Vincent Nguyen
Class: CS 669 – Fall 1

Section One – Absolute Fundamentals

Section Background

In this section, you learn the absolute fundamentals of SQL – creating and dropping a table, getting data into the table, listing the data in the table, and deleting and updating the data. You will be working with a Car dealership table that has basic information about the cars they have for sale. When you have completed some steps in the section, the Cars table will look as illustrated below.

Cars Table

CarBrand: VARCHAR(64)	CarModel: VARCHAR(50)	AcquiredDate: DATE	Price: DECIMAL(10,2)
Ford	Econoline Full-Size Van	15-AUG-2021	29,995.00

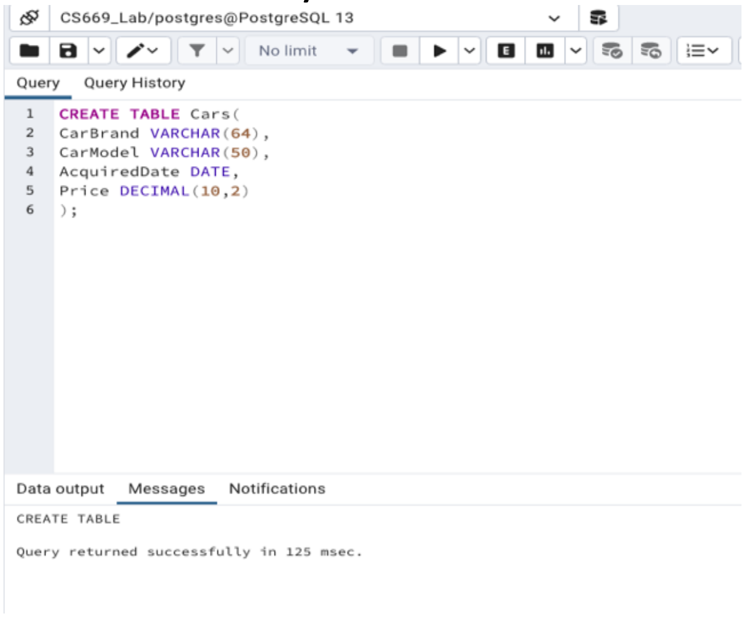
You will create this table and try out SQL commands using the table.

Do not worry if you do not recognize the structure and datatypes in the table above. The Lab 1 Explanation document and supporting lecture and textbook readings give you the information you need. Start reading the explanation document first, then iteratively complete the steps below. Each step below has an accompanying explanation in the explanation document.

For each step that requires SQL, *make sure to capture a screenshot of the command and the results of its execution.* Submissions that do not contain screenshots will be returned to you. A screenshot is more legible if you use one of the many free tools to capture only the relevant portion of the screen, rather than capturing the entire application window. A few steps ask for explanations rather than SQL; no screenshot is needed for such steps.

Section Steps

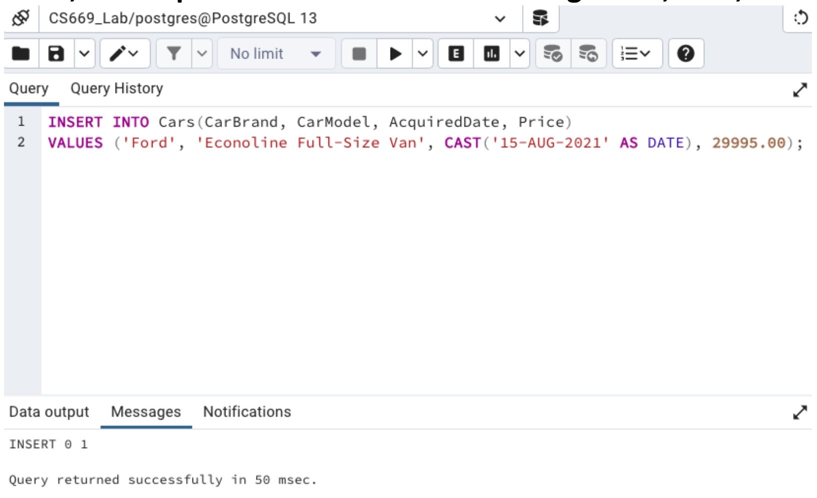
1. ***Creating a Table*** **– Create the Cars table. As a reminder, make sure to follow along in the Lab 1 Explanations document as it shows you how to create tables and complete the other steps.**



```
CREATE TABLE Cars(
CarBrand VARCHAR(64),
CarModel VARCHAR(50),
AcquiredDate DATE,
Price DECIMAL(10,2)
);
```

```
CREATE TABLE

Query returned successfully in 125 msec.
```

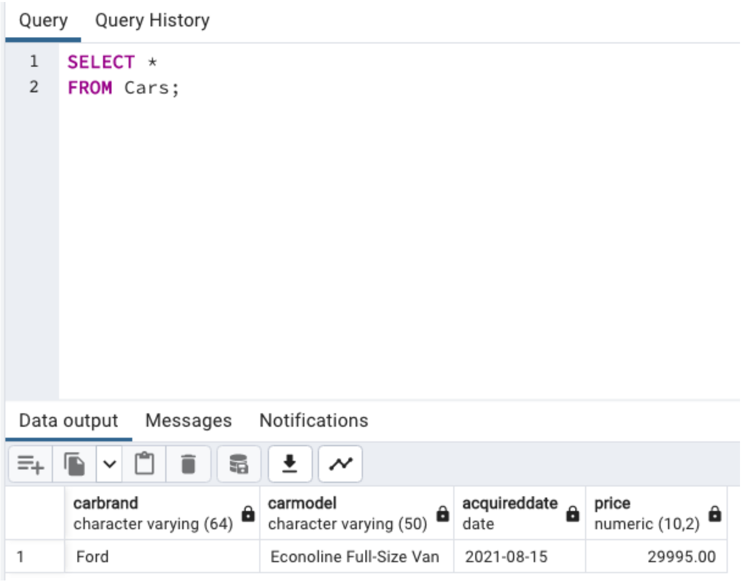
2. ***Inserting a Row*** **– Insert the first row where the Car Brand name is "Ford", the Car Model is "Econoline Full-Size Van", the acquisition date for the car is August 15,2021, and the price is $29,995.00.**



```
INSERT INTO Cars(CarBrand, CarModel, AcquiredDate, Price)
VALUES ('Ford', 'Econoline Full-Size Van', CAST('15-AUG-2021' AS DATE), 29995.00);
```

```
INSERT 0 1

Query returned successfully in 50 msec.
```

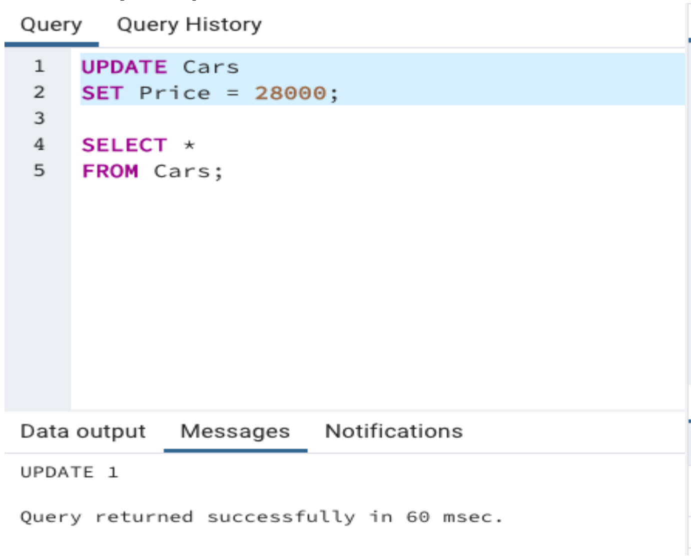
3. ***Selecting All Rows*** **– Select all rows in the table to view the row you inserted.**



```sql
SELECT *
FROM Cars;
```

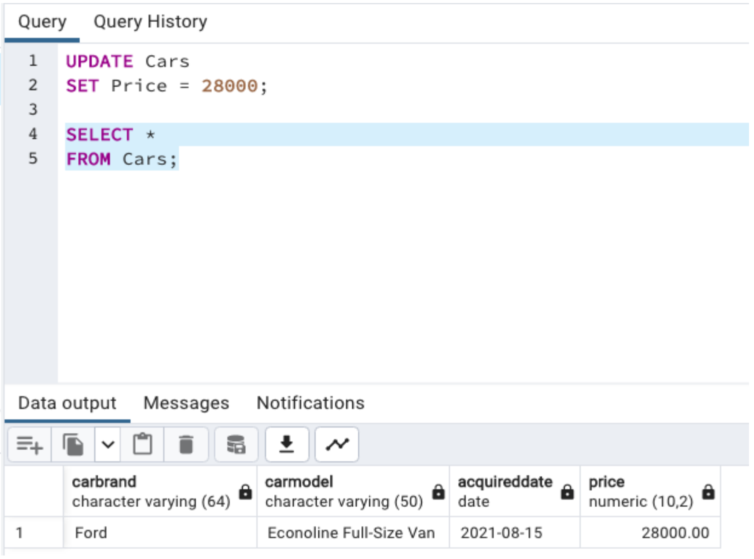
	carbrand character varying (64)	carmodel character varying (50)	acquireddate date	price numeric (10,2)
1	Ford	Econoline Full-Size Van	2021-08-15	29995.00

