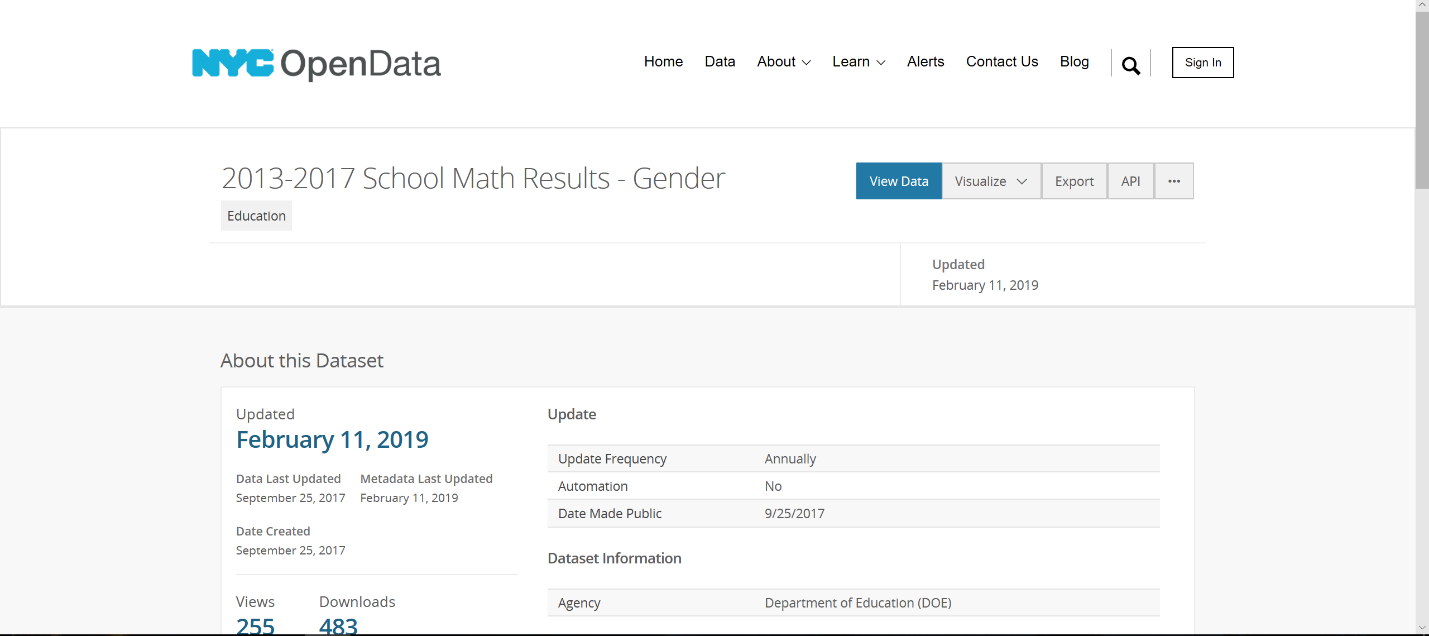
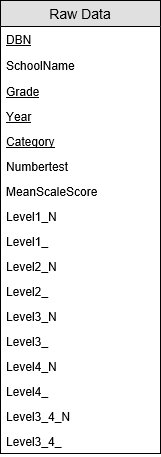
Project Part 1- Jupraj

I chose the 2013-2017 School Math Results based on Gender database from the New York Open data website found here:

<https://data.cityofnewyork.us/Education/2013-2017-School-Math-Results-Gender/x4ai-kstz>

It looks like this:



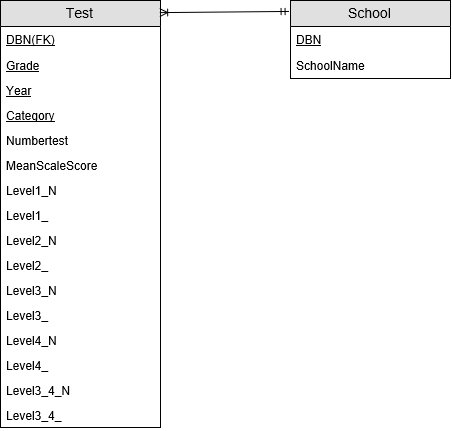
Examining the structure of the dataset downloaded, it will load something similar to:

