

## **Document Overview**

The purpose of this document is to explain, step-by-step, how to complete Lab 1. The explanations cover the concepts you need learn, and provide plenty of examples. You can think of this document as an assistant that is here to help you complete the lab.

This document is organized by the lab's step numbers. So, if for example, you are working on step 1, you would read through the step 1 section in this document to learn how to complete step 1. The same goes for all of the steps. Read each step's explanation here first, then complete the step.

## Table of Contents

Section One – Absolute Fundamentals .....	3
Step 1 – Creating a Table .....	3
Step 2 – Inserting a Row .....	5
Step 3 – Selecting All Rows .....	7
Step 4 – Updating All Rows .....	8
Step 5 – Deleting All Rows .....	10
Step 6 – Dropping a Table .....	12
Section Two – More Precise Data Handling .....	16
Step 7 – Table Setup .....	16
Step 8 – Table Population .....	19
Step 9 – Invalid Insertion .....	26
Step 10 – Valid Insertion .....	30
Step 11 – Filtered Results.....	31
Step 12 – Updating a Column.....	33
Step 13 – Updating to Null.....	35
Step 14 – Targeted Deletion .....	36
Section Three – Concepts Demonstration .....	38
Step 15 – Data Anomalies .....	38
Step 16 – File and Database Table Comparison .....	42

## Section One – Absolute Fundamentals

### Step 1 – Creating a Table

To show you how to create a table, we'll show you an example of creating a simple Person table, which will look as follows.

Person		
person_id: DECIMAL(12)	first_name: VARCHAR(256)	last_name: VARCHAR(256)
1	John	Smith

This Person table has three columns -- person\_id, first\_name, and last\_name -- with datatypes specified in the diagram above. We'll use the person table through the first section to illustrate the various commands.

First, let's look at the command we use to create the Person table.

```
CREATE TABLE Person(  
  person_id DECIMAL(12),  
  first_name VARCHAR(256),  
  last_name VARCHAR(256)  
);
```

The first thing to note is that the spaces, parentheses, and newlines are all important. These assist the SQL processor in differentiating one word from the next. Removing these can cause errors.

Next, let us look at and understand various aspects of this command. The command begins with the SQL keywords CREATE TABLE. A SQL keyword is a word that the designers of the SQL language chose to carry special meaning within the language. The two keywords CREATE TABLE together indicate to the SQL compiler that we are beginning a command to create a table. The next word, "Person", is the name of the table we are creating. This is not a keyword, but simply an identifier of our choosing. We could have chosen an alternative identifier, limited only to what is relevant and our imagination. SQL compilers know that by definition, the identifier following the CREATE TABLE keywords defines the name of the table. Only certain characters are legal in identifiers, and the legal characters depend upon the particular database we are using. Probably the most common characters used are letters, numbers, and the underscore.

Next let us examine the clause that begins and ends with parentheses. The left parenthesis begins the specification of the columns and constraints for the table, while the right parenthesis closes the same specification. We will learn about constraints next week. For now, we focus on the column specifications. The first thing you may notice is that each column specification is separated by a comma, and the last specification in the list has no comma. We would put fewer specifications if we had fewer columns, and more specifications for more columns.

The next thing you may notice is that each column specification has two words. Although each column specification may have more than two words that define additional aspects of the column, it must have at a minimum both the column's name and the column's datatype. Just as with the table name, the column name identifier is not a SQL keyword, and we can choose a variety of names. A good identifier describes well what it represents. For our first column, we chose "person\_id". The second word in a column specification is a SQL keyword indicating the column's datatype. For our first column, the datatype is "DECIMAL(12)".