4. ***Updating All Rows*** **– Update the price of the row in the table to $28,000, then select all rows in the table to view the row you updated.**



```sql
UPDATE Cars
SET Price = 28000;

SELECT *
FROM Cars;
```

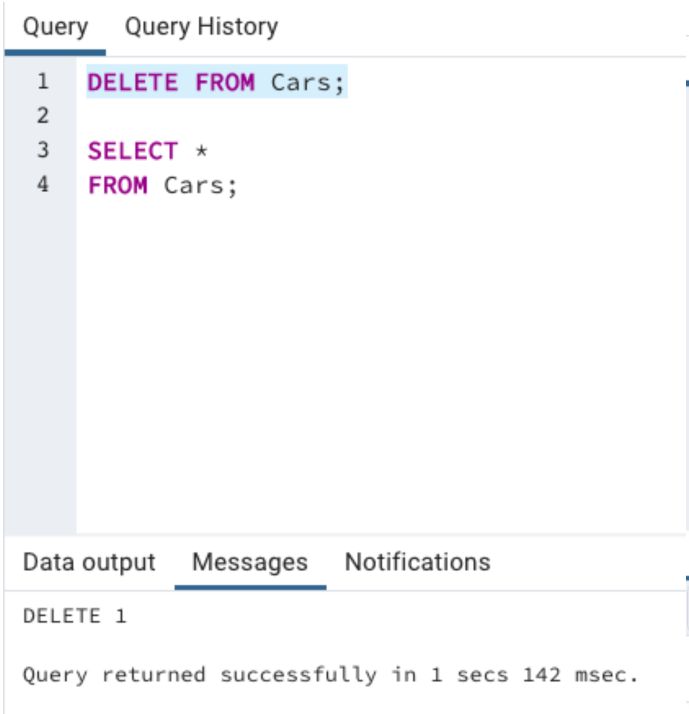
```
UPDATE 1

Query returned successfully in 60 msec.
```



```sql
UPDATE Cars
SET Price = 28000;

SELECT *
FROM Cars;
```

	carbrand character varying (64)	carmodel character varying (50)	acquireddate date	price numeric (10,2)
1	Ford	Econoline Full-Size Van	2021-08-15	28000.00

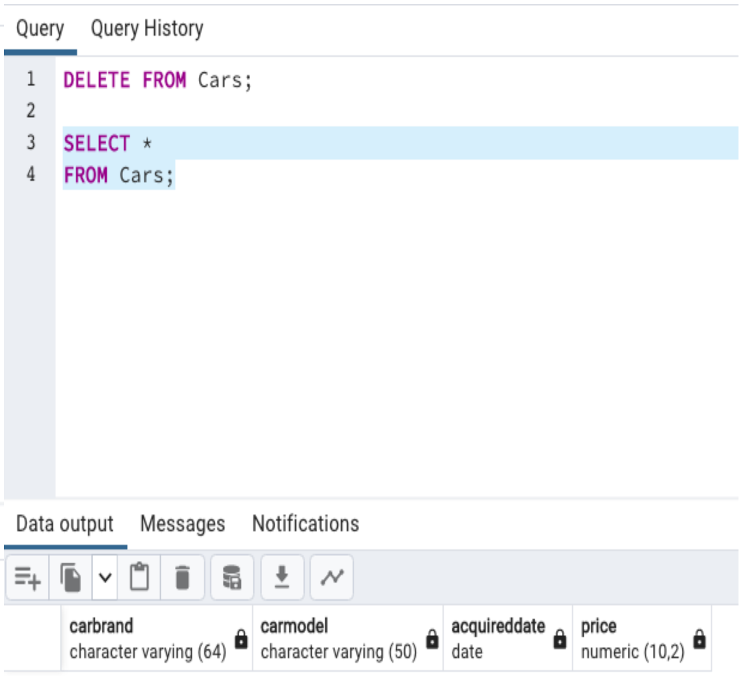
5. ***Deleting All Rows*** **– Remove all rows from the table, then select all rows in the table to verify there are no rows.**


Query | Query History

```
DELETE FROM Cars;

SELECT *
FROM Cars;
```

Data output | Messages | Notifications

```
DELETE 1

Query returned successfully in 1 secs 142 msec.
```

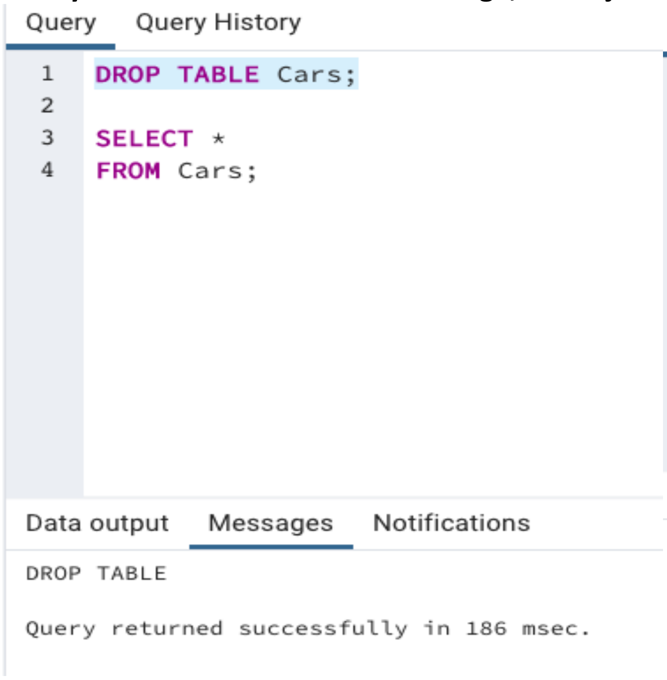

Query | Query History

```
DELETE FROM Cars;

SELECT *
FROM Cars;
```

Data output | Messages | Notifications

carbrand character varying (64)	carmodel character varying (50)	acquireddate date	price numeric (10,2)

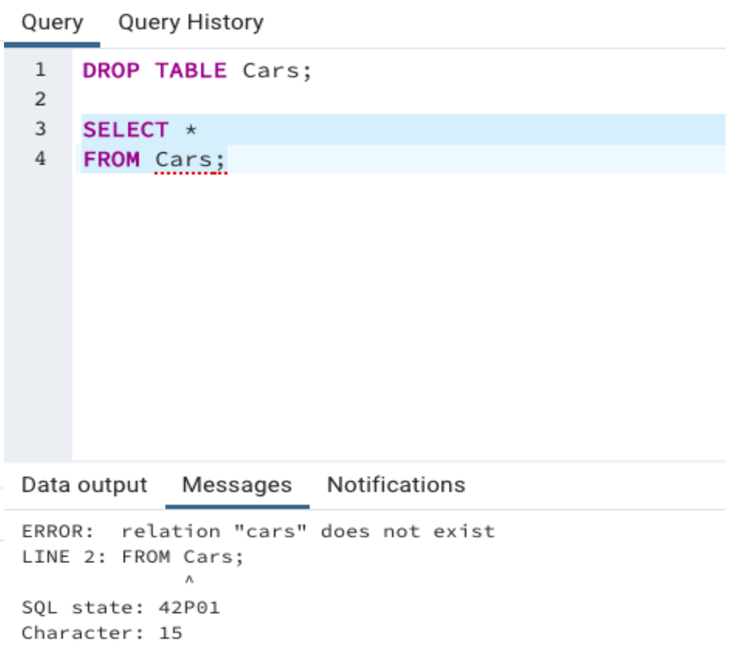
6. ***Dropping a Table*** **– Drop the Cars table, then select all rows in the table to verify the table doesn't exist. Explain how you would use the error message, in conjunction with the SELECT command, to diagnose the error.**


Query | Query History

```
DROP TABLE Cars;

SELECT *
FROM Cars;
```

Data output | Messages | Notifications

```
DROP TABLE

Query returned successfully in 186 msec.
```


Query | Query History

```
DROP TABLE Cars;

SELECT *
FROM Cars;
```

Data output | Messages | Notifications

```
ERROR:  relation "cars" does not exist
LINE 2: FROM Cars;
             ^
SQL state: 42P01
Character: 15
```

Because we use DROP TABLE command to drop a table from database. After we executed DROP TABLE Cars command, it deleted completely Cars table (structures, indexes etc.). It is necessary to use the error massage along with SELECT * command because both do not provide completely error information on their own. SQL command SELECT * FROM Cars; is read table, but the error message tells that now Cars table no longer exists and Line 2 where the root of error begins (Cars word (Cars table) which makes the source of the error) because We already deleted table by using DROP TABLE Cars.