With about 450,000 rows. Based on the dataset, I believe the following dependencies exist in the data:

DBN -> SchoolName

DBN, Grade, Year, Category -> SchoolName, NumberTest, MeanScaleScore, Level1\_N, Level1\_, Level2\_N, Level2\_,Level3\_N, Level3\_,Level4\_N,Level4\_,Level3\_4\_N,Level3\_4\_

After normalizing the data we get:



By sampling the data by sorting, I think there will be one school for every 10 tests. My guess to cardinality is:

Test:45,000

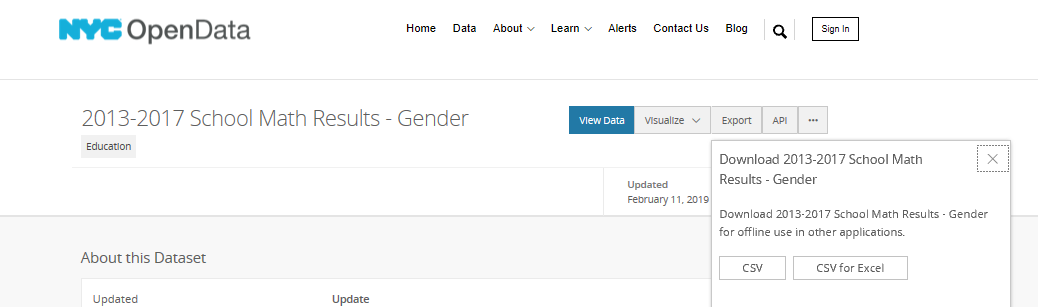
School 4,500

We will create the Test and School table later in this chapter when we are normalizing the data after loading the data set using SQLLDR.

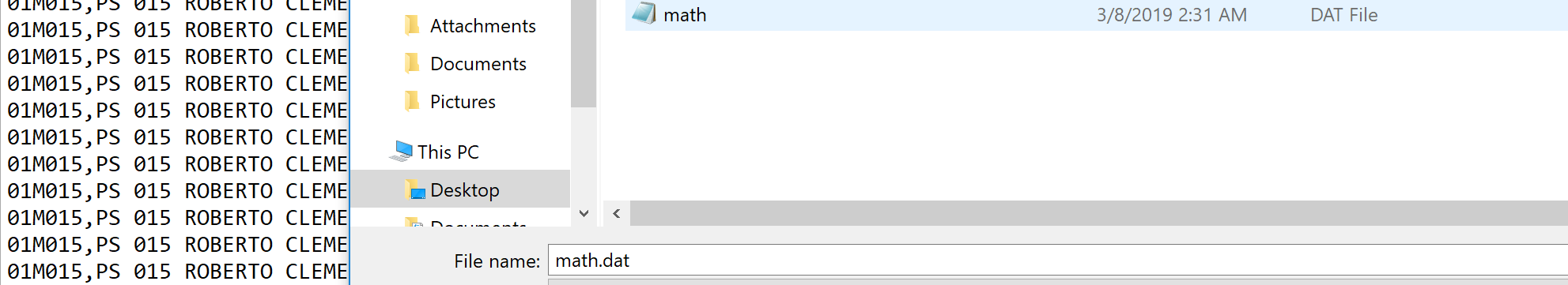
Loading the data using SQLLDR

The first step in order to load a dataset into SQL is to find the dataset you wish implement. https://opendata.cityofnewyork.us/ is a popular website for open data sets in New York. For this example we will be using the 2013-2017 School Math Results based on Gender, found at https://data.cityofnewyork.us/Education/2013-2017-School-Math-Results-Gender/x4ai-kstz.

Once you have found the dataset you will be using, you must download the dataset as a csv file.



Once you have successfully downloaded the dataset as a csv file, you must now open it in a text editor and save it as a .DAT file. For this example, the .DAT file’s name is math.dat.