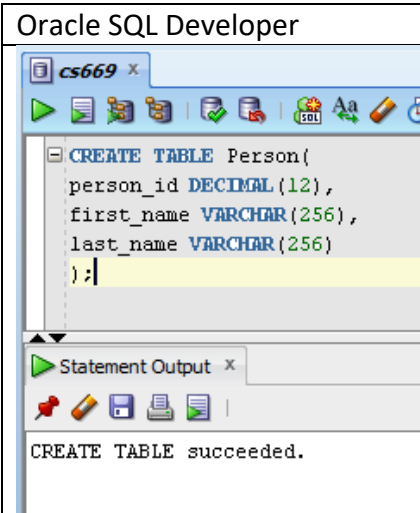
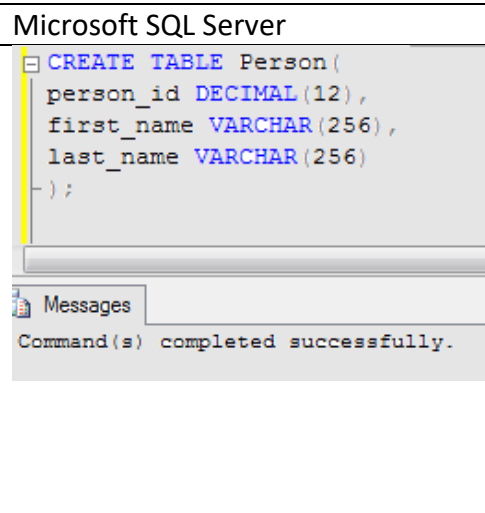
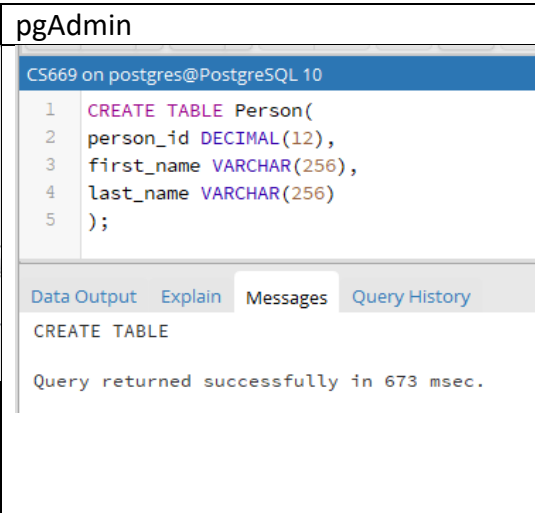
Let us briefly discuss the two datatypes illustrated in our example. A datatype restricts the set of legal values for a column to a particular domain. The DECIMAL datatype in our example indicates that we are restricting the values for person\_id to numbers. The "(12)" after the DECIMAL keyword indicates that we are restricting the numbers further so that there are no decimal points, and a maximum of 12 digits. If we had put "(12, 2)", for example, we would allow two decimal points for a maximum of 12 digits.

The VARCHAR datatype for the first\_name column indicates that we are restricting that column's values to character sequences, for example, "abcd". The "(256)" means that the maximum number of bytes is 256. Some RDBMS allow you to change the meaning of this number from bytes to characters, so that the 256 would mean 256 characters, regardless of the number of bytes per character.

In most modern RDBMS, both spaces and newlines between SQL keywords are categorized simply as "whitespace", and are treated identically by the DBMS. This abstraction means that one could use spaces in lieu of newlines, thereby containing the SQL command to a single line. However, a common convention, as illustrated above, uses newlines to increase the readability and clarity of the SQL command.

In most modern RDBMS including Oracle, Microsoft SQL Server and PostgreSQL, SQL keywords are *not* case-sensitive, meaning that the choice to use an uppercase or lowercase character does not matter to the DBMS. A common convention, as illustrated above, is to capitalize SQL keywords, capitalize the first letter of each word for the table name (known as camel case), and use lowercase for other items.

Below are screenshots of this command being executed in the three supported databases.

Oracle SQL Developer	Microsoft SQL Server	pgAdmin
 The screenshot shows the Oracle SQL Developer interface. At the top, a tab labeled 'cs669 x' is open. Below it, a toolbar contains various icons. The main editor area displays the SQL command: <pre>CREATE TABLE Person(   person_id DECIMAL(12),   first_name VARCHAR(256),   last_name VARCHAR(256) );</pre> Below the editor is a 'Statement Output' window showing the message: 'CREATE TABLE succeeded.'	 The screenshot shows the Microsoft SQL Server Enterprise Manager interface. The 'SQL Server Enterprise Manager' tree is on the left. The main area shows the SQL command: <pre>CREATE TABLE Person(   person_id DECIMAL(12),   first_name VARCHAR(256),   last_name VARCHAR(256) );</pre> Below the command, a 'Messages' window displays the message: 'Command(s) completed successfully.'	 The screenshot shows the pgAdmin interface. At the top, a tab labeled 'CS669 on postgres@PostgreSQL 10' is open. Below it, a toolbar contains various icons. The main editor area displays the SQL command: <pre>1 CREATE TABLE Person( 2   person_id DECIMAL(12), 3   first_name VARCHAR(256), 4   last_name VARCHAR(256) 5 );</pre> Below the editor is a 'Query History' window showing the message: 'Query returned successfully in 673 msec.'

