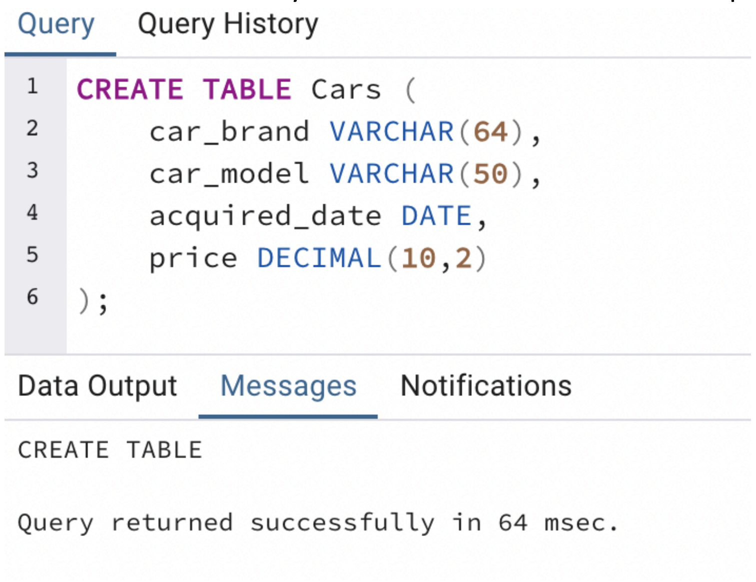
MET CS 669 Database Design and Implementation for Business
Lab 1 Template: SQL From Scratch

Section One – Absolute Fundamentals

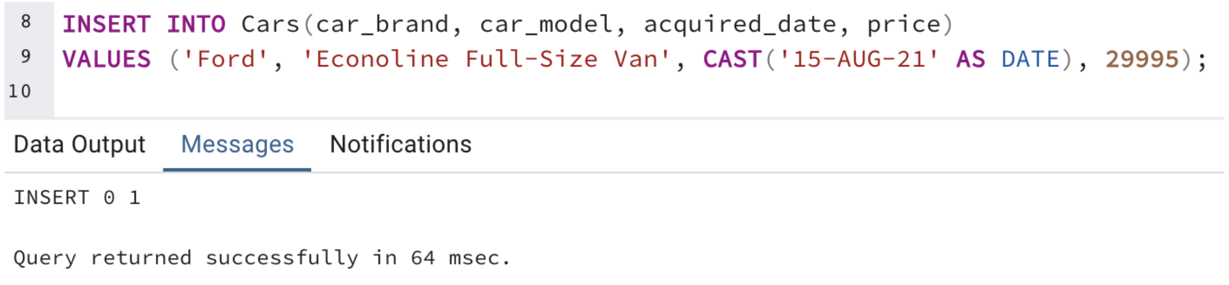
1. *Creating a Table* – Create the Cars table. As a reminder, make sure to follow along in the Lab 1 Explanations document as it shows you how to create tables and complete the other steps.



```sql
CREATE TABLE Cars (
     car_brand VARCHAR(64),
     car_model VARCHAR(50),
     acquired_date DATE,
     price DECIMAL(10,2)
);
```

```
CREATE TABLE

Query returned successfully in 64 msec.
```

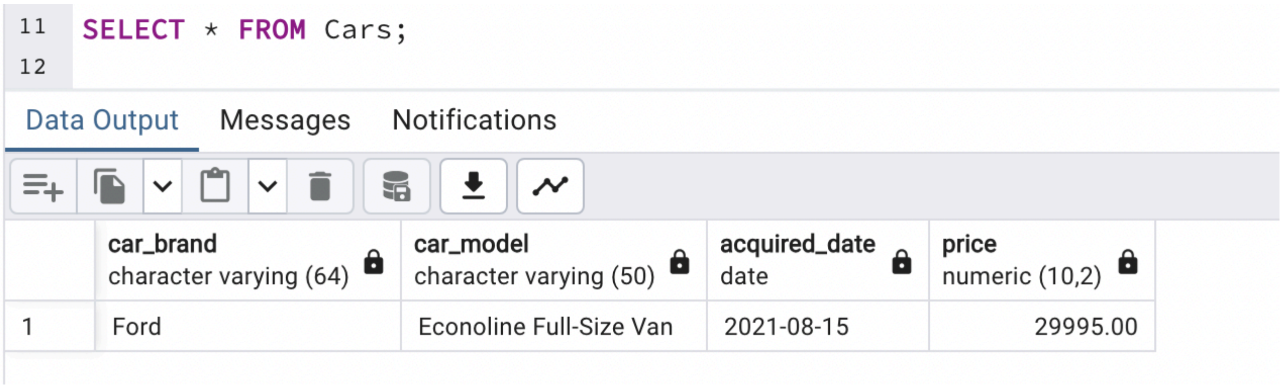
2. *Inserting a Row* – Insert the first row where the Car Brand name is “Ford”, the Car Model is “Econoline Full-Size Van”, the acquisition date for the car is August 15,2021, and the price is $29,995.00.



```sql
INSERT INTO Cars(car_brand, car_model, acquired_date, price)
VALUES ('Ford', 'Econoline Full-Size Van', CAST('15-AUG-21' AS DATE), 29995);
```

```
INSERT 0 1

Query returned successfully in 64 msec.
```