Now that you have a .dat that SQL can later on use to read data from, you now need to create a table in SQL that will hold the values inside the .dat file. For this example, our table will be called mathdb and the table structure will be shown below in figure 1-1:

CREATE TABLE mathdb(

DBN varchar2(100),

SchoolName varchar2(100),

Grade varchar2(20),

Year varchar2(100),

Category varchar2(100),

NumberTest varchar2(100),

MeanScaleScore varchar2(100),

Level1\_N number,

Level1\_ number,

Level2\_N number,

Level2\_ number,

Level3\_N number,

Level3\_ number,

Level4\_N number,

Level4\_ number,

Level3\_4\_N number,

Level3\_4\_ number

);

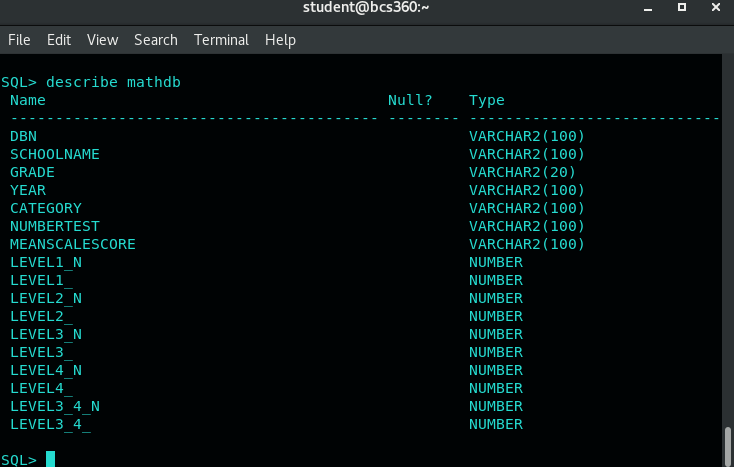


Figure 1-1

Now that you have a table to store the data in, you will need to create a .CTL file to tell the computer how you want SQL to read in your data from the file. To create a .CTL file, you will need to open a text editor such as Notepad. For this example, the .CTL file will be shown below:

OPTIONS (skip=1) //this is saying skip the first row which shows what is in each column. We just need //the data, not the column names

LOAD DATA INFILE math.dat // use the data from math.dat for the data

INTO TABLE mathdb //which table the data will be loaded into using SQLLDR shown later in this //chapter

REPLACE FIELDS TERMINATED BY ',' // using ‘,’ as a delimiter to read in the data

OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS

(DBN, SchoolName, Grade, Year, Category, NumberTest, MeanScaleScore, level1\_N,level1\_,

level2\_n,level2\_, level3\_N,level3\_,level4\_N,level4\_,level3\_4\_N,level3\_4\_) // the columns that will be //loaded.

