Homework Problem Set 3: Introduction to SQL

# Overview

In this lab, we will explore how implement a relational model in SQL.

## Learning Objectives

Upon completion of the lab, you should be able to:

* Use SQL DDL (data-definition language) commands for metadata management activities.
* Use SQL DML (data-manipulation language) commands to perform data management activities.
* Create a single SQL script to execute the creation of the database objects and place the data in an initial state.
* Apply relational concepts and best practices learned in previous units using the SQL language.
* Troubleshoot data integrity constraints.
* Demonstrate how to create tables, keys, and constraints.

## What You Will Ned

To complete this lab, you will need the learn-databases environment up and running, specifically:

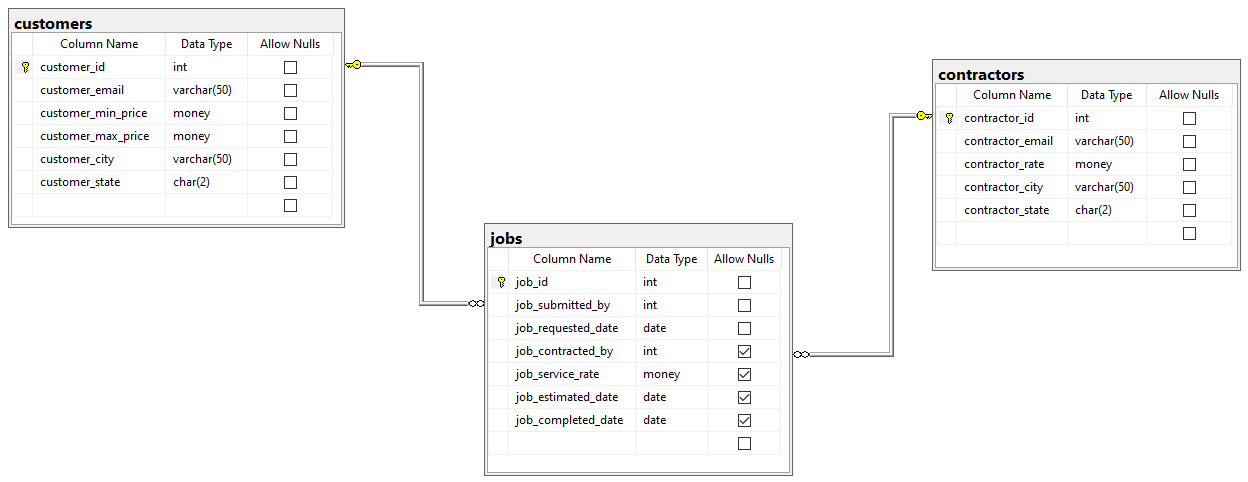
* Microsoft SQL Server DBMS.
* Azure Data Studio connected to SQL Server with an open query window in the **master** database.
* Please review the first lab if you require assistance with these tools.

## The Database: Moze.com

Moze.com connects people who want their lawn grass cut (customers) with those who are willing to cut it (contractors). Here is the business model:

1. Customers submit a lawn-cutting job for a specific date and time.
2. Contractors with billing rates within the customer’s minimum/maximum threshold and within the same city are matched and can choose to accept the job.
3. The customer is notified when a contractor accepts the job and is provided with a date and time of the service.
4. Once the job is done, the customer can rate the contractor with one to five stars.

### Moze: Logical Model

Here is the logical model of Moze.com. You will need to refer to this information throughout the lab.  