Section Two – More Precise Data Handling

Section Background

In this section, you enhance your skills by more precisely working with data. In the prior section, you learned to work with all rows in the table. In this section, you add to that by learning to pinpoint specific rows to be retrieved, modified, or deleted. You also learn how to add SQL constraints to your table, and to work with nulls.

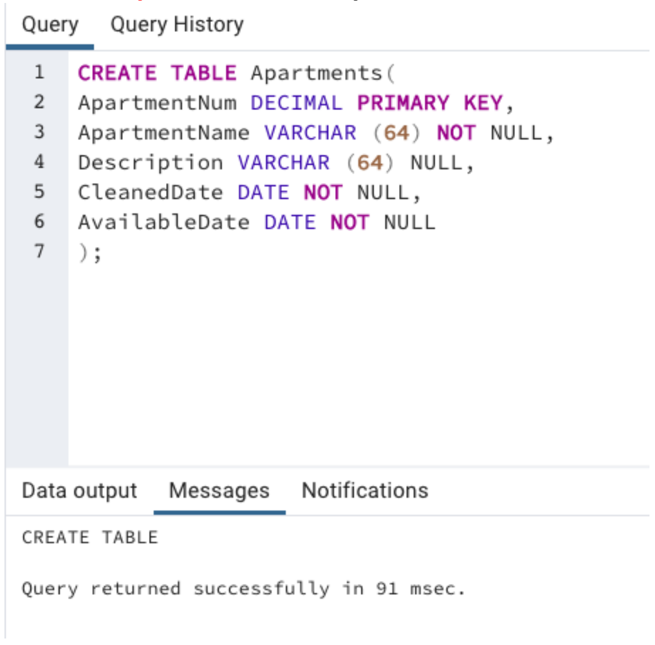
You will work with an Apartments table, which will ultimately look like the below when all steps have been completed.

Apartments Table

ApartmentNum: DECIMAL Primary Key	ApartmentName: VARCHAR(64) NOT NULL	Description: VARCHAR(64) NULL	CleanedDate: DATE NOT NULL	AvailableDate: DATE NOT NULL
498	Deer Creek Crossing	Great view of Riverwalk	19-APR-2022	25-APR-2022
128	Town Place Apartments	Convenient walk to Parking	20-MAY-2022	25-MAY-2022
316	Paradise Palms		02-JUN-2021	08-JUN-2021

Section Steps

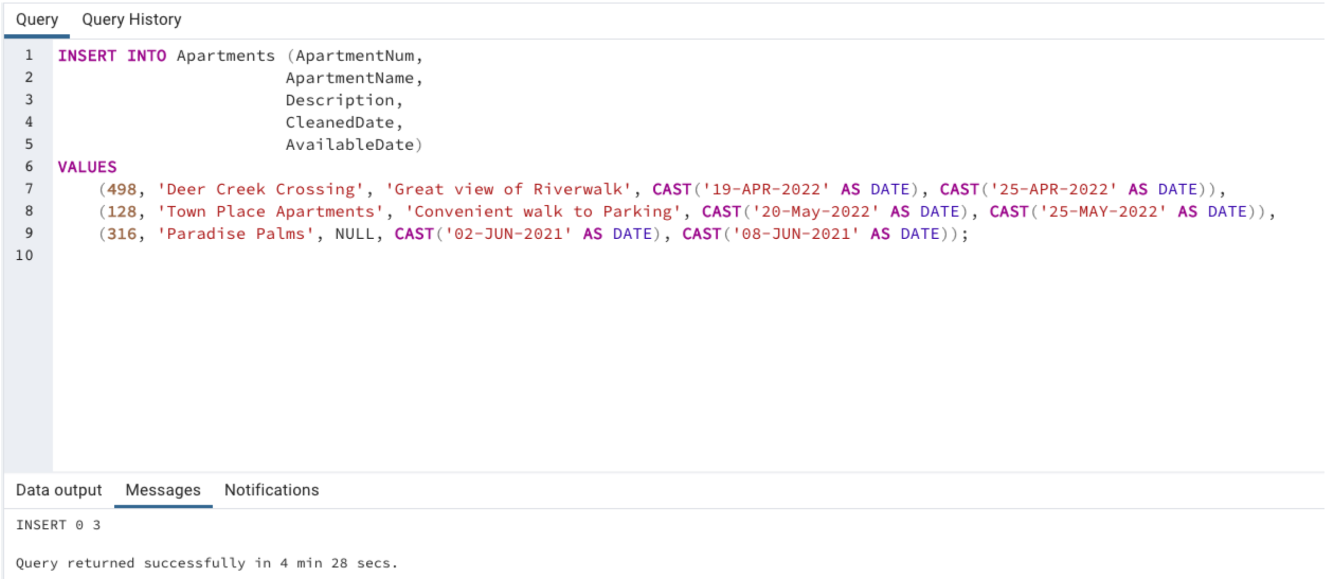
7. ***Table Setup*** **– Create the Apartments table with its columns, datatypes, and constraints.**



```
CREATE TABLE Apartments(
ApartmentNum DECIMAL PRIMARY KEY,
ApartmentName VARCHAR (64) NOT NULL,
Description VARCHAR (64) NULL,
CleanedDate DATE NOT NULL,
AvailableDate DATE NOT NULL
);
```

```
CREATE TABLE

Query returned successfully in 91 msec.
```

8. ***Table Population*** **– Insert the rows illustrated in the figure above. Note that the description for Apartment 316 at Paradise Palms is null. Then select all rows from the Apartments table to show that the inserts were successful.**

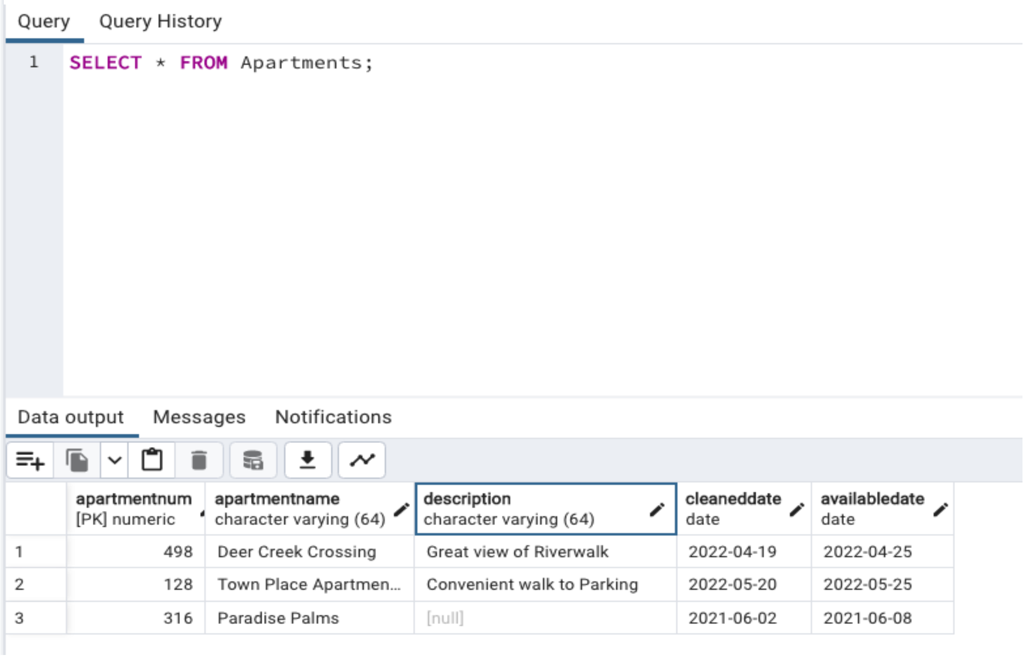


Query Query History

```
INSERT INTO Apartments (ApartmentNum,
                        ApartmentName,
                        Description,
                        CleanedDate,
                        AvailableDate)
VALUES
    (498, 'Deer Creek Crossing', 'Great view of Riverwalk', CAST('19-APR-2022' AS DATE), CAST('25-APR-2022' AS DATE)),
    (128, 'Town Place Apartments', 'Convenient walk to Parking', CAST('20-May-2022' AS DATE), CAST('25-MAY-2022' AS DATE)),
    (316, 'Paradise Palms', NULL, CAST('02-JUN-2021' AS DATE), CAST('08-JUN-2021' AS DATE));
```

Data output Messages Notifications

```
INSERT 0 3

Query returned successfully in 4 min 28 secs.
```



Query Query History

```
SELECT * FROM Apartments;
```

Data output Messages Notifications

	apartmentnum [PK] numeric	apartmentname character varying (64)	description character varying (64)	cleaneddate date	availabledate date
1	498	Deer Creek Crossing	Great view of Riverwalk	2022-04-19	2022-04-25
2	128	Town Place Apartmen...	Convenient walk to Parking	2022-05-20	2022-05-25
3	316	Paradise Palms	[null]	2021-06-02	2021-06-08

9. ***Invalid Insertion*** **– The following values leave the Apartment Name with no value.**