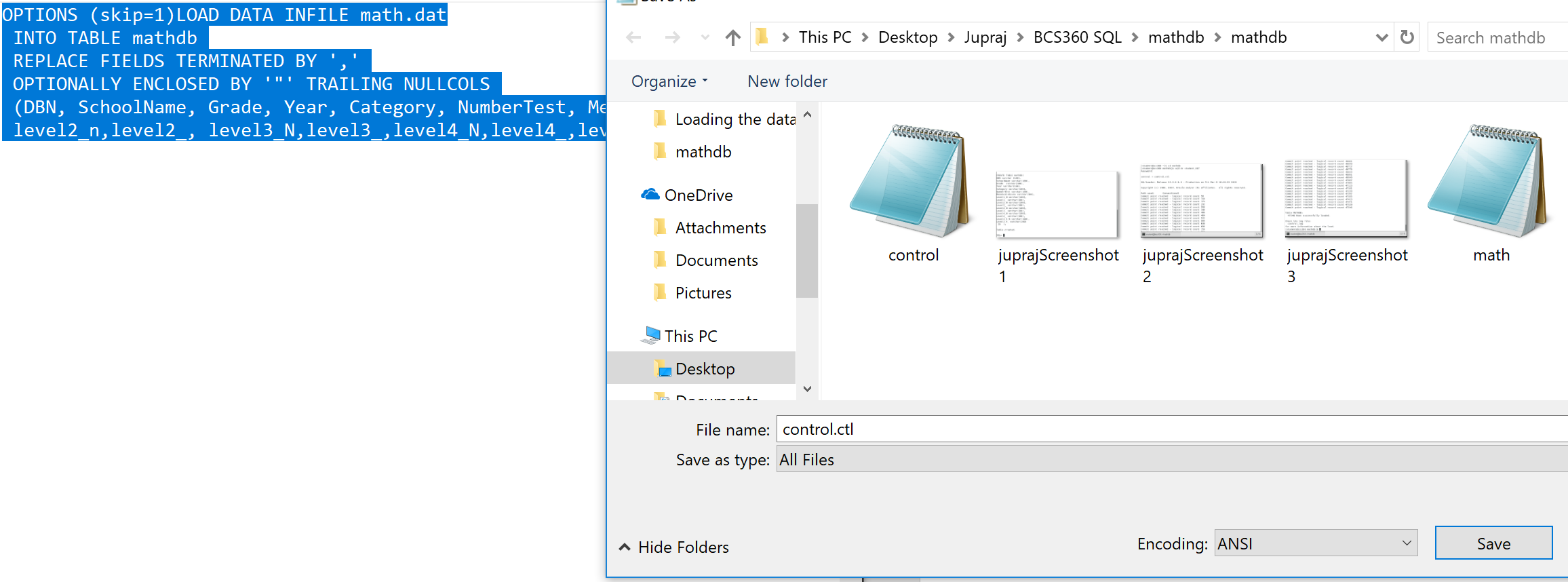


Figure 1-2

Be sure to save the file as a .CTL file as shown in the Figure 1-2 above. For this example, the .CTL file is called control.ctl.

Now that you have a .dat file, a table created inside SQL to store the data, and a .ctl file to tell SQL how to read the data, it is time to use SQLLDR. A SQLLDR statement will combine all of the steps we did above to load all of the data into SQL. For this example, our SQLLDR statement will be shown in figure 2-1 below:

Sqlldr \*username/\*password

control = control.ctl

// Note: The Data statement is not needed since it specified inside the .ctl file