*Figure 1. Internal model of moze.com*

Table: **customers**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Domain** | **Comments** |
| customer\_id | Surrogate key, not null | Will be the primary key |
| customer\_email | Varchar length 50, not null | Natural key |
| customer\_min\_price | Money, not null | Smallest price the customer will pay for a lawn cutting |
| customer\_max\_price | Money, not null | Maximum price the customer is willing to pay for a lawn cutting |
| customer\_city | Varchar 50, not null | U.S. city of the customer |
| customer\_state | Char 2, not null | U.S. state code of the customer (lookup table) |
|  | | |
| **Constraint Name** | **Type** | **Comments** |
| pk\_customers\_customer\_email | Primary key on customer\_id | Enforces PK over surrogate key on table |
| u\_customers\_customer\_email | Unique on customer\_email | Enforces natural key to establish entity integrity |
| ck\_customers\_valid\_prices | Check customer\_min\_price <=customer\_max\_price | Make sure the minimum price is below or equal to the maximum price |
| fk\_customers\_customer\_state | Foreign key references state\_lookup table | A lookup table to restrict customer state to valid U.S. states only |

Table: **contractor**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Domain** | **Comments** |
| contractor\_id | Surrogate key, not null | Will be the primary key |
| contractor\_email | Varchar length 50, not null | Natural key |
| contractor\_rate | Money, not null | The amount the contractor charges for the job |
| contractor\_city | Varchar 50, not null | U.S. city of the contractor |
| contractor\_state | Char 2, not null | U.S. state code of the contractor (lookup table) |
|  | | |
| **Constraint Name** | **Type** | **Comments** |
| pk\_contractors\_contractor\_email | Primary key on contractor\_id | Enforces PK over surrogate key on table |
| u\_contractors\_contractor\_email | Unique on contractor\_email | Enforces natural key to establish entity integrity |
| fk\_contractor \_contractor\_state | Foreign key references state\_lookup table | A lookup table to restrict contractor state to valid U.S. states only |

Table: **jobs**  
The jobs table connects customer to contractor.

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Domain** | **Comments** |
| job\_id | Surrogate key, not null | Will be the primary key |
| job\_submitted\_by | Int, not null | The customer ID of the customer who submitted the job |
| job\_requested\_date | Date, not null | When the job should be done |
| job\_contracted\_by | Int, null | The contractor ID of the contractor who accepted the job |
| job\_service\_rate | Money, null | The amount of the job |
| job\_estimated\_date | Date, null | The estimated date of when the job will be done |
| job completed\_date | Date, null | The actual date the job was done |
| job\_customer\_rating | Int null | An integer value between 1 and 5 |
|  | | |
| **Constraint Name** | **Type** | **Comments** |
| pk\_jobs\_job\_id | Primary key on job\_id | Enforces PK over surrogate key on table |
| fk\_jobs\_job\_submitted\_by | Foreign key, references customers table PK | Customer who submitted the job |
| fk\_jobs\_contracted\_by | Foreign key, references contractors table PK | Contractor who accepted the job |
| ck\_valid\_job\_dates | Check requested\_date=<=estimated\_date and estimated\_date<=completed\_date | Makes sure the dates are valid |

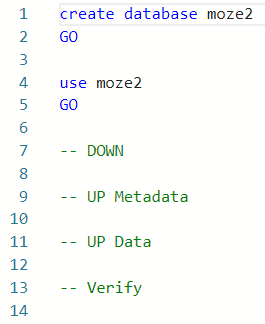
# Walkthrough

In this walkthrough we will begin the process of creating a single SQL data migration script (an up/down script) to create the Moze schema and data. The name of this script will be **moze-up-down.sql**.

Open a new query window, and save the empty query under this name.

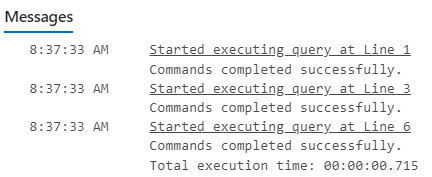
## Creating the Database

Let’s begin by creating the database and switching the query to that database. In your query window, type the following and then run the **moze-up-down.sql**  script. This is the basic template for an up/down script.



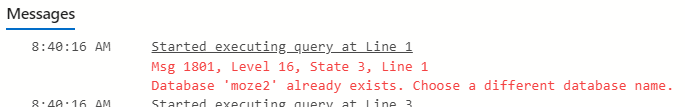
Notice how we must batch this script. You cannot create a database and use it in the same SQL command, so we must divide the script into two batches. The first batch creates the database, and the second, which executes after the first is complete, switches to the moze2 database.