Notice that the screenshots contain the SQL command and the result of the SQL command, and only captures the relevant portion of the screen. You should do the same.

One datatype the Movie table includes, but the Person table does not, is DATE. The DATE datatype lets us track calendar dates, as its name suggests.

## Step 2 – Inserting a Row

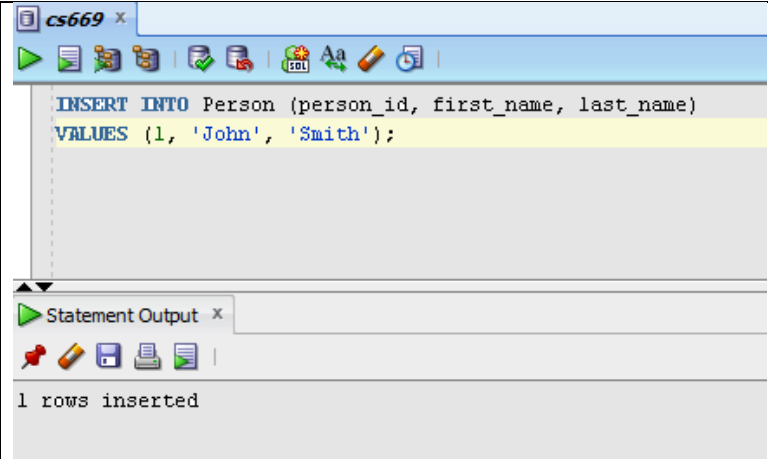
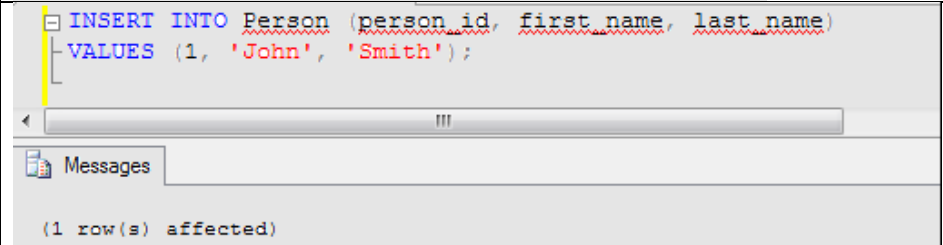
Adding data to a table is more technically named an “insert” in SQL. Following with our Person example, here is the command to insert a row with values `person_id = 1`, `first_name = John`, and `last_name = Smith` by executing the following command.

```
INSERT INTO Person (person_id, first_name, last_name)
VALUES (1, 'John', 'Smith');
```

The comma separated list on the first line indicates the names of the columns, and the comma separated list on the second line indicates the values to insert into those columns, respectively. The value 1 is inserted into the `person_id` column, the value “John” is inserted into the `first_name` column, and the value “Smith” is inserted into the `last_name` column. Though it is possible to omit the first comma separated list by inserting the values in the order they exist in the database, it is not recommended to do so. Production strength SQL insertions specify the column names as illustrated above, to help prevent several cases where data is unknowingly inserted into the wrong column.

Numeric values can be typed without apostrophes. Character based values such as a `first_name` and `last_name` must be quoted with an apostrophe ('). These apostrophes tell the RDBMS where the character sequence begins and ends. One reason apostrophes are necessary is that character sequences can contain spaces, and the RDBMS needs a way to know when a space is a separator for a token in the SQL language, and when a space is part of a single sequence of characters for a value. Make sure to use the regular apostrophe ('), and not the typographer's apostrophe (').