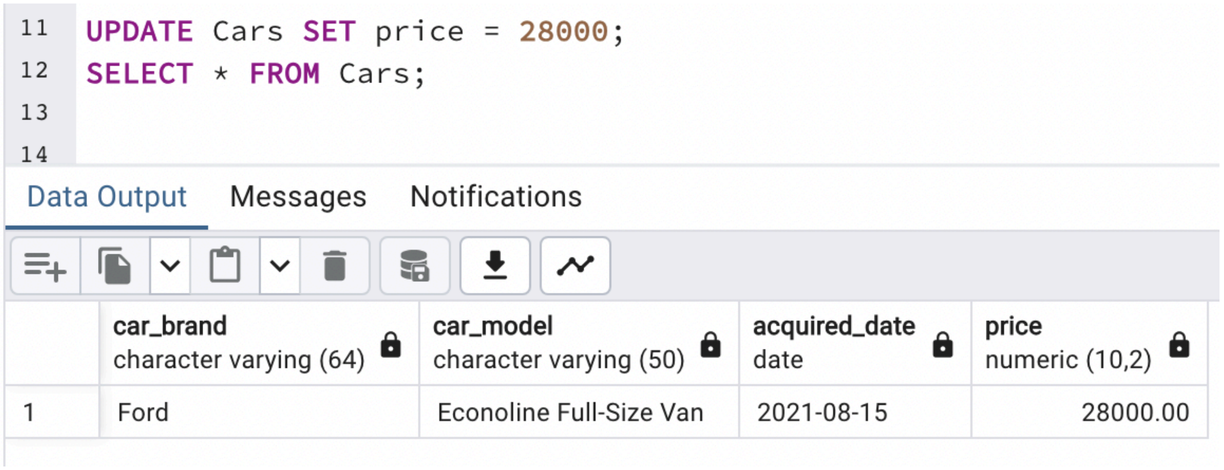
3. *Selecting All Rows* – Select all rows in the table to view the row you inserted.



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

	car_brand character varying (64)	car_model character varying (50)	acquired_date 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.

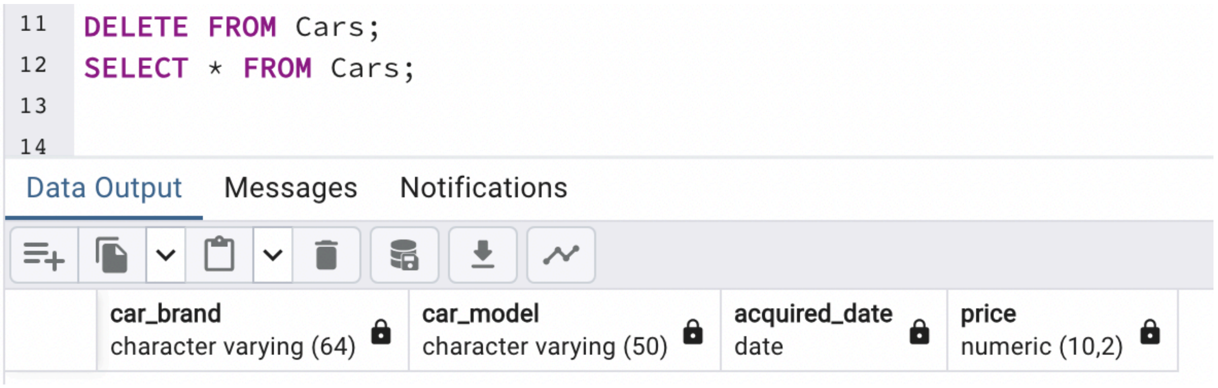


```
UPDATE Cars SET price = 28000;
SELECT * FROM Cars;
```

Data Output | Messages | Notifications

	car_brand character varying (64)	car_model character varying (50)	acquired_date 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.

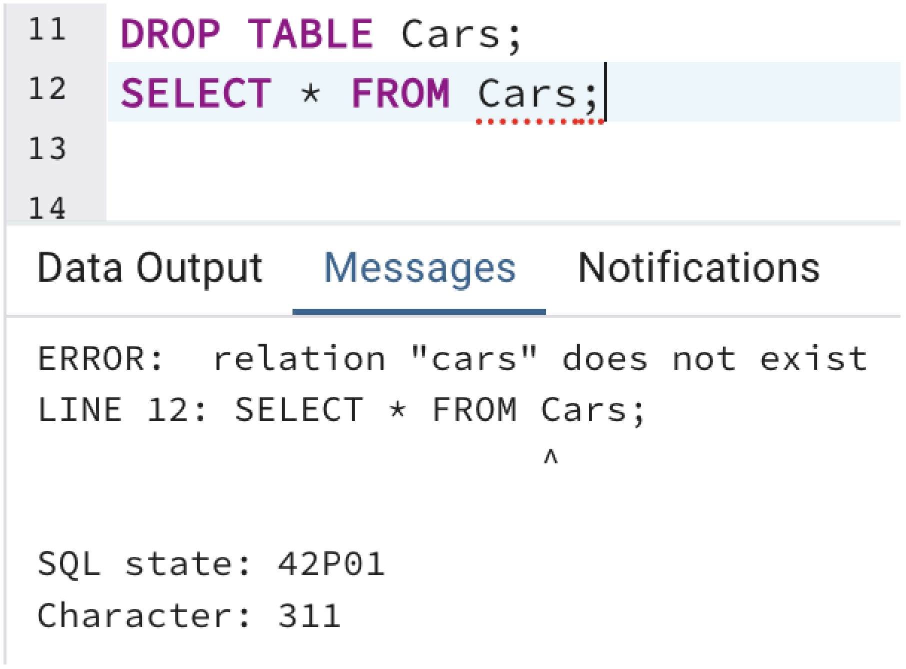


```
DELETE FROM Cars;
SELECT * FROM Cars;
```