ApartmentNum = 252
ApartmentName = NULL
Description = Close to Downtown shops
CleanedDate = 17-JUL-2020
AvailableDate = 13-JUL-2020

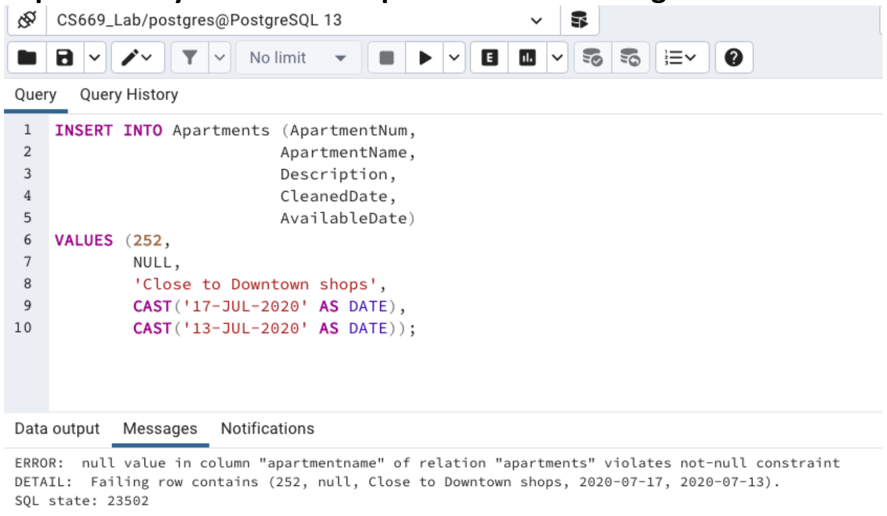
a. In your own words, explain what a null value is.

- Null Value is no value at all. A Null Value is the missing of any data value and Null Value is not a blank.

b. **In your own words, explain what a NOT NULL constraint is.**

- NOT NULL constraint is mean that when we insert value into a column (field), it does not accept NULL VALUE (Must have value), It forces column (field) to always contain a value otherwise it will raise error when we try to insert NULL VALUE into the column(field) which applied NOT NULL constraint.

c. **Attempt to insert the values as listed and explain how the database handles this attempt.**
Explain how you would interpret the error message conclude that the location column is missing a required value.



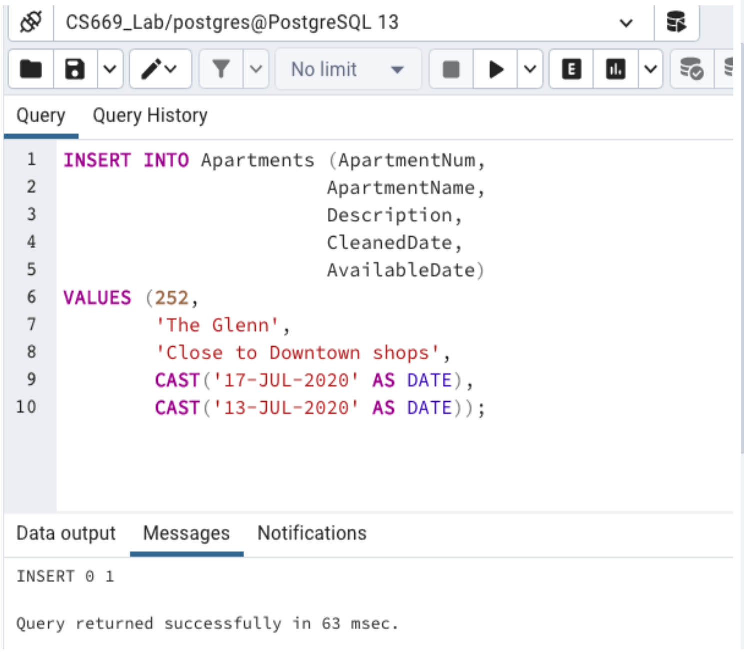
```sql
INSERT INTO Apartments (ApartmentNum,
                        ApartmentName,
                        Description,
                        CleanedDate,
                        AvailableDate)
VALUES (252,
        NULL,
        'Close to Downtown shops',
        CAST('17-JUL-2020' AS DATE),
        CAST('13-JUL-2020' AS DATE));
```

```
ERROR:  null value in column "apartmentname" of relation "apartments" violates not-null constraint
DETAIL:  Failing row contains (252, null, Close to Downtown shops, 2020-07-17, 2020-07-13).
SQL state: 23502
```

- Because when we set up table, we created ApartmentName column (field) with NOT NULL constraint in it. It means that ApartmentName column only accept a value. So, the pgAdmin raises error message when we try to insert NULL value into ApartmentName. It clearly shows in ERROR message that null value in column “ApartmentName” violates not null constraint.

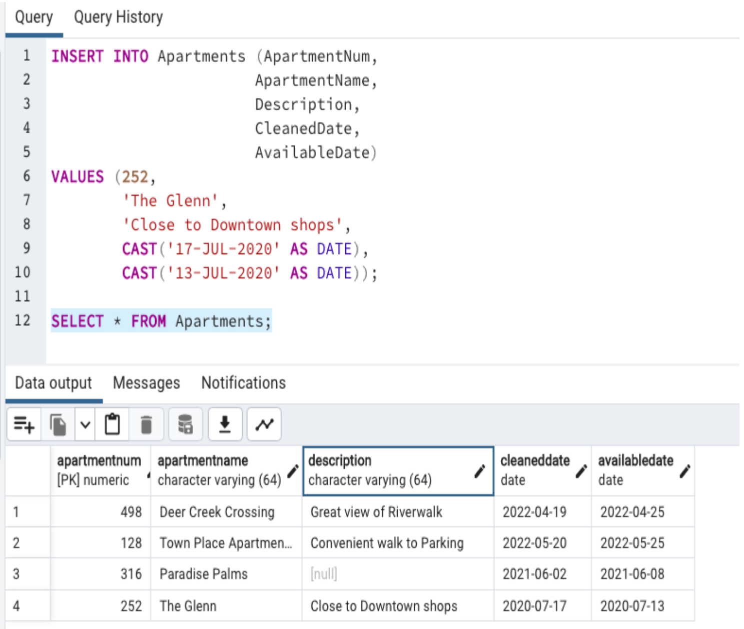
10. ***Valid Insertion*** **– Now insert the row with the Apartment Name intact, with the following values.**

ApartmentNum = 252
ApartmentName = The Glenn
Description = Close to Downtown shops
CleanedDate = 17-JUL-2020
AvailableDate = 13-JUL-2020



```sql
INSERT INTO Apartments (ApartmentNum,
                        ApartmentName,
                        Description,
                        CleanedDate,
                        AvailableDate)
VALUES (252,
        'The Glenn',
        'Close to Downtown shops',
        CAST('17-JUL-2020' AS DATE),
        CAST('13-JUL-2020' AS DATE));
```

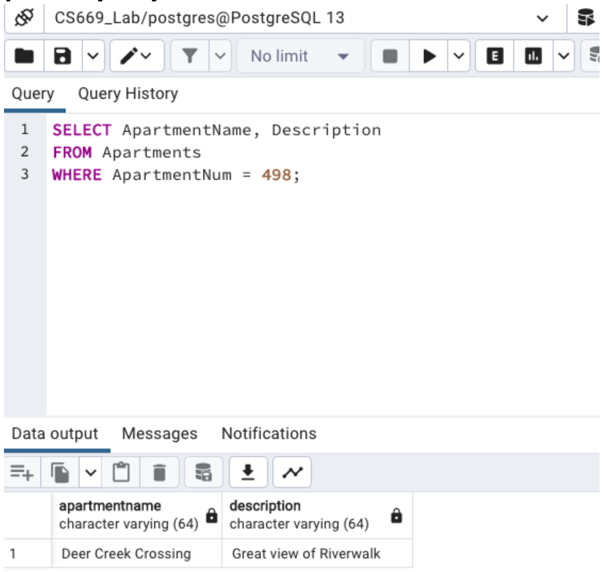
```
INSERT 0 1

Query returned successfully in 63 msec.
```



```sql
INSERT INTO Apartments (ApartmentNum,
                        ApartmentName,
                        Description,
                        CleanedDate,
                        AvailableDate)
VALUES (252,
        'The Glenn',
        'Close to Downtown shops',
        CAST('17-JUL-2020' AS DATE),
        CAST('13-JUL-2020' AS DATE));

SELECT * FROM Apartments;
```

	apartmentnum [PK] numeric	apartmentname character varying (64)	description character varying (64)	cleaneddate date	availabledate date
1	498	Deer Creek Crossing	Great view of Riverwalk	2022-04-19	2022-04-25
2	128	Town Place Apartmen...	Convenient walk to Parking	2022-05-20	2022-05-25
3	316	Paradise Palms	[null]	2021-06-02	2021-06-08
4	252	The Glenn	Close to Downtown shops	2020-07-17	2020-07-13