Below is a screenshot of the command and the result of its execution.

Oracle SQL Developer	 <p>The screenshot shows the Oracle SQL Developer interface. The top pane displays the SQL command: <code>INSERT INTO Person (person_id, first_name, last_name) VALUES (1, 'John', 'Smith');</code>. The bottom pane, titled "Statement Output", shows the result: "1 rows inserted".</p>
Microsoft SQL Server Management Studio	 <p>The screenshot shows the Microsoft SQL Server Management Studio interface. The top pane displays the SQL command: <code>INSERT INTO Person (person_id, first_name, last_name) VALUES (1, 'John', 'Smith');</code>. The bottom pane, titled "Messages", shows the result: "(1 row(s) affected)".</p>

pgAdmin

CS669 on postgres@PostgreSQL 10

6

7

8

9

10

11

INSERT INTO Person (person\_id, first\_name, last\_name)

VALUES (1, 'John', 'Smith');

Data Output

Explain

Messages

Query History

INSERT 0 1

Query returned successfully in 438 msec.

Since the person table does not include a DATE datatype, here is the syntax of how you can hardcode a particular date:

CAST('01-MAR-2019' AS DATE)

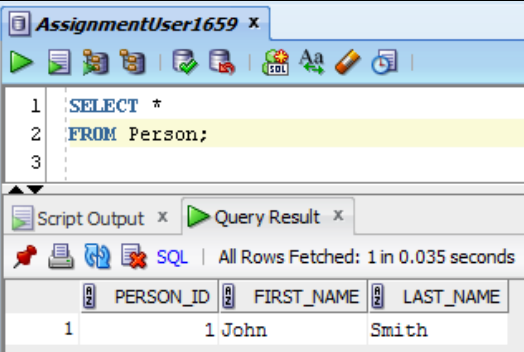
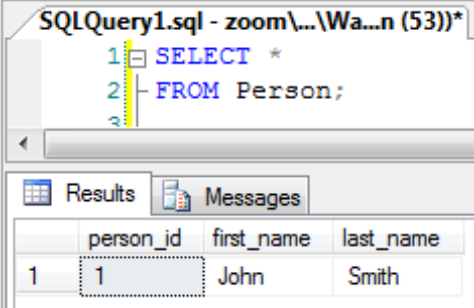
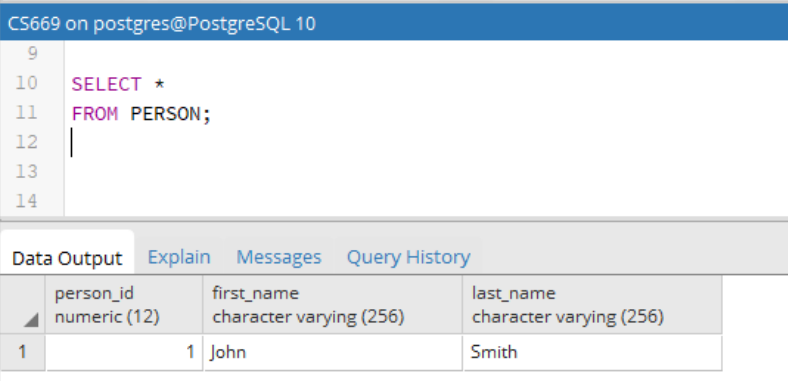
This example hardcodes 3/1/2019 as a date. The day number comes first, followed by a month abbreviation, followed by the year.

## Step 3 – Selecting All Rows

It's not too complex to retrieve all rows from a table. The technical word for retrieving data is "select" in SQL. As an example, we can select all rows from the Person table by executing the following command.

```
SELECT *  
FROM Person;
```

The first line instructs the RDBMS to retrieve all columns, because the "\*" value indicates "all". The second line instructs the RDBMS to retrieve from the Person table. A screenshot of the command and the result of its execution is below.

Oracle SQL Developer	
Microsoft SQL Server Management Studio	
pgAdmin	

Notice that we see the row we just inserted in the result set.