Data Output | Messages | Notifications

	car_brand character varying (64)	car_model character varying (50)	acquired_date 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.



```
DROP TABLE Cars;
SELECT * FROM Cars;
```

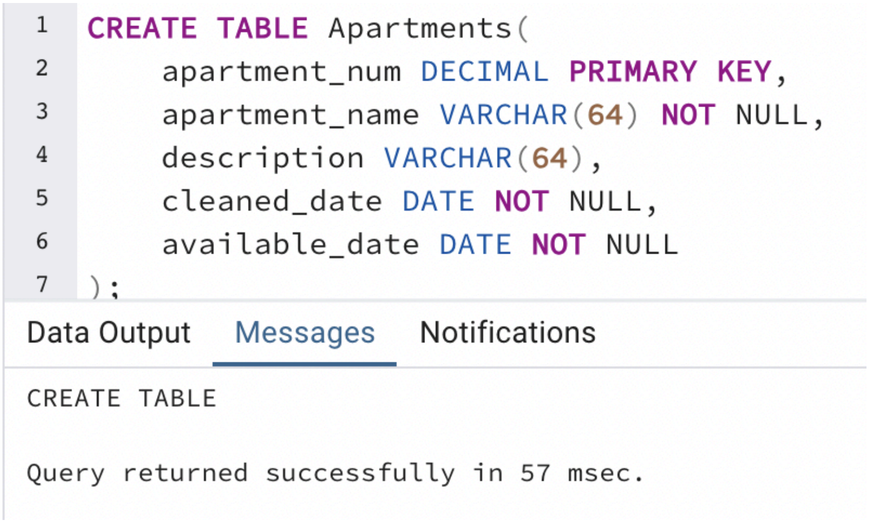
Data Output | Messages | Notifications

```
ERROR:  relation "cars" does not exist
LINE 12: SELECT * FROM Cars;
                       ^


SQL state: 42P01
Character: 311
```

The error message is straightforward in that it says that "cars" does not exist, meaning that the table has been dropped and is no longer available, and shows the exact line in which the Cars table is referenced in the SELECT SQL command to pinpoint the error.

Section Two – More Precise Data Handling

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

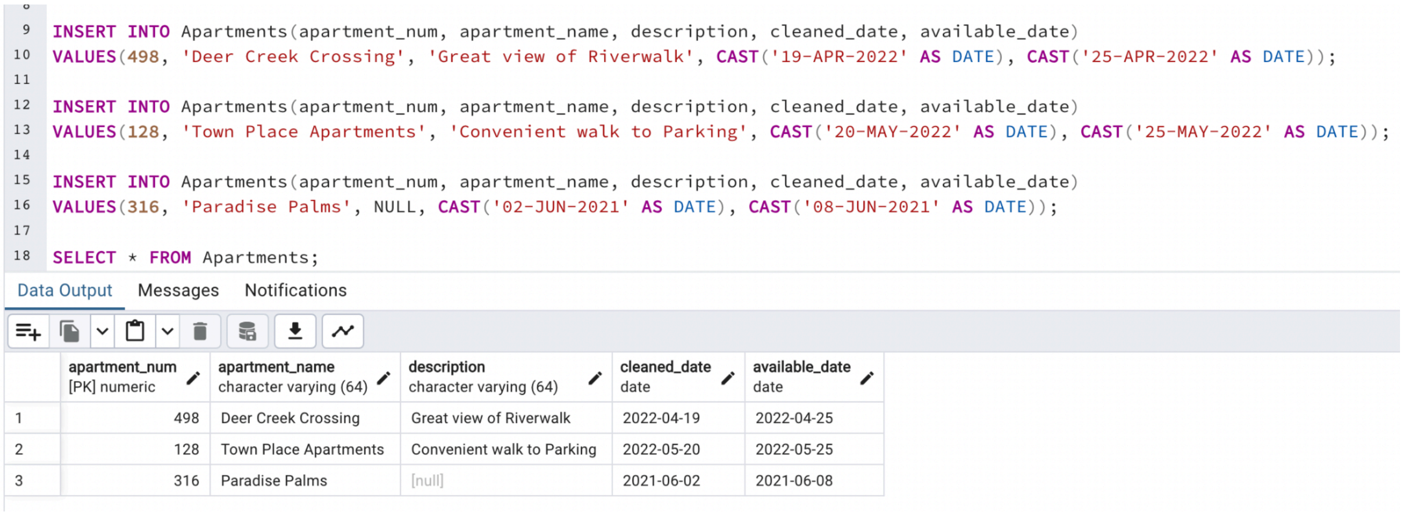


```
CREATE TABLE Apartments(
    apartment_num DECIMAL PRIMARY KEY,
    apartment_name VARCHAR(64) NOT NULL,
    description VARCHAR(64),
    cleaned_date DATE NOT NULL,
    available_date DATE NOT NULL
);
```

Data Output | Messages | Notifications

```
CREATE TABLE

Query returned successfully in 57 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.