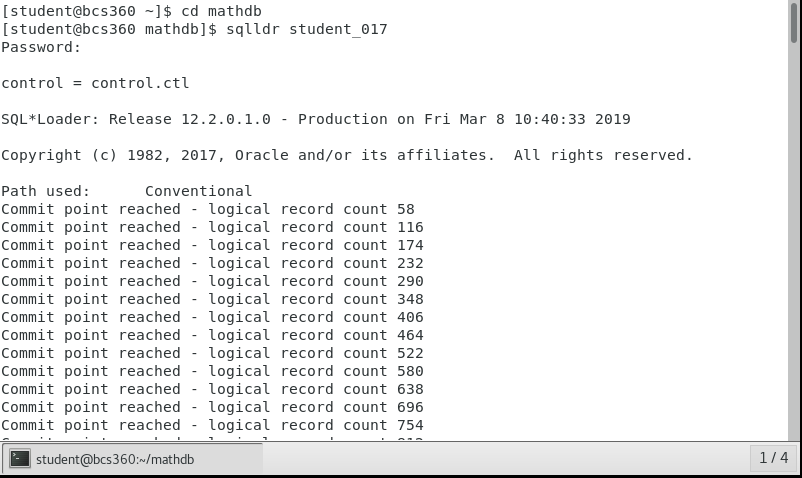


Figure 2-1

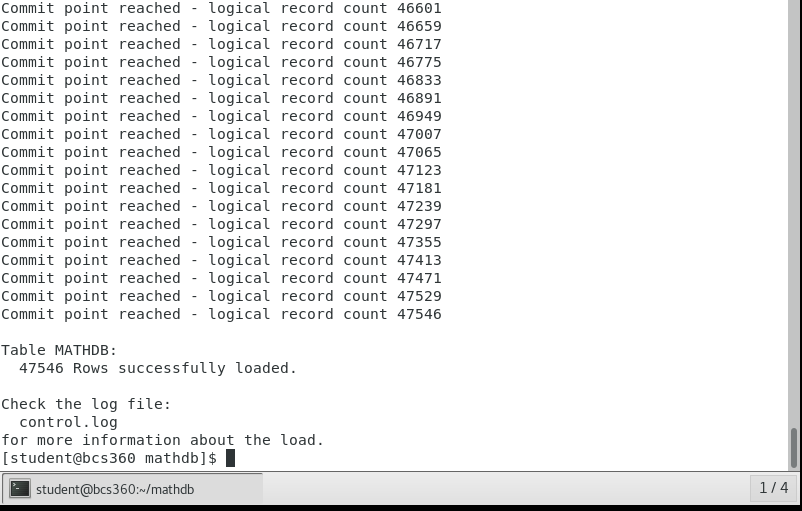


Figure 2-2

47,546 rows have been loaded by the SQLLDR statement

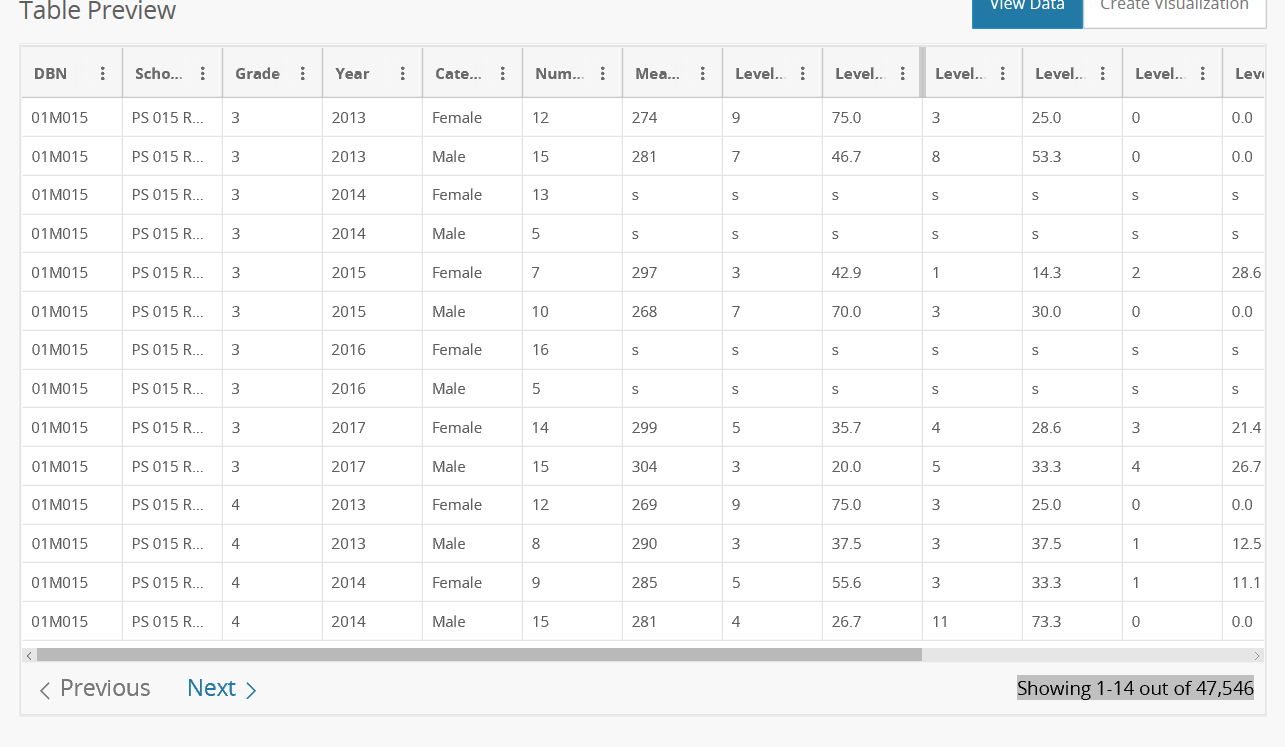


Figure 2-3

There are 47,546 rows in the original dataset

If the .CTL file, .dat file, or create table statement is incorrect then you will have a different number of tuples loaded from SQLLDR because the entire dataset was not loaded properly. To check if all of these are correct, run SQLLDR and keep track of how many rows have been successfully created. If the number of rows matches the original dataset, all the files have been successfully created. This is shown in the figures 2-2 and 2-3 above.

Now that we have all of the records inside of a table, it is time to normalize the dataset and distribute the data into the corresponding tables. This will be shown later in the chapter.