## Step 4 – Updating All Rows

Now you have a chance to update existing data. Here is an example of how we update all rows in the Person table. Since there is only one row it has the effect of updating the last name of John Smith from “Smith” to “Glass”.

```
UPDATE Person
SET last_name = 'Glass';
```

The first word “UPDATE” indicates to the RDBMS that we are updating row(s) in a table. The second word “Person” indicates that we are updating row(s) in the Person table. The SET keyword on the second line begins a comma-separated list of column names and their new values. By using the phrase:

```
last_name = 'Glass'
```

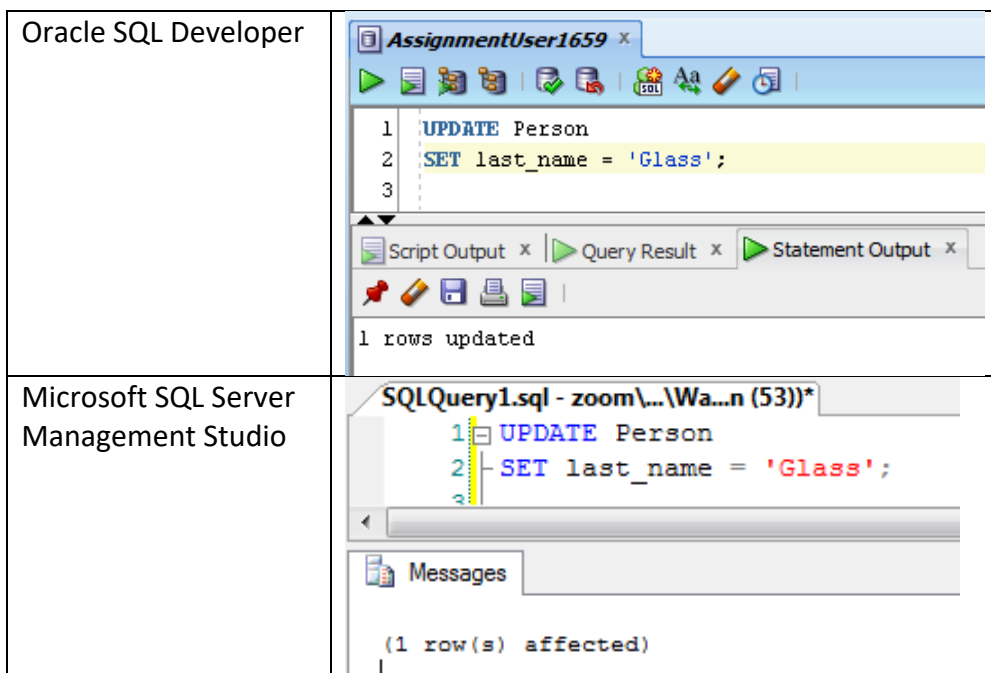
we are instructing the RDBMS to set the value of the last\_name column to the value “Glass”. If we had instead used the phrase:

```
first_name = 'Jane'
```

for example, we would be instructing the RDBMS to set the first\_name value to “Jane”.

The UPDATE command above updates all rows in the Person table. If there were to be more than one row in the Person table, the last\_name values for *all* of the rows would be updated. We will learn in the next section how to limit updates to a specific row or group of rows.

Here is a screenshot of the command the result of its execution.





pgAdmin

CS669 on postgres@PostgreSQL 10

5

6 UPDATE Person

7 SET last\_name = 'Glass';

8

9

Data Output Explain Messages Query History

UPDATE 1

Query returned successfully in 498 msec.

We view the results of our update by selecting all rows in the table.

```
SELECT *
FROM Person;
```

In the screenshots below, you’ll notice the last name is now “Glass”.