```
INSERT INTO Apartments(apartment_num, apartment_name, description, cleaned_date, available_date)
VALUES(498, 'Deer Creek Crossing', 'Great view of Riverwalk', CAST('19-APR-2022' AS DATE), CAST('25-APR-2022' AS DATE));

INSERT INTO Apartments(apartment_num, apartment_name, description, cleaned_date, available_date)
VALUES(128, 'Town Place Apartments', 'Convenient walk to Parking', CAST('20-MAY-2022' AS DATE), CAST('25-MAY-2022' AS DATE));

INSERT INTO Apartments(apartment_num, apartment_name, description, cleaned_date, available_date)
VALUES(316, 'Paradise Palms', NULL, CAST('02-JUN-2021' AS DATE), CAST('08-JUN-2021' AS DATE));

SELECT * FROM Apartments;
```

Data Output | Messages | Notifications

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

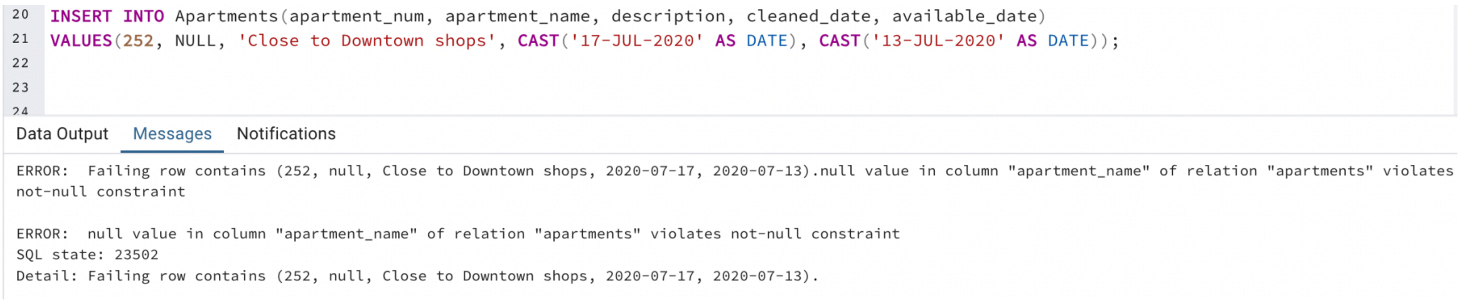
9. *Invalid Insertion* – The following values leave the Apartment Name with no value.

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.
A null value is basically equivalent to not having a value indicated for that column, resulting in essentially an empty field in the table.

b. In your own words, explain what a NOT NULL constraint is.
A NOT NULL constraint is used on a column to make sure every row has a value in that column, and that null values are not allowed to be input into that column.

c. Attempt to insert the values as listed and explain how the database handles this attempt.



```
INSERT INTO Apartments(apartment_num, apartment_name, description, cleaned_date, available_date)
VALUES(252, NULL, 'Close to Downtown shops', CAST('17-JUL-2020' AS DATE), CAST('13-JUL-2020' AS DATE));
```

Data Output | Messages | Notifications

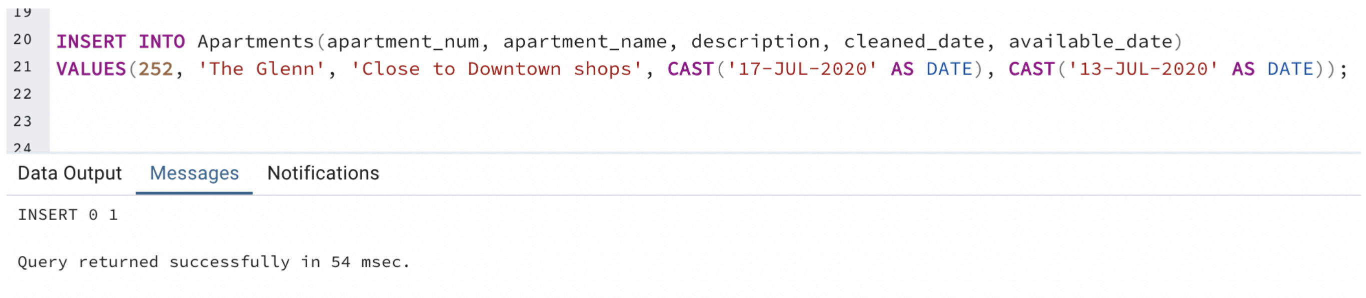
```
ERROR:  Failing row contains (252, null, Close to Downtown shops, 2020-07-17, 2020-07-13).null value in column "apartment_name" of relation "apartments" violates
not-null constraint

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

The database does not cooperate with inserting a null value while the column has a NOT NULL constraint, and it throws an error as a result.

Explain how you would interpret the error message to conclude that the location column is missing a required value.
The error message is straightforward in that it tells us the exact name of the column that is missing the required value and also shows what other values were in the row that we were attempting to insert with the null value.