Normalizing and Creating the Data

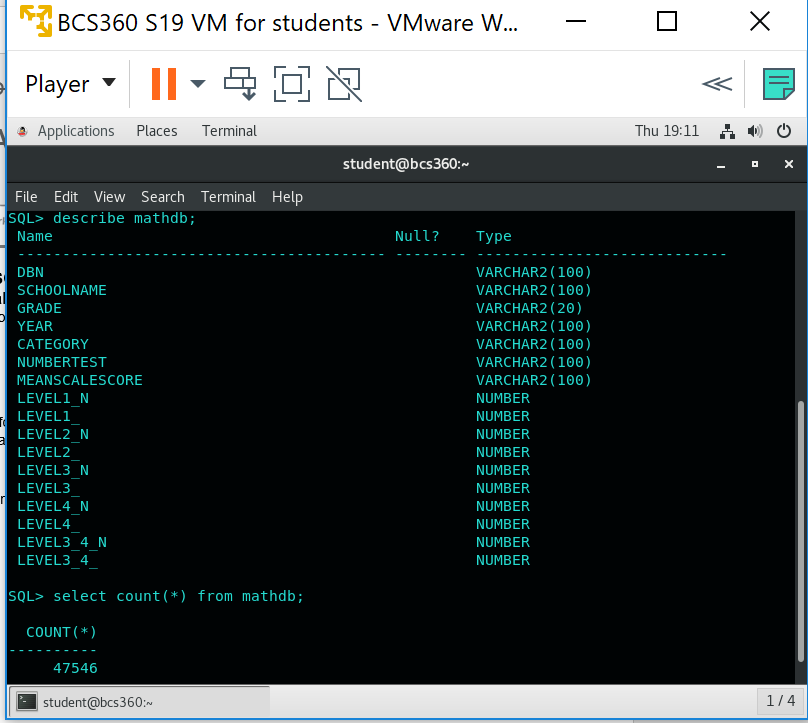
As of right now, all of the data is stored within the mathdb table as seen in figure 4-1.

Figure 4-1

In order to normalize this table, we must first find all of the functional dependencies. A functional dependency tells you if a unique attribute or attributes can determine other attributes. For example, in the mathdb if we want to retrieve a unique record we need the DBN, GRADE, YEAR, and CATEGORY. Once we have all of these attributes, we can determine all of the other attribute inside of the table. To write this functional dependency, we would say:

DBN, GRADE, YEAR, CATEGORY -> SCHOOLNAME, NUMBERTEST, MEANSCALESCORE, LEVEL1\_N, LEVEL1, LEVEL2\_N, LEVEL2, LEVEL3\_N, LEVEL3\_, LEVEL4\_N, LEVEL4\_, LEVEL3\_4\_N, LEVEL3\_4\_

However, the mathdb is not normalized because there is another functional dependency within the database. This functional dependency is: DBN -> SCHOOLNAME. What this means is that SCHOOLNAME is functionally dependent on DBN so if we have DBN, we are able to find out what the SCHOOLNAME is. In order for us to make this table which will give us the school names, we must do a CREATE TABLE AS SELECT statement. We will call this table SCHOOL, and in the create table statement, we will select all of the DINSTINCT DBNs from the mathdb. This create table will be shown below:

CREATE TABLE SCHOOL AS SELECT DISTINCT DBN, SCHOOLNAME FROM MATHDB;

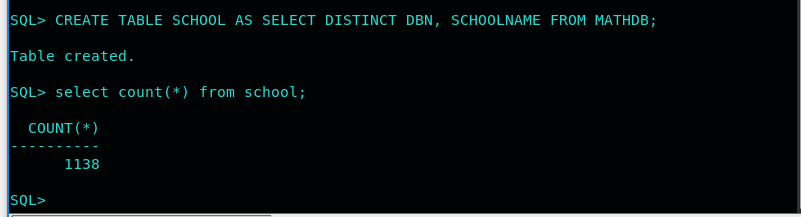


Figure 4-2

After running the create table statement, we can select the count (\*) from the new table SCHOOL to see how many unique schools are inside the dataset. This select statement is shown in Figure 4-2 above.

Now that we have created the SCHOOL table, we must revisit the original functional dependency. DBN, GRADE, YEAR, CATEGORY -> SCHOOLNAME, NUMBERTEST, MEANSCALESCORE, LEVEL1\_N, LEVEL1, LEVEL2\_N, LEVEL2, LEVEL3\_N, LEVEL3\_, LEVEL4\_N, LEVEL4\_, LEVEL3\_4\_N, LEVEL3\_4\_. Now that we are able to determine the SCHOOLNAME from the SCHOOL table, SCHOOLNAME inside the first functional dependency does not need to exist. What’s left of this function dependency is this:

DBN, GRADE, YEAR, CATEGORY -> NUMBERTEST, MEANSCALESCORE, LEVEL1\_N, LEVEL1, LEVEL2\_N, LEVEL2, LEVEL3\_N, LEVEL3\_, LEVEL4\_N, LEVEL4\_, LEVEL3\_4\_N, LEVEL3\_4\_.