11. ***Filtered Results*** **– Retrieve only the Apartment Name and the Description for Deer Creek Crossing, using the primary key as the column that determines which row is retrieved.**



CS669_Lab/postgres@PostgreSQL 13

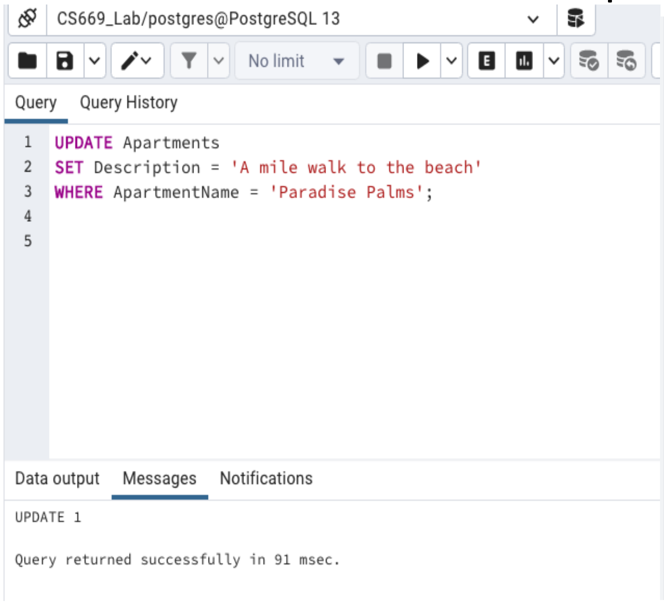
```
SELECT ApartmentName, Description
FROM Apartments
WHERE ApartmentNum = 498;
```

	apartmentname character varying (64)	description character varying (64)
1	Deer Creek Crossing	Great view of Riverwalk

Explain why it is useful to limit the number of rows and columns returned from a SELECT statement.

- Because of efficiency. If we have big database which has a lot of columns and rows, it will take a lot of time and resource to retrieve all columns and rows. For example, if we have more than 1000 columns and 50 rows, it would be unnecessary to retrieve all of columns and rows, it will be very difficult and take time for us to find specific information which we need. So that it is useful to limit the number of rows and columns return from a SELECT statement. In the example above, we have indicated that we would like to see only ApartmentName and Description for Deer Creek Crossing, using Primary Key (ApartmentNum = 498).

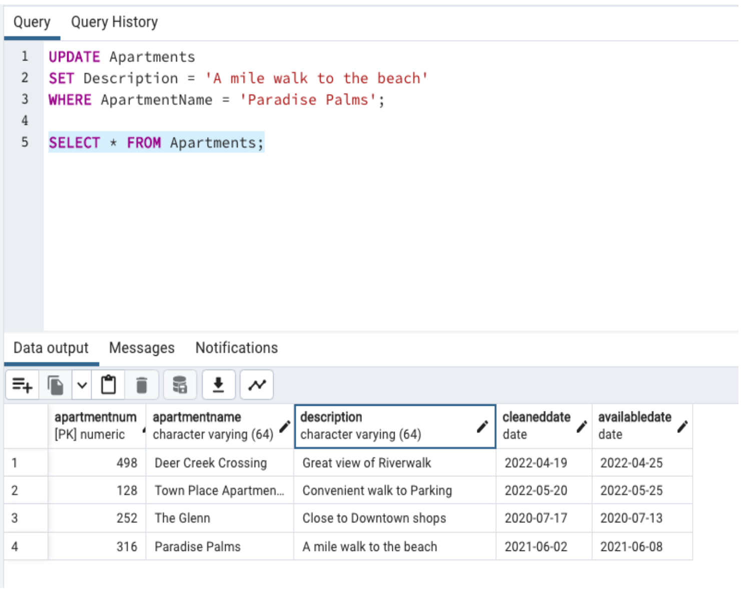
12. ***Targeted Update*** **– The Paradise Palms apartment has no description. Update the row so that its description says "A mile walk to the beach".**
Select all rows in the table to show that the update was successful.



CS669_Lab/postgres@PostgreSQL 13

```
UPDATE Apartments
SET Description = 'A mile walk to the beach'
WHERE ApartmentName = 'Paradise Palms';
```

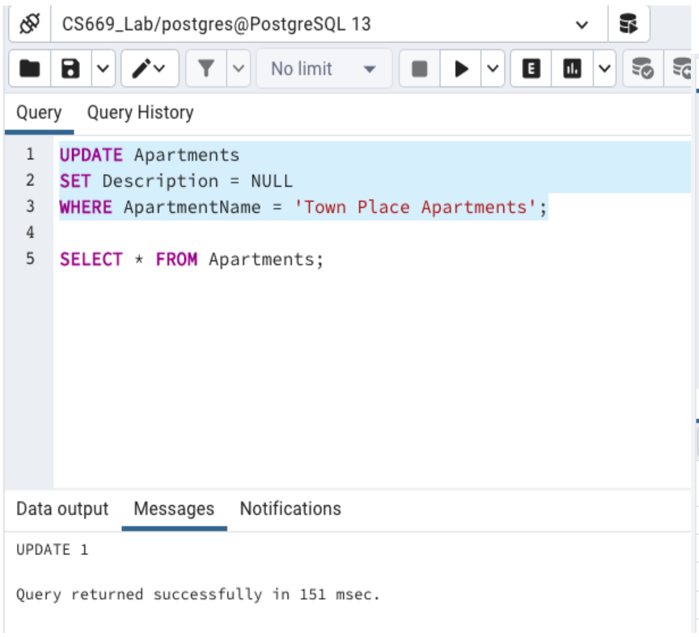
```
UPDATE 1

Query returned successfully in 91 msec.
```



```
UPDATE Apartments
SET Description = 'A mile walk to the beach'
WHERE ApartmentName = 'Paradise Palms';

SELECT * FROM Apartments;
```

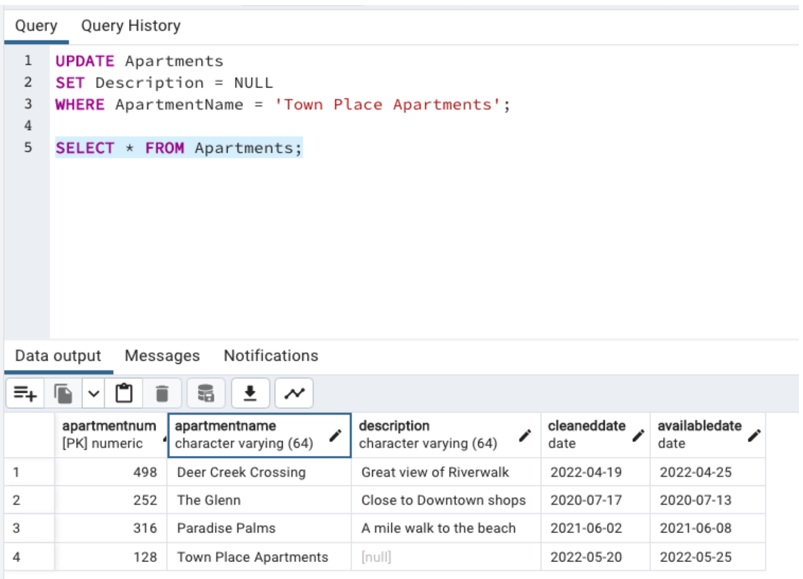
	apartmentnum [PK] numeric	apartmentname character varying (64)	description character varying (64)	cleaneddate date	availabledate date
1	498	Deer Creek Crossing	Great view of Riverwalk	2022-04-19	2022-04-25
2	128	Town Place Apartmen...	Convenient walk to Parking	2022-05-20	2022-05-25
3	252	The Glenn	Close to Downtown shops	2020-07-17	2020-07-13
4	316	Paradise Palms	A mile walk to the beach	2021-06-02	2021-06-08

13. ***Updating to Null*** **– Update the Town Place Apartments so that it no longer has a description (i.e., its description is null).**
Select all rows in the table to show that the update was successful.



```sql
UPDATE Apartments
SET Description = NULL
WHERE ApartmentName = 'Town Place Apartments';

SELECT * FROM Apartments;
```