Oracle SQL Developer

AssignmentUser1659 x

1 SELECT \*

2 FROM Person;

3

Script Output x Statement Output x Query Result x

SQL | All Rows Fetched: 1 in 0 seconds

	PERSON_ID	FIRST_NAME	LAST_NAME
1	1	John	Glass

Microsoft SQL Server Management Studio

SQLQuery1.sql - zoom\...\Wa...n (53))\*

1 SELECT \*

2 FROM Person;

3

Results Messages

	person_id	first_name	last_name
1	1	John	Glass

pgAdmin

CS669 on postgres@PostgreSQL 10

1 SELECT \*

2 FROM PERSON;

3

4

5

Data Output Explain Messages Query History

	person_id	first_name	last_name
1	1	John	Glass

## Step 5 – Deleting All Rows

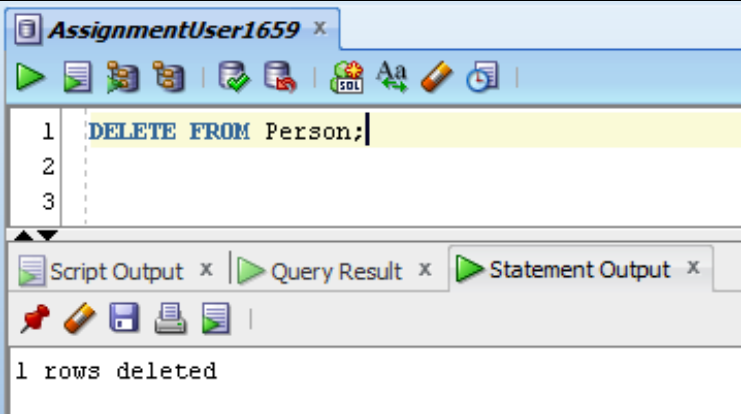
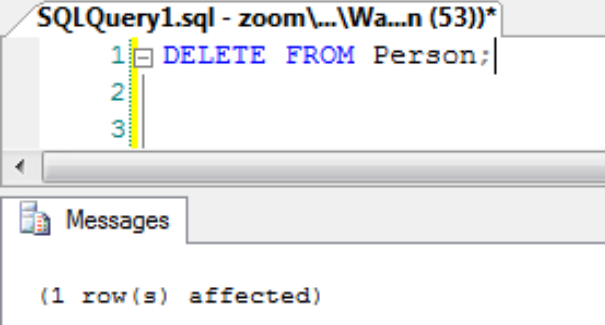
To remove all rows from a table, you use the DELETE command. Here is an example of doing so with the Person table.

```
DELETE FROM Person;
```

The DELETE FROM keywords indicate to the RDBMS that we are deleting one or more rows from a table. The next word, “Person”, indicates that the table is the Person table.

Executing the command above will delete *all* rows in the Person table. In our example we only have one row, but tables in production environments usually have many rows. We will learn in the next section how to limit a delete to a specific row or group of rows.

Here is a screenshot of executing this command.

Oracle SQL Developer	 <p>The screenshot shows the Oracle SQL Developer interface. The title bar indicates the user is 'AssignmentUser1659'. The main editor window contains the SQL command 'DELETE FROM Person;' on line 1. Below the editor, there are tabs for 'Script Output', 'Query Result', and 'Statement Output'. The 'Statement Output' tab is active, displaying the message '1 rows deleted'.</p>
Microsoft SQL Server Management Studio	 <p>The screenshot shows the Microsoft SQL Server Management Studio interface. The title bar indicates the file is 'SQLQuery1.sql - zoom\...\Wa...n (53))'. The main editor window contains the SQL command 'DELETE FROM Person;' on line 1. Below the editor, there is a 'Messages' pane displaying the message '(1 row(s) affected)'.</p>

pgAdmin

CS669 on postgres@PostgreSQL 10

8

9

10

11

12

DELETE FROM Person;

Data Output

Explain

Messages

Query History

DELETE 1

Query returned successfully in 264 msec.

Notice that each RDBMS indicate the number of rows that were deleted.

In order to see that the table now has no rows, let us again select all rows in the table.

Oracle SQL Developer

AssignmentUser1659 x

1

2

3

SELECT \*  
FROM Person;

Script Output x

Statement Output x

Query... x

SQL | All Rows Fetched: 0 in 0.001 seconds

PERSON... FIRST\_... LAST\_N...

Microsoft SQL Server Management Studio

SQLQuery1.sql - zoom\...\Wa...n (53))\*

1

2

3

SELECT \* FROM Person;

Results

Messages

person\_id

first\_name

last\_name

pgAdmin

CS669 on postgres@PostgreSQL 10

1

2

3

4

5

SELECT \*  
FROM PERSON;

Data Output

Explain

Messages

Query History

person\_id

first\_name

last\_name

numeric (12)

character varying (256)

character varying (256)

Notice that although the columns still appear in the result set, no rows of data appear. This is because all rows have been removed from the Person table.