The functional dependency above will give all of the test data from the mathdb. We can create a table named TEST which will store all of this data for us. We will also include a count to see if all of the tests from the original mathdb table have transferred over. The select statement will be shown below and in figure 4-3:

CREATE TABLE TEST AS SELECT DBN, GRADE, YEAR, CATEGORY, NUMBERTEST, MEANSCALESCORE, LEVEL1\_N, LEVEL1\_, LEVEL2\_N, LEVEL2\_, LEVEL3\_N, LEVEL3\_, LEVEL4\_N, LEVEL4\_, LEVEL3\_4\_N, LEVEL3\_4\_ FROM MATHDB;

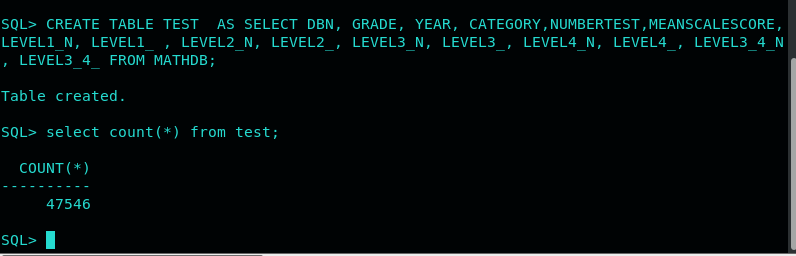


Figure 4-3

Now we have a normalized dataset because there are no more functional dependencies inside of our data and we have successfully transferred all of the data from the original mathdb into our SCHOOL and TEST tables. Now we must set some constraints for each of our tables.

For our SCHOOL table we must set a primary key which is DBN because School is functionally dependent on DBN. To create this primary key constraint outside of the create table statement we will have to make an alter table statement. This statement will be shown in figure 4-4 below.

ATLER TABLE SCHOOL ADD CONSTRAINT school\_dbn\_pk primary key (dbn);

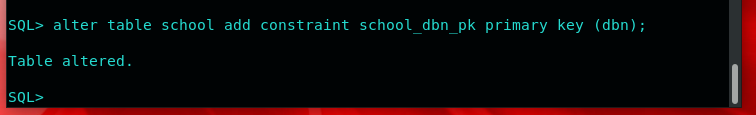


Figure 4-4

For the TEST table, our primary key will be a composite of DBN, YEAR, GRADE, CATEGORY. This alter table statement will be shown below and in figure 4-5.

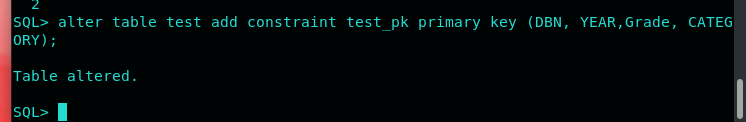
Alter table test add constraint test\_pk primary key (dbn, year, grade, category);

Figure 4-5

However, TEST table also needs a foreign key to the SCHOOL table. This foreign key will be DBN because both tables share this attribute. The foreign key constraint statement will be shown below and in figure 4-6.

Alter table test add constraint test\_fk foreign key (DBN) references school(dbn);

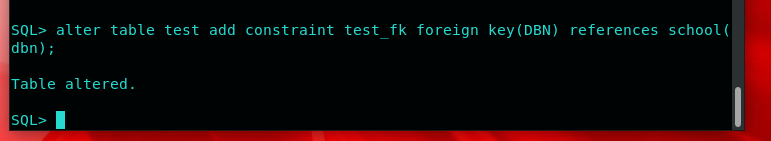


Figure 4-6

Now that we have all of our constraints in the TEST and SCHOOL tables, we should check that they are actually there. To check for constraints, the process is relatively simple. All we have to do is the following statement:

select constraint\_type, constraint\_name from all\_constraints where table\_name = ‘insert table name here’ ;

This statement will give you the name and type of all the constraints inside of a particular table.

For the TEST table the select statement will be: select constraint\_type, constraint\_name from all\_constraints where table\_name = ‘TEST’;

The result will be shown in figure 4-7 below.