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

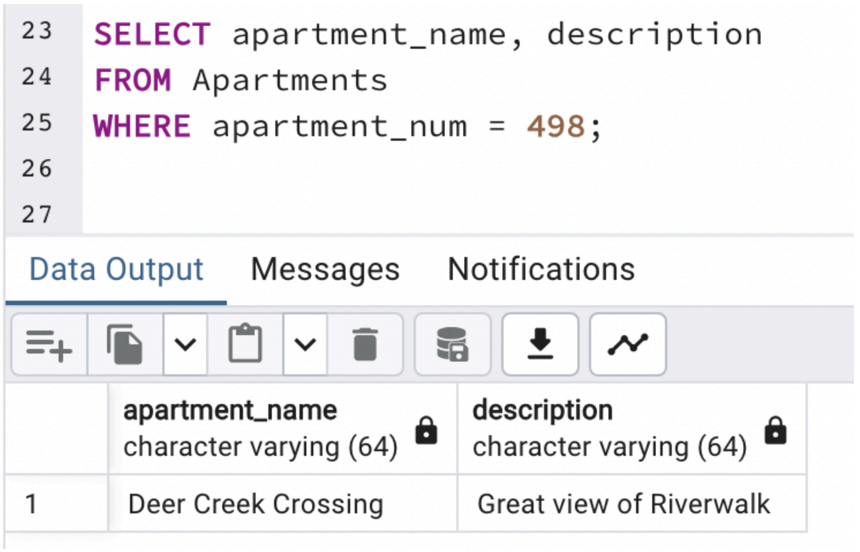


```
INSERT INTO Apartments(apartment_num, apartment_name, description, cleaned_date, available_date)
VALUES(252, 'The Glenn', 'Close to Downtown shops', CAST('17-JUL-2020' AS DATE), CAST('13-JUL-2020' AS DATE));
```

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 54 msec.
```

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.