## Step 6 – Dropping a Table

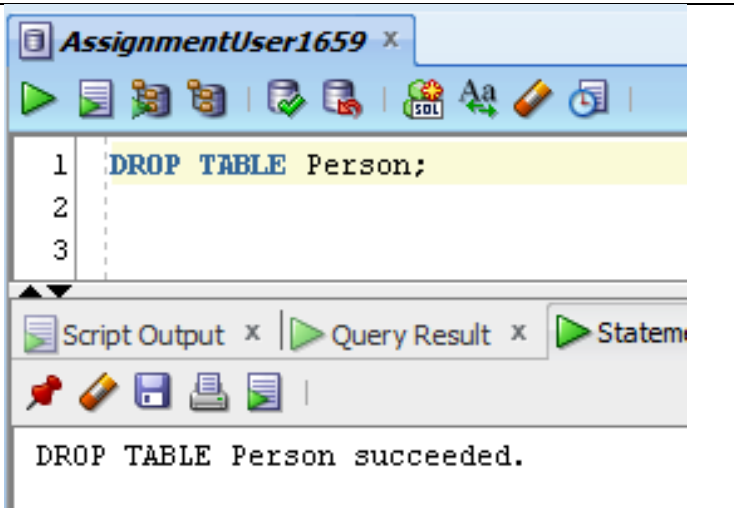
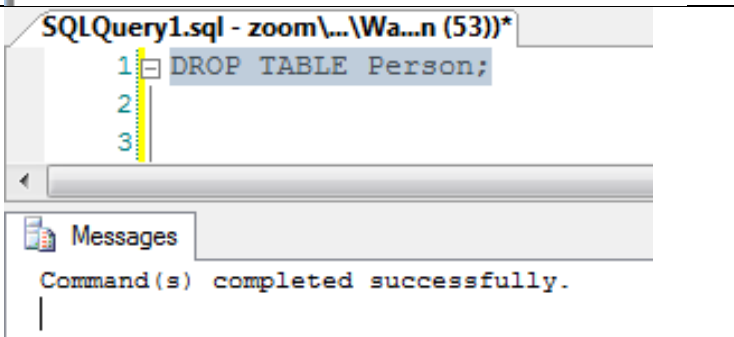
The command for getting rid of a table is DROP. To drop the Person table, we would use this command.

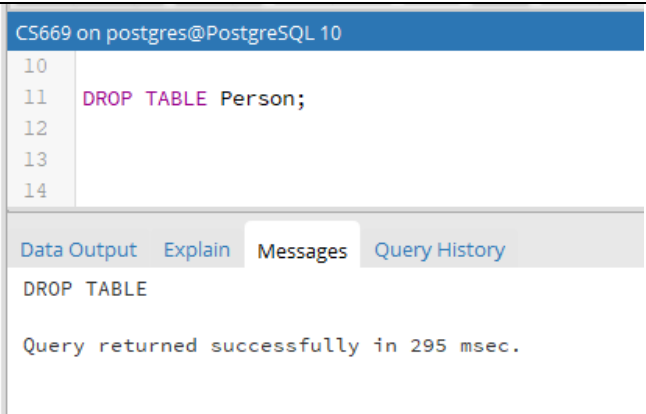
```
DROP TABLE Person;
```

The first two keywords DROP TABLE instruct the RDBMS to remove a table. The third word, “Person”, is the name of the table to be dropped, in this case, the Person table.