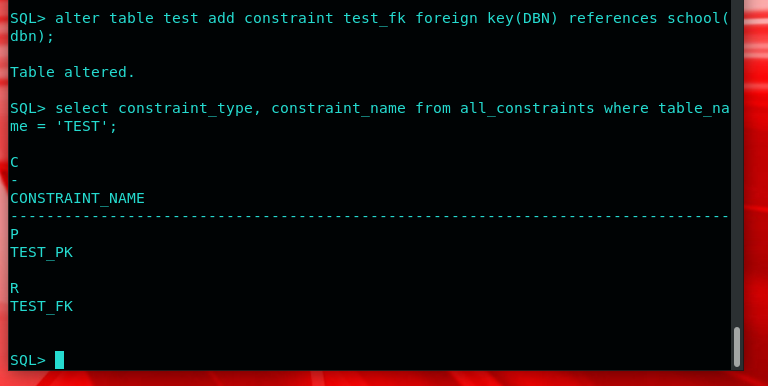


Figure 4-7

For the SCHOOL table, the select statement is: select constraint\_type, constraint\_name from all\_constraints where table\_name = ‘SCHOOL’;

The result will be shown below in figure 4-8.

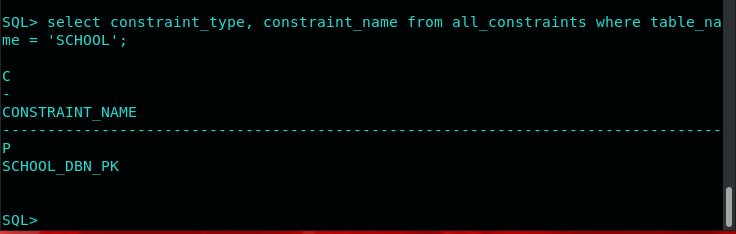


Figure 4-8

As of right now, we normalized mathdb and fully integrated the SCHOOL and TEST tables with constraints. Now we are able to create queries and find specific data from the data set which we will do later in this chapter.

Making Correlated Subqueries with the data

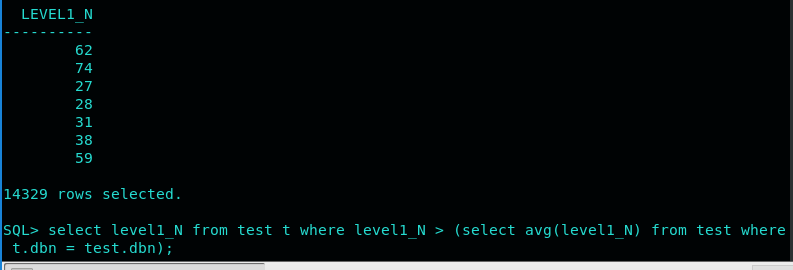
We’ve studied correlated subqueries in the WHERE and HAVING clauses as well as correlated scalar subqueries in the SELECT clauses. Please write a short section explaining

1. Correlated subqueries in the WHERE or HAVING clauses (pick one of those), and
2. correlated scalar subqueries in the SELECT clauses

Now the only step left to do in this chapter is query the data which means finding specific data using SQL statements. For our purposes, we will be creating correlated subqueries. A standard subquery is a query that has another query inside of it. However, a correlated subquery is unique in that the outer query is run first and then the inside query is run. This allows us to select aggregates such as max (), min (), and avg () inside the inner query while also selecting attributes from the outer query without needing a group by statement. You can differentiate between correlated and uncorrelated subqueries by looking at the inner queries. If the inner query is able to run by itself then it is an uncorrelated query. If the inner query is not able to run just by itself then it is a correlated subquery.

An example of a correlated subquery would be finding the DBN of all the schools who scored higher than average on LEVEL1\_N. This statement would look like the following:

Select level1\_N from test t where level1\_N > (select avg(level1\_N) from test where t.dbn = test.dbn);



This correlated subquery is inside the where clause, but you can also create correlated subqueries inside a having clause or even in the select clause. When you create a correlated subquery in the select clause, it is called a scalar subquery.

An example of this would be finding the schoolname and score of LEVEL2\_N of every single school. This scalar subquery would look like:

Select schoolname, (select max(level2\_N) from test where test.dbn = school.dbn) as testtwo from school order by DBN;



This scalar subquery gives us all of the schools and their score for LEVEL2\_N.

From this chapter we have learned how to find a dataset online, the steps taken to load that data into SQL, how to normalize that data and break it down into smaller tables, and finally how to query the data.