```
SELECT apartment_name, description
FROM Apartments
WHERE apartment_num = 498;
```

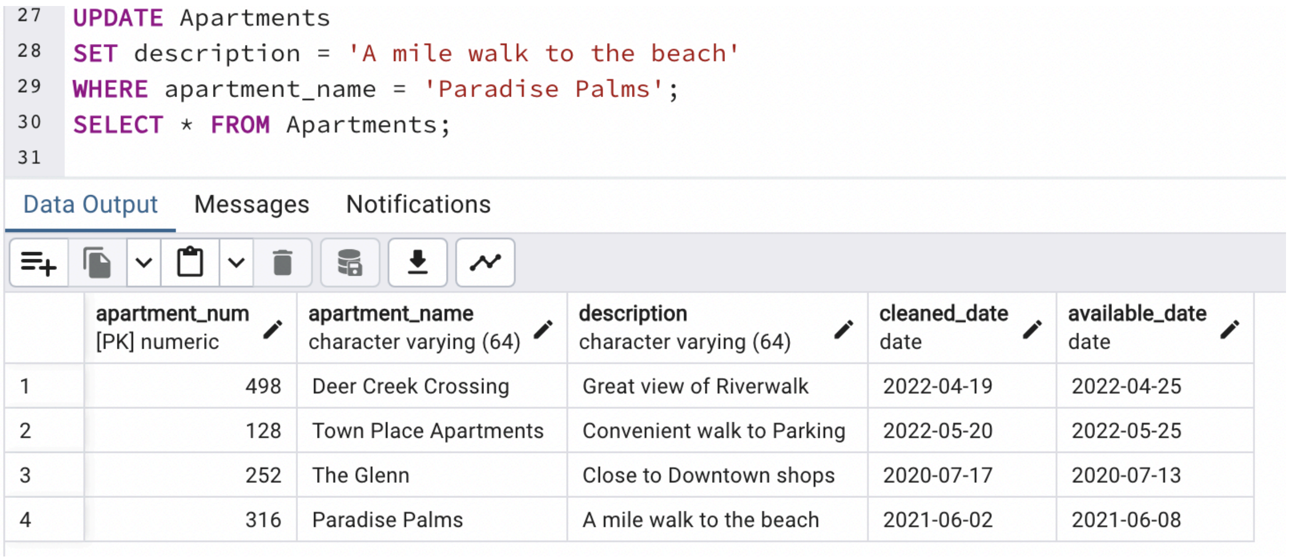
Data Output | Messages | Notifications

	apartment_name 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.
It is useful to limit the rows and columns retrieved by a SELECT statement because when a dataset has hundreds or thousands of rows or columns, limiting the information retrieved is time and resource efficient.

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.



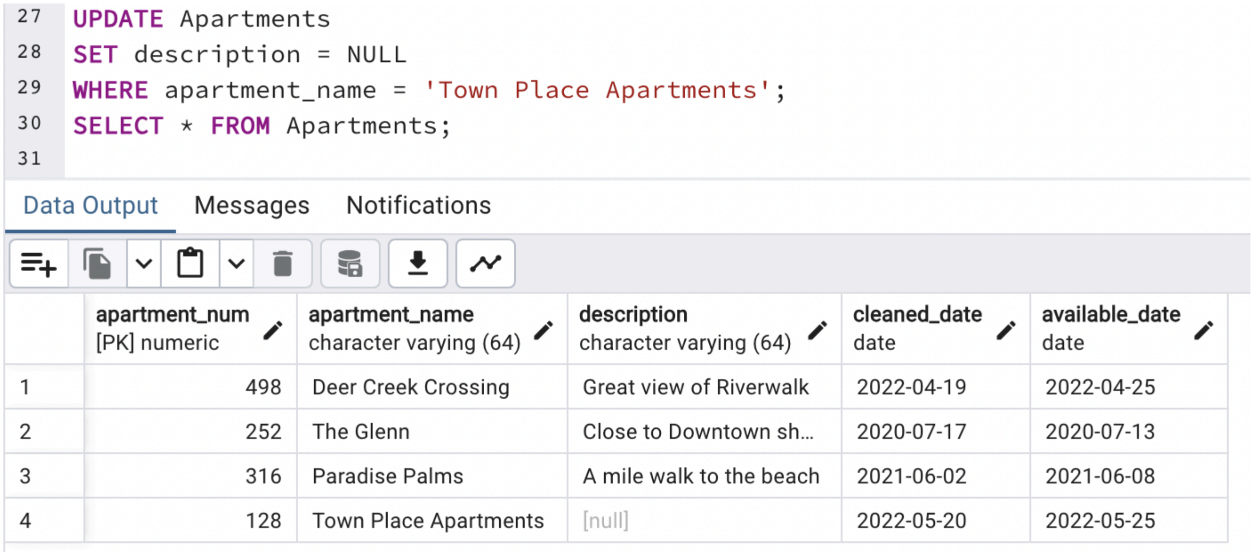
```
UPDATE Apartments
SET description = 'A mile walk to the beach'
WHERE apartment_name = 'Paradise Palms';
SELECT * FROM Apartments;
```

Data Output Messages Notifications

	apartment_num [PK] numeric	apartment_name character varying (64)	description character varying (64)	cleaned_date date	available_date date
1	498	Deer Creek Crossing	Great view of Riverwalk	2022-04-19	2022-04-25
2	128	Town Place Apartments	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.



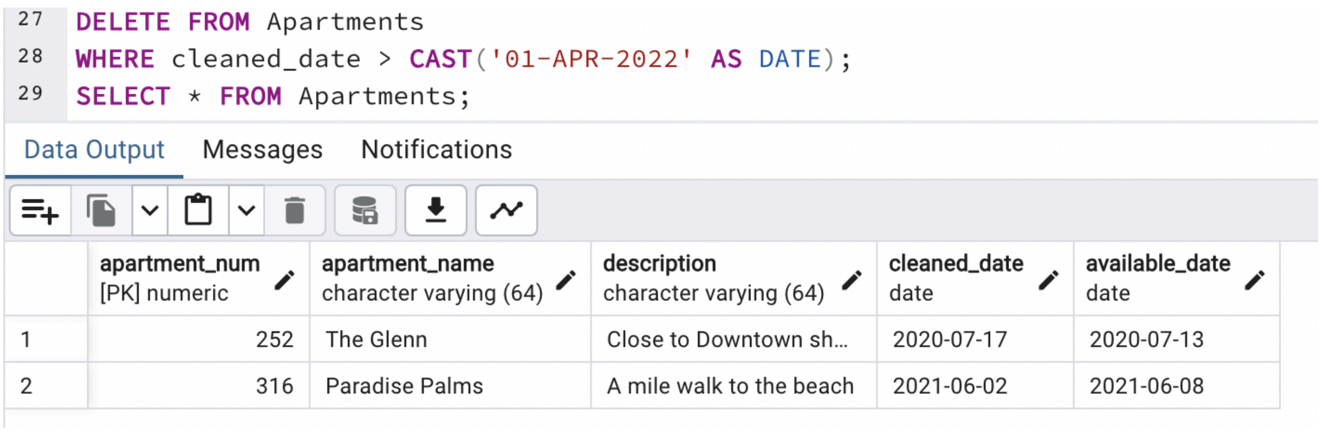
```
UPDATE Apartments
SET description = NULL
WHERE apartment_name = 'Town Place Apartments';
SELECT * FROM Apartments;
```

Data Output Messages Notifications

	apartment_num [PK] numeric	apartment_name character varying (64)	description character varying (64)	cleaned_date date	available_date date
1	498	Deer Creek Crossing	Great view of Riverwalk	2022-04-19	2022-04-25
2	252	The Glenn	Close to Downtown sh...	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 cleaned_date > CAST('01-APR-2022' AS DATE);
SELECT * FROM Apartments;
```

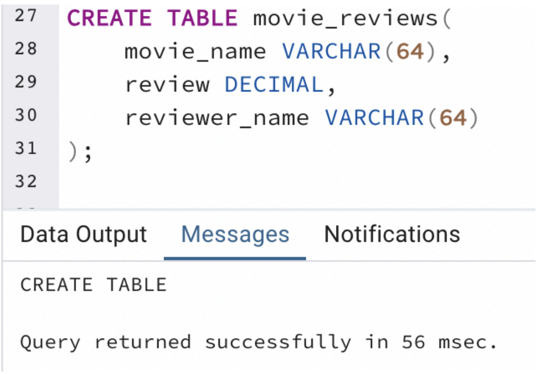
Data Output | Messages | Notifications

	apartment_num [PK] numeric	apartment_name character varying (64)	description character varying (64)	cleaned_date date	available_date date
1	252	The Glenn	Close to Downtown sh...	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