After you run this script you should see messages showing success.



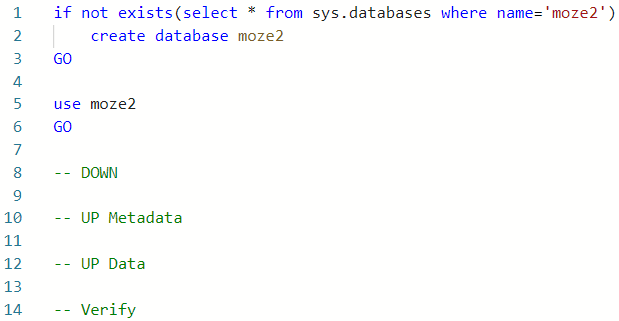
Make sure to save your script file!!! Save it often—you will not be reminded to do so!

## Creating the Database… Just One Time!

If you execute this script again, you will see this error:  


The database management system cannot create the same database more than once. You might think in the spirit of the up/down script we should reverse this operation and perform a **drop database** command, but dropping a database is quite costly to the system and should not be used in this manner. We are better off performing a soft create, meaning if the database exists, do not create it again. For this we use SQL Server’s **sys.databases** internal table. You should note every DBMS does this differently.

Change the script to look like this, and then execute it to ensure there are no errors:



## Understanding the Up/Down Philosophy: Making the Stateful Stateless

Database management systems are persistent stores of data and metadata. Unlike other programming languages, the SQL commands are not stateless, meaning the execution of a command has a permanent effect on the system. This can make it extremely difficult to track changes to a DBMS and replay those changes on another DBMS, which is essential to database development.

We turn stateful into stateless using the up/down philosophy. Every time we use a DDL command in the ---UP section, we write the opposite “undo” DDL command in the --DOWN section so that the script will run and recreate the schema each time. Running only the --DOWN portion of the script will revert the database back to its state prior to the changes.

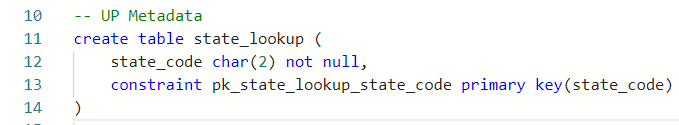
Furthermore, we can take a working up/down and execute it against another DBMS as a means of placing that database in the same state as our development database. Very valuable, indeed!

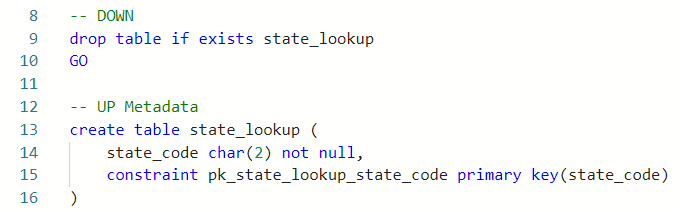
Although this is not required in production environments, it is crucial for development environments where we must provide a consistent and reproducible set of tables and data.

## Adding the **state\_lookup** Table

Let’s start simple with the **state\_lookup** table. This table will hold the U.S. states where Moze.com conducts business.

Add the following SQL to the --UP metadata section to create the table:



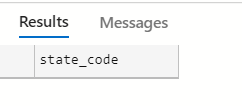
What goes up must come down. We add the **drop table** statement to the --DOWN section:  


Notice we added the GO batch operator between the drop and the create. This makes sense because we should not perform these operations within the same query. We need to make sure the database has dropped the table before we actually attempt to recreate it.

As a final step, let’s add a **select** statement to --Verify so we can see the table.