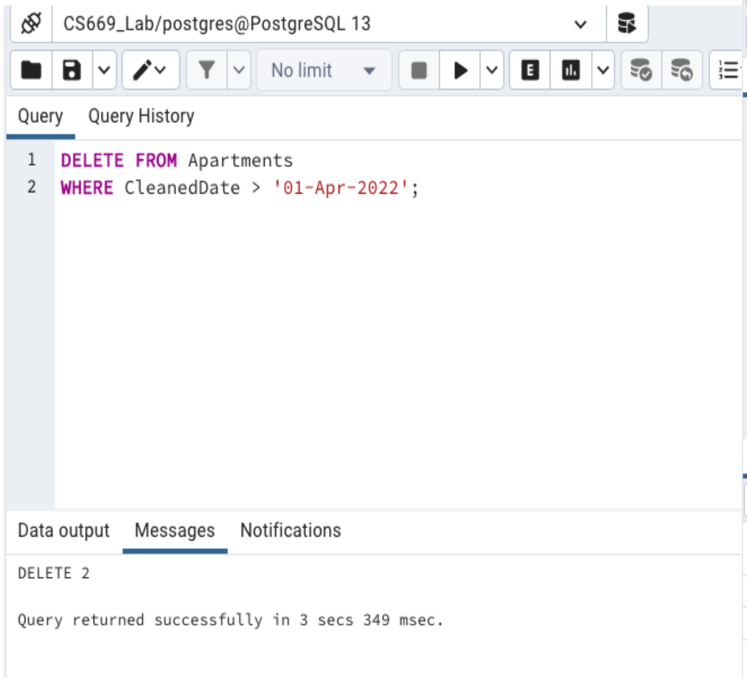
```
UPDATE 1

Query returned successfully in 151 msec.
```



```sql
UPDATE Apartments
SET Description = NULL
WHERE ApartmentName = 'Town Place Apartments';

SELECT * FROM Apartments;
```

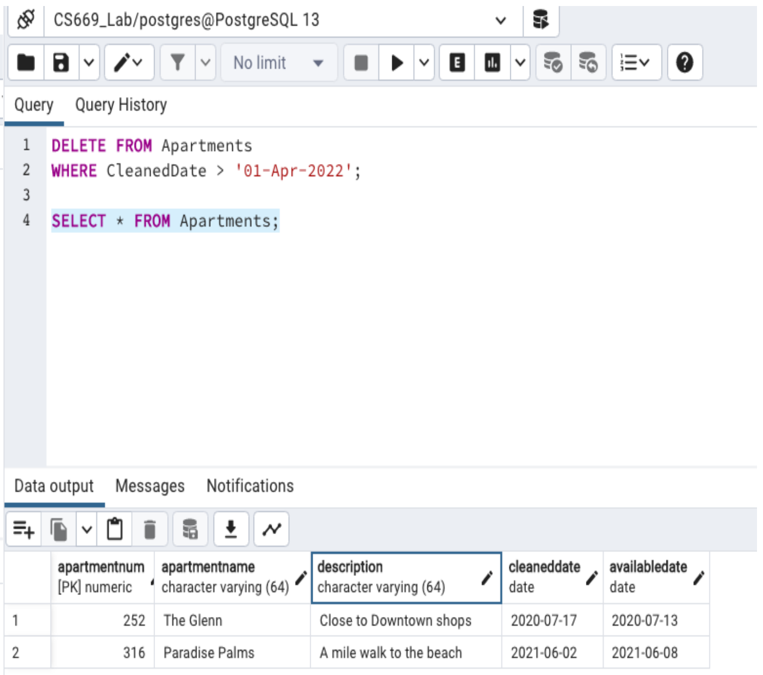
	apartmentnum [PK] numeric	apartmentname character varying (64)	description character varying (64)	cleaneddate date	availabledate date
1	498	Deer Creek Crossing	Great view of Riverwalk	2022-04-19	2022-04-25
2	252	The Glenn	Close to Downtown shops	2020-07-17	2020-07-13
3	316	Paradise Palms	A mile walk to the beach	2021-06-02	2021-06-08
4	128	Town Place Apartments	[null]	2022-05-20	2022-05-25

14. ***Targeted Deletion*** **– Delete all rows where the Cleaned date is greater than April 1, 2022, by using the Cleaned Date column as the determinant of which rows are deleted.**
Select all rows in the table to show the delete was successful.



```sql
DELETE FROM Apartments
WHERE CleanedDate > '01-Apr-2022';
```

```
DELETE 2

Query returned successfully in 3 secs 349 msec.
```



```sql
DELETE FROM Apartments
WHERE CleanedDate > '01-Apr-2022';

SELECT * FROM Apartments;
```

	apartmentnum [PK] numeric	apartmentname character varying (64)	description character varying (64)	cleaneddate date	availabledate date
1	252	The Glenn	Close to Downtown shops	2020-07-17	2020-07-13
2	316	Paradise Palms	A mile walk to the beach	2021-06-02	2021-06-08

Section Three – Data Anomalies and Formats

Section Background

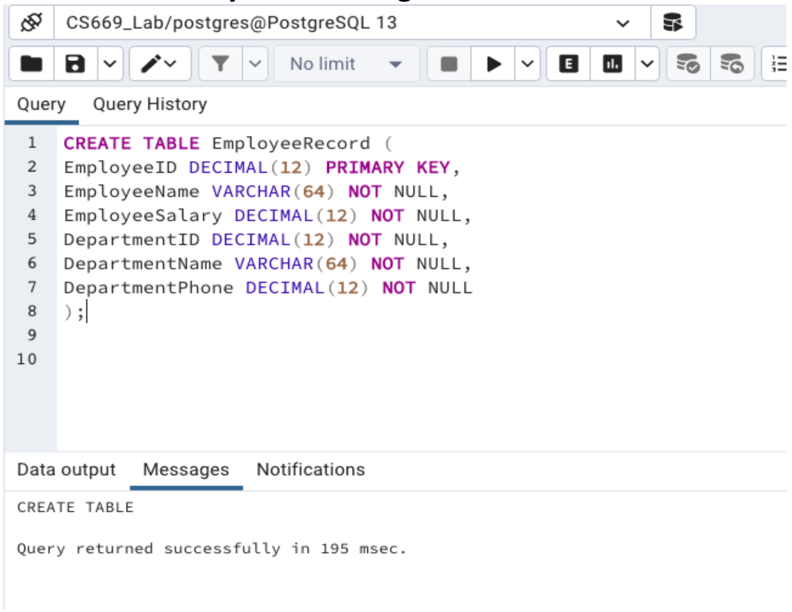
When the same data is repeated multiple times, anomalies can result. In this section, you demonstrate and explore three such anomalies in a relational database – insert, update, and delete anomalies.

Databases provide several advantages over files. In this section, you explore putting the same data in a relational table and in a file, then compare the two.

Section Steps

15. *Data Anomalies* – In this step you demonstrate anomalies that can occur in improperly designed tables.

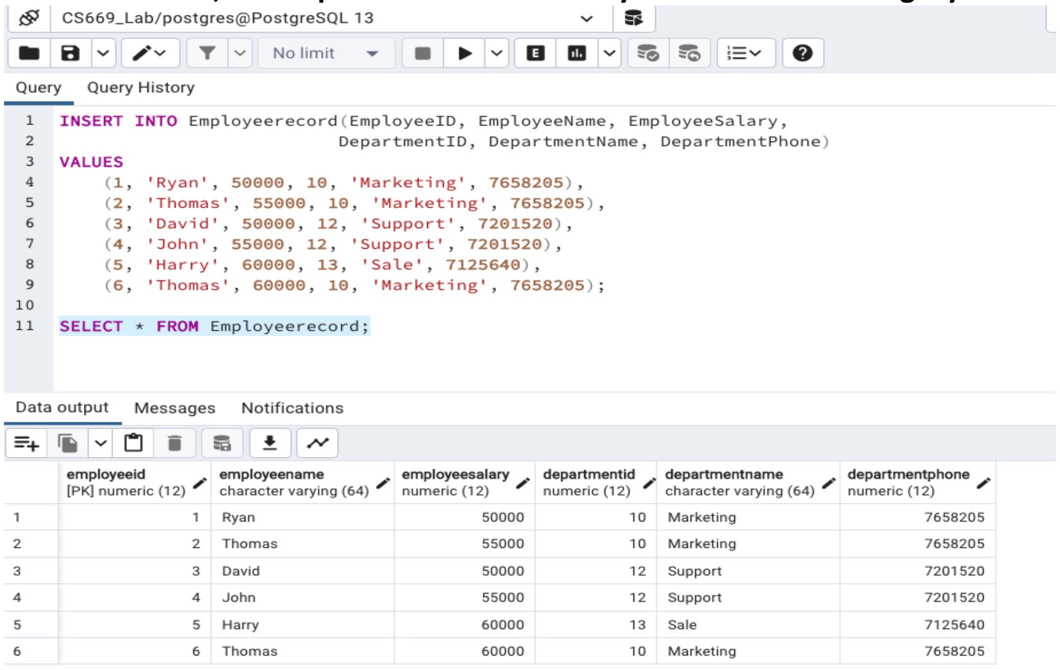
a. Create a table of your choosing that has at least three columns.



```
CREATE TABLE EmployeeRecord (
EmployeeID DECIMAL(12) PRIMARY KEY,
EmployeeName VARCHAR(64) NOT NULL,
EmployeeSalary DECIMAL(12) NOT NULL,
DepartmentID DECIMAL(12) NOT NULL,
DepartmentName VARCHAR(64) NOT NULL,
DepartmentPhone DECIMAL(12) NOT NULL
);
```

```
CREATE TABLE

Query returned successfully in 195 msec.
```