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

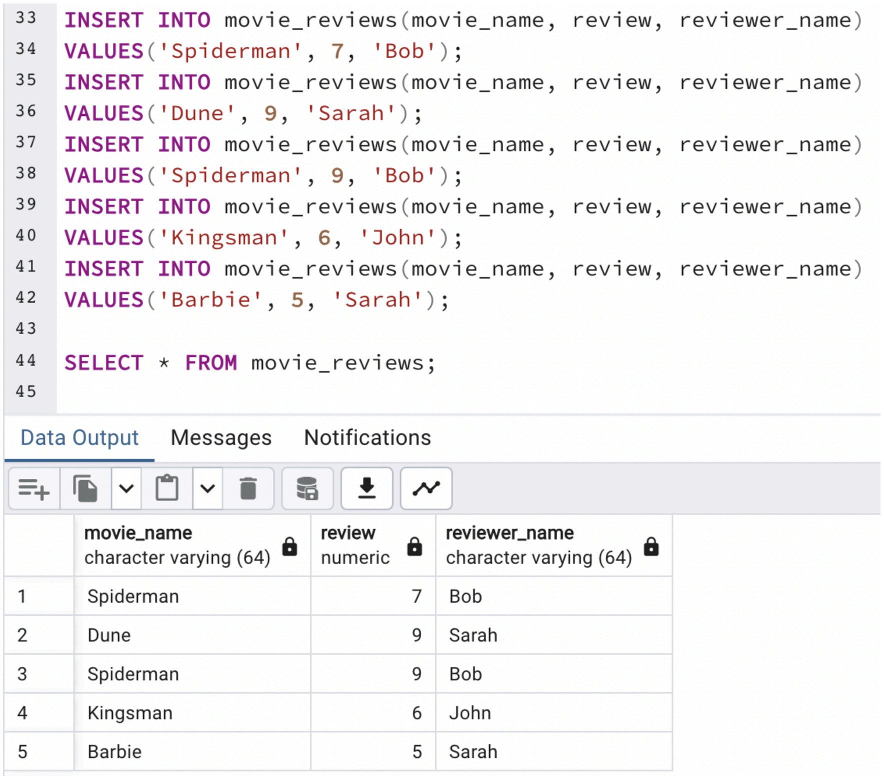
a. Create a table of your choosing that has at least three columns. *Do not add constraints or primary keys to the table, as that may limit your ability to complete parts #b and #c.*


```sql
CREATE TABLE movie_reviews(
    movie_name VARCHAR(64),
    review DECIMAL,
    reviewer_name VARCHAR(64)
);
```

Data Output | Messages | Notifications

```
CREATE TABLE

Query returned successfully in 56 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.

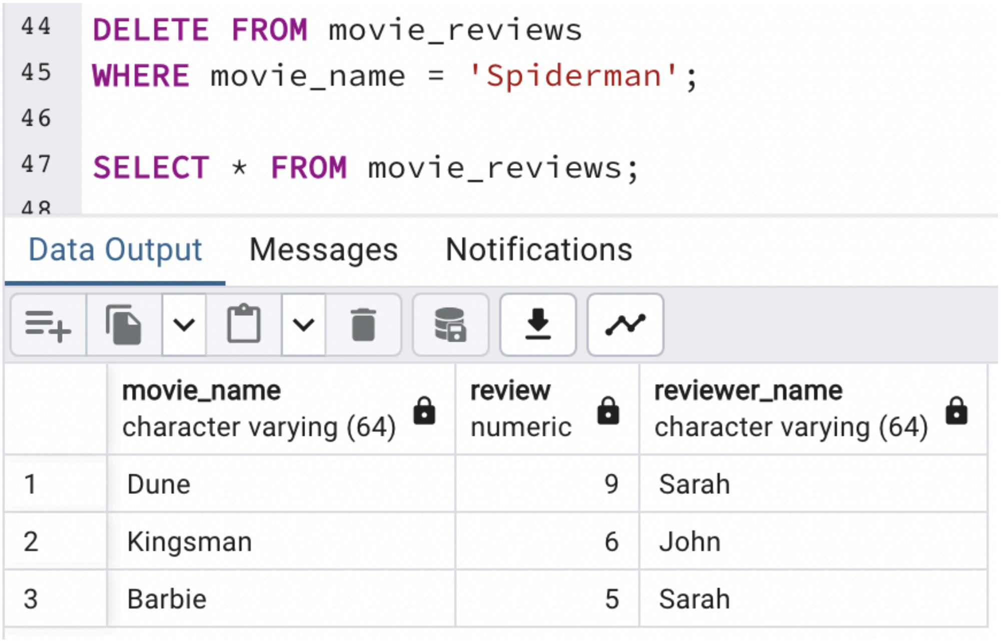

```sql
INSERT INTO movie_reviews(movie_name, review, reviewer_name)
VALUES('Spiderman', 7, 'Bob');
INSERT INTO movie_reviews(movie_name, review, reviewer_name)
VALUES('Dune', 9, 'Sarah');
INSERT INTO movie_reviews(movie_name, review, reviewer_name)
VALUES('Spiderman', 9, 'Bob');
INSERT INTO movie_reviews(movie_name, review, reviewer_name)
VALUES('Kingsman', 6, 'John');
INSERT INTO movie_reviews(movie_name, review, reviewer_name)
VALUES('Barbie', 5, 'Sarah');