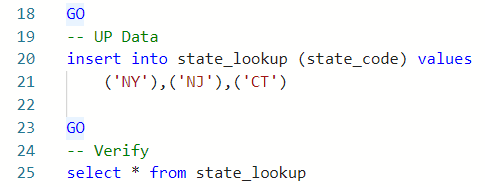
A screenshot of SQL code

Now save and execute the script. You should see this output if everything worked:

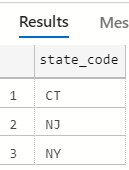


*Figure 2. The empty table state\_code*

Let’s finish up the **state\_lookup** table by inserting some data. In the --UP Data section, add:

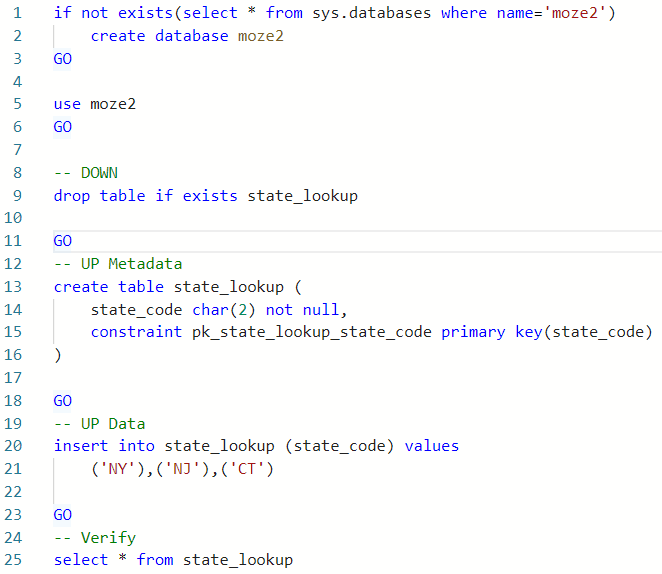


Execute the script, and you will see this output:



*Figure 3. moze.com only does business in NY, NJ, and CT.*

Here is entire up/down script at this point. Notice how there are GO batch operators separating each section to ensure the changes to the DBMS are committed before executing the next step.

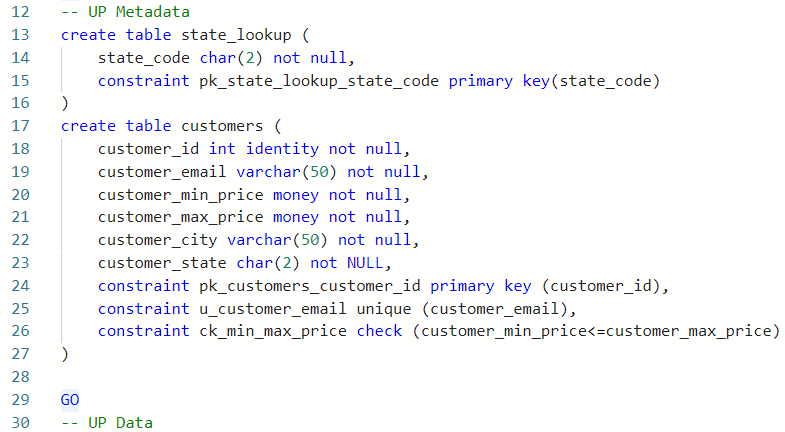


*Figure 4. Entire script at this point*

## Adding the customers Table

In this next and final step of the walkthrough, let’s add the **customers** table, a foreign key constraint to the lookup table, and insert data.

Let’s add the **create table customers** statement between the **create table state\_lookup** and the **GO** like this:



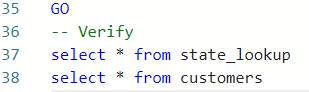
We created this table based on the information found in the **Moze: Logical Model** section at the top of this document. Except for the foreign key, all constraints were added at the time the table was created. It is good practice to add the foreign key constraints separately. But before we add it, we should reverse the operation in the down script section.

Notice how we drop the **customers** table before the **state\_lookup** table. We must complete the DOWN operation in the opposite order from the UP operation to avoid any dependency issues—LIFO (last in, first out) or in this case, I suppose, it’s last DOWN is first UP (LDFU).