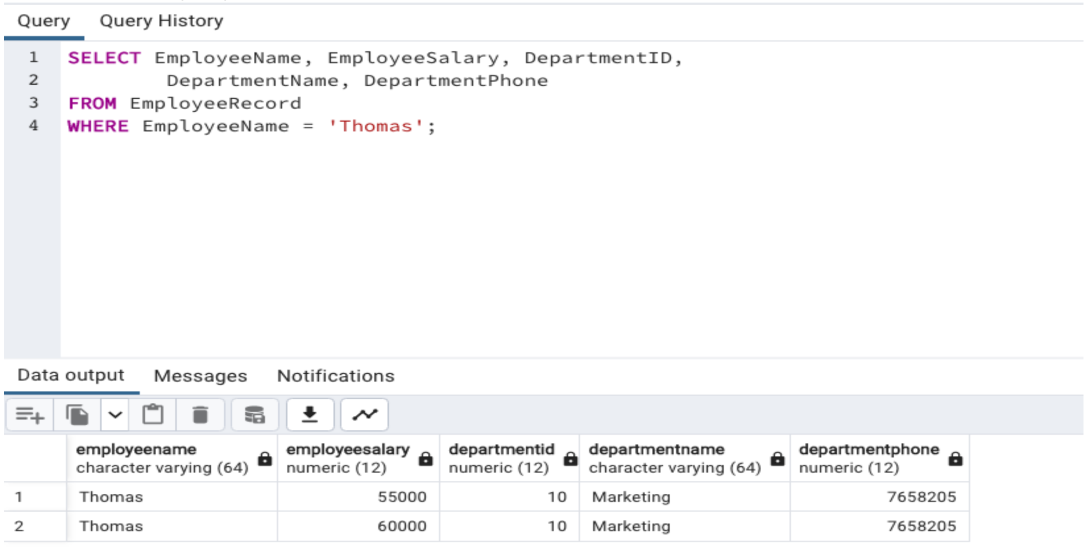
b. **Using the table, demonstrate an anomaly that occurs when the same data is inserted multiple times with different values, and explain what the anomaly means for data integrity.**



```
INSERT INTO Employeerecord(EmployeeID, EmployeeName, EmployeeSalary,
                           DepartmentID, DepartmentName, DepartmentPhone)
VALUES
    (1, 'Ryan', 50000, 10, 'Marketing', 7658205),
    (2, 'Thomas', 55000, 10, 'Marketing', 7658205),
    (3, 'David', 50000, 12, 'Support', 7201520),
    (4, 'John', 55000, 12, 'Support', 7201520),
    (5, 'Harry', 60000, 13, 'Sale', 7125640),
    (6, 'Thomas', 60000, 10, 'Marketing', 7658205);

SELECT * FROM Employeerecord;
```

	employeeid [PK] numeric (12)	employeename character varying (64)	employeesalary numeric (12)	departmentid numeric (12)	departmentname character varying (64)	departmentphone numeric (12)
1	1	Ryan	50000	10	Marketing	7658205
2	2	Thomas	55000	10	Marketing	7658205
3	3	David	50000	12	Support	7201520
4	4	John	55000	12	Support	7201520
5	5	Harry	60000	13	Sale	7125640
6	6	Thomas	60000	10	Marketing	7658205

- Data anomaly occurs when data is already existed is inserted again with some different values. We will have a question about original values and new values whether which of the values is accurate. For example, when we use command:
  SELECT EmployeeName, EmployeeSalary, DepartmentID, DepartmentName, DepartmentPhone
  FROM EmployeeRecord
  WHERE EmployeeName = 'Thomas';



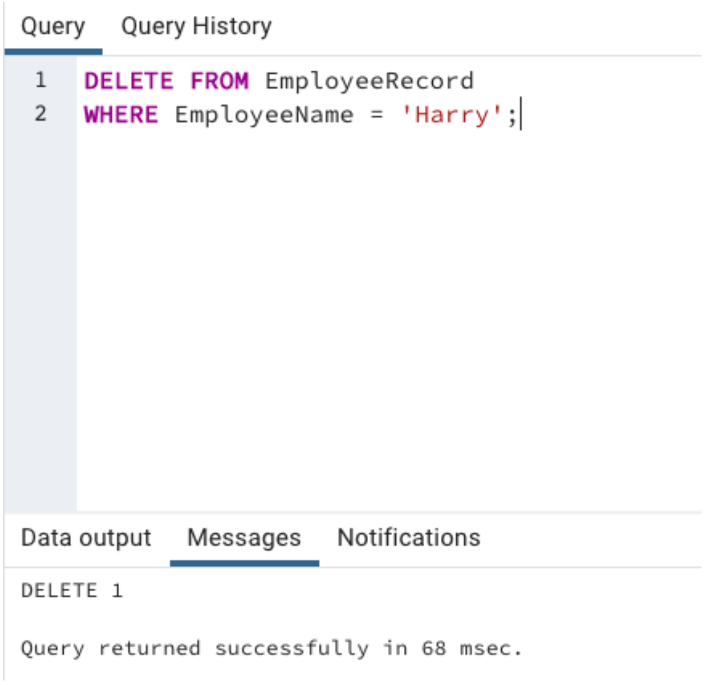
```
SELECT EmployeeName, EmployeeSalary, DepartmentID,
        DepartmentName, DepartmentPhone
FROM EmployeeRecord
WHERE EmployeeName = 'Thomas';
```

	employeename character varying (64)	employeesalary numeric (12)	departmentid numeric (12)	departmentname character varying (64)	departmentphone numeric (12)
1	Thomas	55000	10	Marketing	7658205
2	Thomas	60000	10	Marketing	7658205

- The results will show two different EmployeeSalary values. It makes us confuse which one is accurate because I have two rows for Thomas (Maybe Thomas got salary increase but somehow new record inserted instead update the record), data inconsistencies.

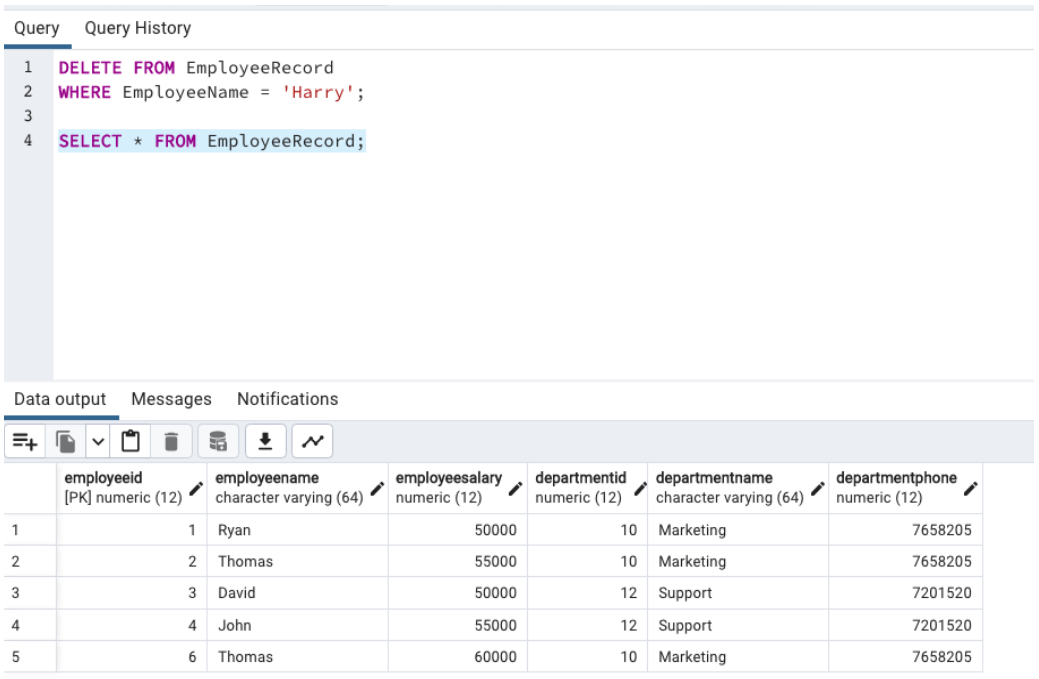
c. Using the table, demonstrate a deletion anomaly with SQL, and explain what the anomaly means for data integrity.

