	Ideas and designs are entirely obscure and disorganized	
					Assignment Grade:	#N/A
The resulting grade is calculated as a weighted average as listed using A+=100, A=96, A-=92, B+=88, B=85, B-=82 etc.						
To obtain an A grade for the course, your weighted average should be >=95, A- >=90, B+ >=87, B >= 82, B- >= 80 etc.						

Use the **Ask the Facilitators Discussion Forum** if you have any questions regarding how to approach this iteration. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.