SELECT * FROM movie_reviews;
```

Data Output | Messages | Notifications

	movie_name character varying (64)	review numeric	reviewer_name character varying (64)
1	Spiderman	7	Bob
2	Dune	9	Sarah
3	Spiderman	9	Bob
4	Kingsman	6	John
5	Barbie	5	Sarah

In this table, the premise is that movie reviewers are supposed to review movies without inputting multiple contradictory reviews for the same movie. However, this table illustrates an insertion anomaly, with one of the movies having two different ratings from the same reviewer, who watched the movie twice and changed his mind on his initial review. This has a negative impact on data integrity because there is no standardization for the rating for this movie, and now nobody knows which rating is the correct one upon looking at the table.

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



```
DELETE FROM movie_reviews
WHERE movie_name = 'Spiderman';

SELECT * FROM movie_reviews;
```

Data Output Messages Notifications

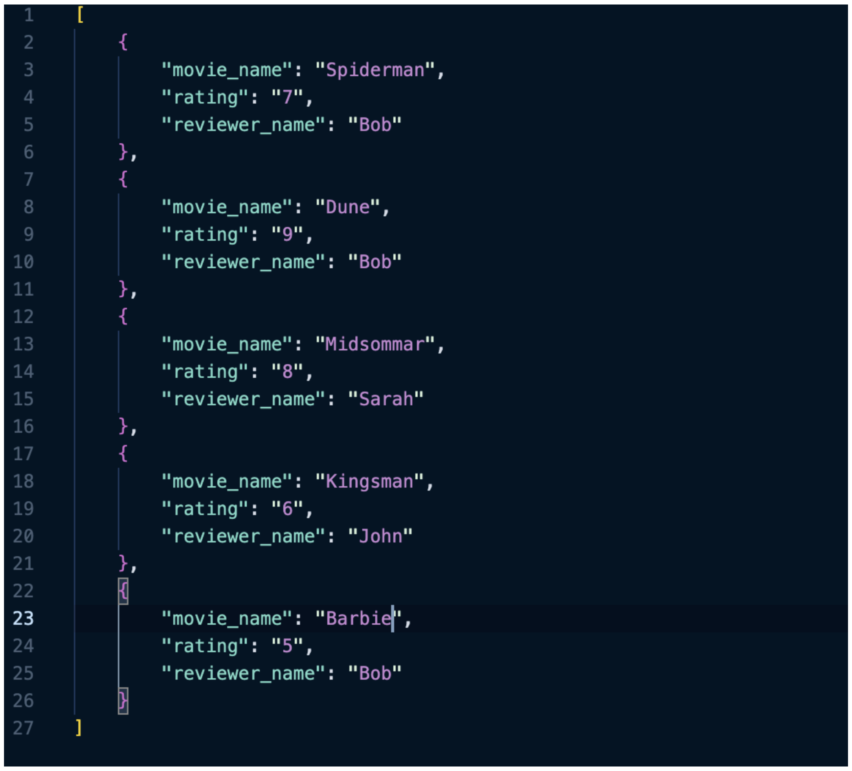
	movie_name character varying (64)	review numeric	reviewer_name character varying (64)
1	Dune	9	Sarah
2	Kingsman	6	John
3	Barbie	5	Sarah

To fix the insertion anomaly posed in the previous section and create consistency, the reviewers decide to delete duplicate reviews. However, in doing so, a deletion anomaly is created because not only are all records of the Spiderman movie reviews deleted, but all records of the reviewer Bob are deleted as well. This negatively affects data integrity because now the reviewers may forget that Spiderman was previously reviewed by their colleague because this information is now permanently deleted.

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

d. 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.

The screenshot below is of the JSON file I chose to create due to my prior experience working with JSON and my appreciation for its ease of use and readability. It is easy to see the different rows and columns when they are represented in JSON because each row is shown within a different pair of brackets.



```json
[
    {
        "movie_name": "Spiderman",
        "rating": "7",
        "reviewer_name": "Bob"
    },
    {
        "movie_name": "Dune",
        "rating": "9",
        "reviewer_name": "Bob"
    },
    {
        "movie_name": "Midsommar",
        "rating": "8",
        "reviewer_name": "Sarah"
    },
    {
        "movie_name": "Kingsman",
        "rating": "6",
        "reviewer_name": "John"
    },
    {
        "movie_name": "Barbie",
        "rating": "5",
        "reviewer_name": "Bob"
    }
]
```

e. 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?
   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?
   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?

Ultimately, it is easier to access data in the table in a variety of contexts compared to the file. It is much more efficient for an application designed to handle a table to pull one record out of millions of rows, than it is for a file handling application to pull a record out of the file. This is due to the nature of relational databases and their ability to support large amounts of data, and this process takes seconds for a table while it may take a much longer amount of time with a file.

Security wise, it would be easier to access just one or a few rows/records in the table compared to the file. File security is all or nothing in that a person can be granted access to the entire file or none of it, while relational databases allow for the granting of access on a row/field level while protecting the rest of the information in the table.

Structurally, accessing data from the table is easier regardless of the manipulation of its columns or rows because applications are not dependent on the structure of the table being consistent and unchanged. However, with the JSON file, applications designed to parse through it and access data are heavily dependent on the syntax of the file being unchanged and being formatted in a very specific way. Additionally, applications that access data in JSON or other file formats need the full file path of the file in question and can run into issues when the file path is changed due to moving the file to different directories. Meanwhile, tables can be moved anywhere within a database and still be accessed as it was previously without issue.