- For example, for some reasons EmployeeName 'Harry' quitted, and we decided to delete his record. The important about this deletion that it also deleted all of other's information such as DepartmentID, DepartmentName, DepartmentPhone because multiple tables such as EmployeeID, EmployeeName, EmployeeSalary, DepartmentID, DepartmentName, DepartmentPhone in this case are combined into a single table, We should have different tables for Employee's information and Department's information.



```
DELETE FROM EmployeeRecord
WHERE EmployeeName = 'Harry';
```

```
DELETE 1

Query returned successfully in 68 msec.
```



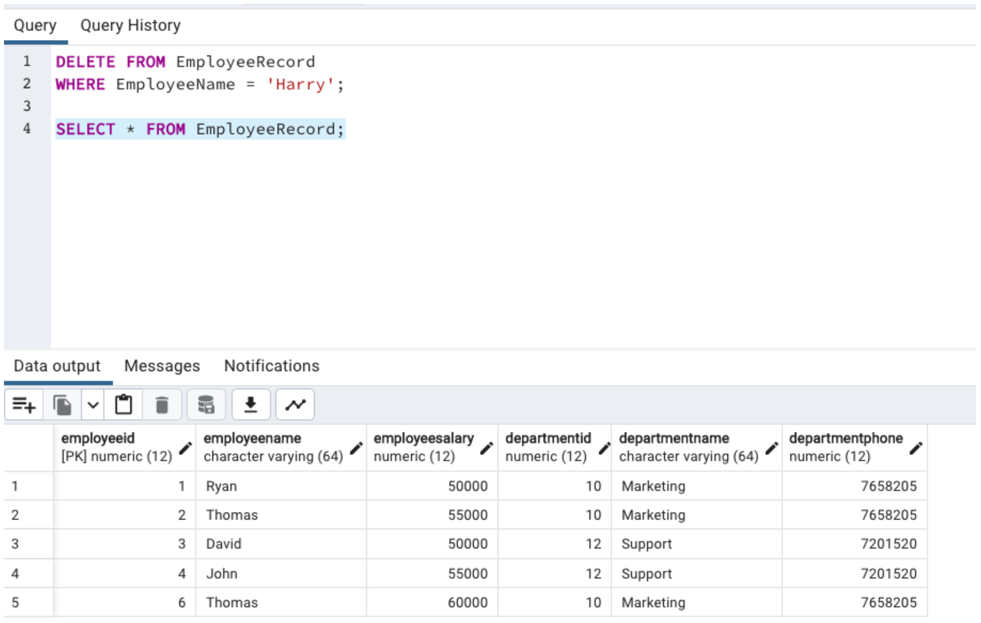
```sql
DELETE FROM EmployeeRecord
WHERE EmployeeName = 'Harry';

SELECT * FROM EmployeeRecord;
```

	employeeid [PK] numeric (12)	employeename character varying (64)	employeesalary numeric (12)	departmentid numeric (12)	departmentname character varying (64)	departmentphone numeric (12)
1	1	Ryan	50000	10	Marketing	7658205
2	2	Thomas	55000	10	Marketing	7658205
3	3	David	50000	12	Support	7201520
4	4	John	55000	12	Support	7201520
5	6	Thomas	60000	10	Marketing	7658205

16. *File and Database Table Comparison* – In this step you compare the table created in #15 with a file that contains all the same information.

a. Create a file in any format you'd like that contains all the same columns and at least 4 rows of information as the table you created in #15. There are many formats you can use. Some examples include XML, flat file, binary, text, and JSON; this list is not exhaustive. All columns and at least 4 rows should be present in the file in its new format. Make sure to provide the file or a screenshot of the file and to explain your choices.

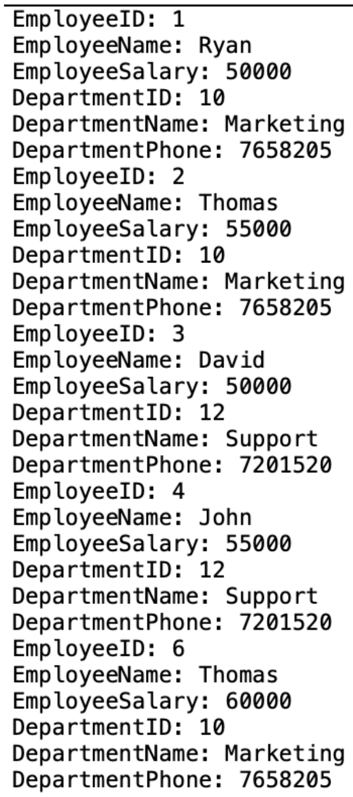
- Table in #15:



```sql
DELETE FROM EmployeeRecord
WHERE EmployeeName = 'Harry';

SELECT * FROM EmployeeRecord;
```

	employeeid [PK] numeric (12)	employeename character varying (64)	employeesalary numeric (12)	departmentid numeric (12)	departmentname character varying (64)	departmentphone numeric (12)
1	1	Ryan	50000	10	Marketing	7658205
2	2	Thomas	55000	10	Marketing	7658205
3	3	David	50000	12	Support	7201520
4	4	John	55000	12	Support	7201520
5	6	Thomas	60000	10	Marketing	7658205

File name is EmployeeRecord.txt. I chose txt file format because it is very easy to read, I does not need complex software to read it and it also very easy to modify and open.



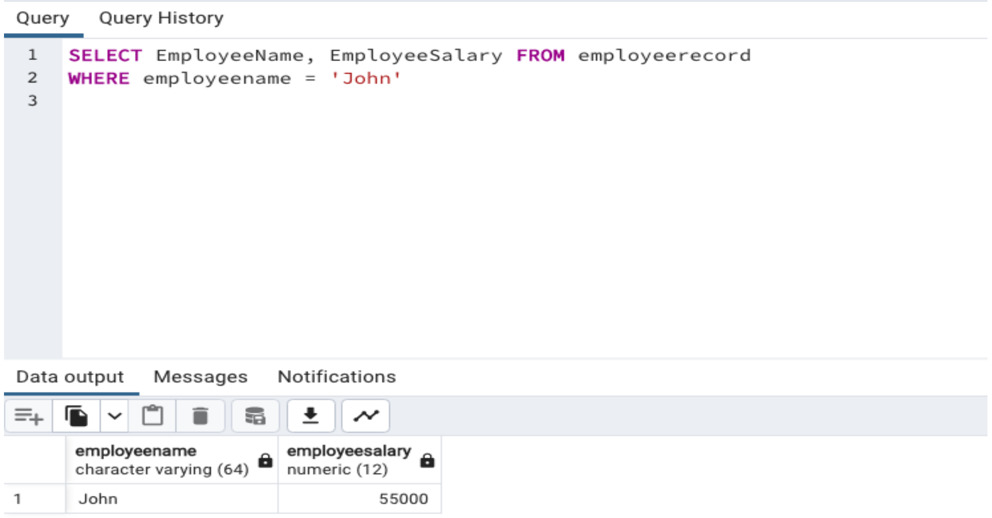
```
EmployeeID: 1
EmployeeName: Ryan
EmployeeSalary: 50000
DepartmentID: 10
DepartmentName: Marketing
DepartmentPhone: 7658205
EmployeeID: 2
EmployeeName: Thomas
EmployeeSalary: 55000
DepartmentID: 10
DepartmentName: Marketing
DepartmentPhone: 7658205
EmployeeID: 3
EmployeeName: David
EmployeeSalary: 50000
DepartmentID: 12
DepartmentName: Support
DepartmentPhone: 7201520
EmployeeID: 4
EmployeeName: John
EmployeeSalary: 55000
DepartmentID: 12
DepartmentName: Support
DepartmentPhone: 7201520
EmployeeID: 6
EmployeeName: Thomas
EmployeeSalary: 60000
DepartmentID: 10
DepartmentName: Marketing
DepartmentPhone: 7658205
```

b. With a few paragraphs, compare what it's like to access data in the table versus in the file. You may need to first research how applications typically access data in this type of file. Make sure to at least use these comparison points:

i. Efficiency – If there were millions of rows of data, would it be more efficient to access a single record in the relational table, or the file, and why?

- Yes. It would be more efficient to access a single record in relational table if there were millions of rows of data because relational table stores the data in rows and relational database can easily support tables with millions of rows in them and allows us to pull our desired record in seconds out of tables. However, file is not likely to be organized in that way, so file takes a lot of time to retrieve a specific single record if there were millions of rows of data.

- In addition, the relational table also can filter the result of the data. For example, we can retrieve specific data of employee' name John from table #15 by using SQL command



```
SELECT EmployeeName, EmployeeSalary FROM employeerecord
WHERE employeename = 'John'
```

	employeename character varying (64)	employeesalary numeric (12)
1	John	55000

ii. **Security – Imagine you needed to restrict access to one specific row/record, allowing only one person to access it, while the rest of the rows could be accessed by many people. Would it be easier or more difficult to secure this row in the relational table compared to the file, and why?**

- Yes. It would be easier to securely restrict access to one specific row/record in relational table compared to the file because file system only support security on the file itself, but not on the data in the file. There is also a chance to change data if someone got granted access into file. However, we can set permission to allow specific people to access to specific records or specific fields within those records in the relational table while file would have to implement its own complex logic to handle it.

iii. **Structural Independence – Imagine the table structure was modified by adding or taking away columns, and equivalent changes were made to the file. Would these changes affect an app using the table differently than an app using the file, and why?**

- Yes. These changes affect an app using the table differently than an app using the file. An app using table will not break when the database changes how it stores the data in its data file, it does not care if we move database from one location to another location or file representation, we still can connect and access all the data in table unless you actually change table file because an app using table does not depend upon the structure of file, it only depends upon of the structure of table itself. In contrast, An app using the file may break if we change the file location, file structures or the order of field, data because an app using file depends upon on location, structures of the file used the represent the data.