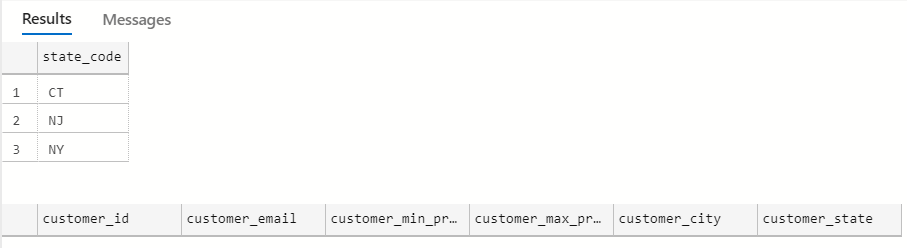
A screenshot of SQL code, highlighting the reversing of the DOWN statements


*Figure 5. The DOWN statements should execute in the opposite order as the UP statements, last in, first out.*

Finally let’s add the customers table to the –Verify section and then run the entire script.

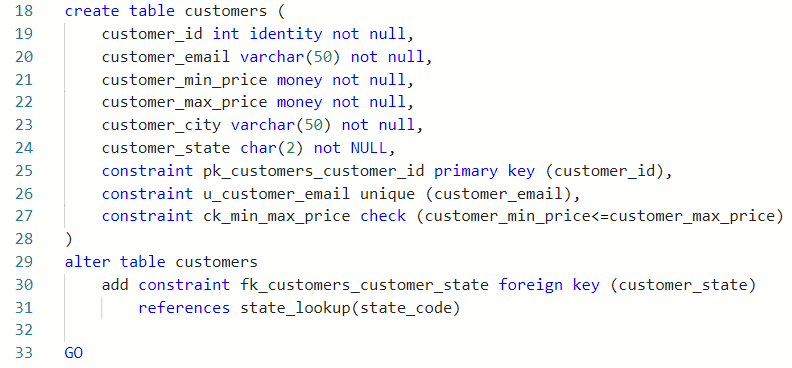


When you execute the script, you should see this output:

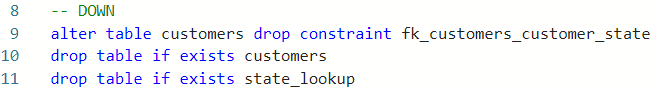


## Adding the Foreign Key

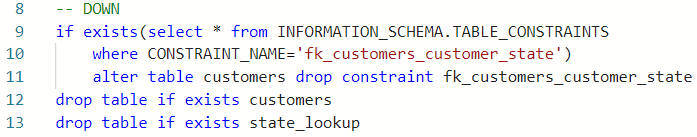
Now that we have the **state\_lookup** and **customers** tables, we can add a foreign key over the **customer\_state** column referencing the lookup table. We add this below the **create table customers** statement in the –UP metadata section:



Keeping with our last down, first up approach, we add theSQL to drop the foreign key first in the --DOWN section:



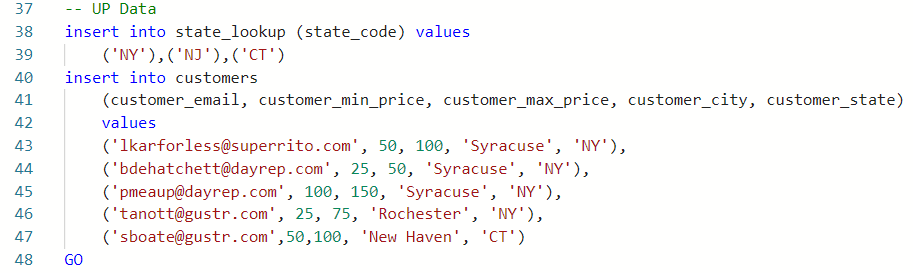
Because we want all of our drops to be soft (drop only if exists), we need to use **INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS** to find the foreign key before we drop it:



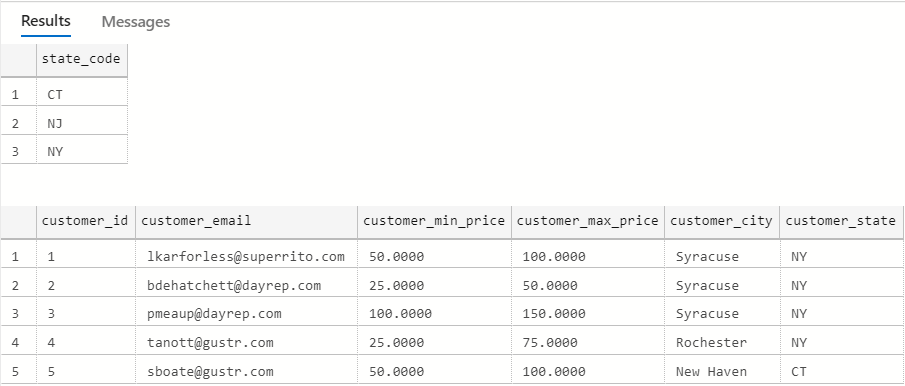
Run the script, and make sure it works as expected.

## Adding Data

With the **customers** table built and the foreign key set, we can now add some customers. Add these insert statements below the **state\_lookup**:



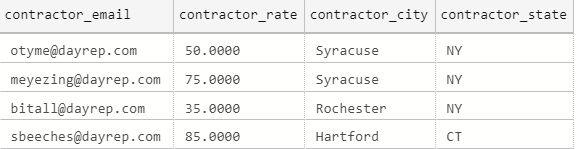
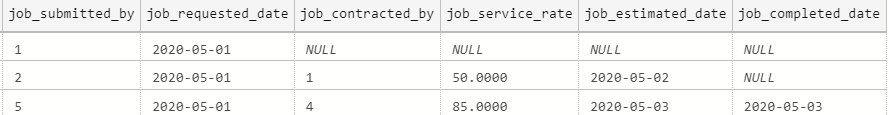
And run the entire script to verify it works.



Don’t forget to save your work often!

# Questions

Answer these questions using the problem set submission template. You will need to consult the logical model in the overview section for details. For any screen shots provided, please follow the guidelines for submitting a screen shot.

1. Add the **contractors** table as defined in the overview section to your SQL script at the bottom of your --UP metadata section. Include columns, indexes (PK/unique) in the create table statement. Provide a screen shot of the SQL code.
2. Add the reverse command to the --DOWN section of your SQL script, dropping the table. Provide a screen shot of the code.
3. Alter the **contractors** table adding a foreign key over the **contractor\_state** column, **fk\_contractors\_contractor\_state.** Add it to the --UP metadata portion of the script. Provide a screen shot of the SQL code.
4. Add the reverse command to the DOWN section of your SQL script, dropping the foreign key. It should be a soft delete as with the other foreign key in the walkthrough. Provide a screen shot of the code.
5. At the bottom of the --UP data section, insert the following contractor data:  
    Add a select statement to the –Verify section. Provide evidence your script works to this point by including a screen shot of the table outputs.
6. Create the **jobs table** with PK and check constraints. Add it to the appropriate section of the script, and provide a screen shot of the SQL code.
7. Add the drop table statement for the **jobs table**, add it to the appropriate section of the script, and provide a screen shot of the SQL code.
8. Add the two foreign key constraints to the **jobs table.** Add them to the appropriate section of the script, and provide a screen shot of the SQL code.
9. Add code to softly remove the foreign key constraints from the **jobs** table (should be two separate checks for drops). Add it to the appropriate section of the script, and provide a screen shot of the SQL code.
10. Write SQL code to insert the following jobs to the **jobs table**:  
      
    Provide evidence the entire script works by including a screen shot off all four tables with data in them.