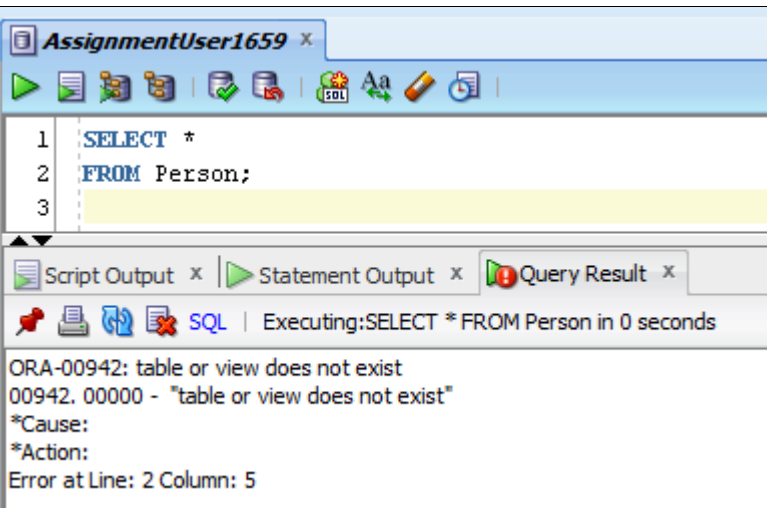
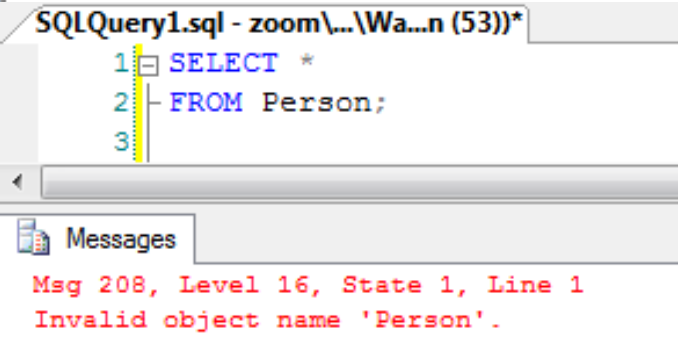
Although in this exercise we are casually dropping the Person table, in production environments command should be used with extreme care. All of the data in the table will also be removed, and the data cannot always be recovered if you later decide that you want to keep the data.

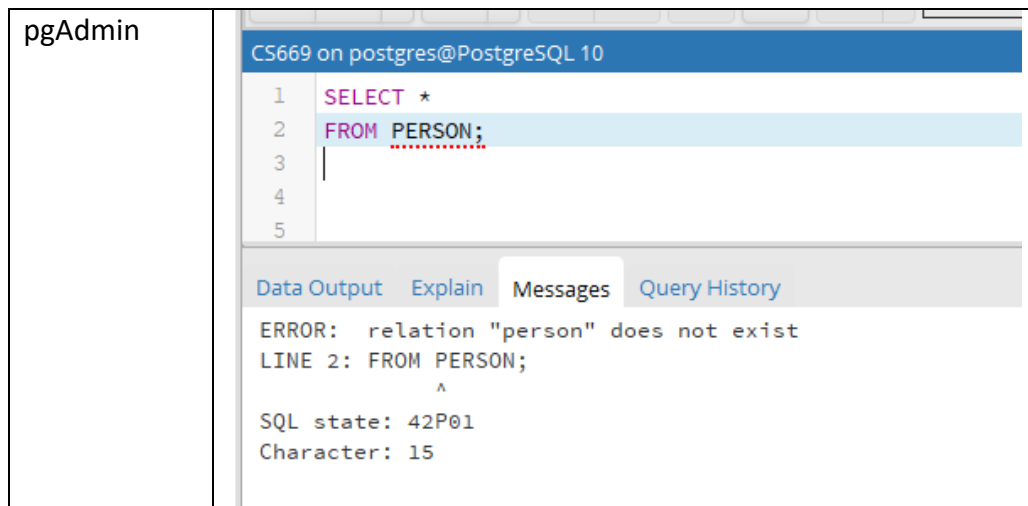
Below is a screenshot of dropping the Person table.

Oracle SQL Developer	
Microsoft SQL Server Management Studio	

pgAdmin	 <p>A screenshot of the pgAdmin interface. The top bar shows 'C5669 on postgres@PostgreSQL 10'. The SQL editor contains the command 'DROP TABLE Person;' on line 11. Below the editor, the 'Messages' tab is active, displaying 'DROP TABLE' and 'Query returned successfully in 295 msec.'</p>
---------	--

For good measure, let us attempt to retrieve data from the Person table we just deleted. Of course, now that the Person table has been dropped, the command will not be able to retrieve the results. It is helpful however to become accustomed to the error message indicating that the table does not exist.

Oracle SQL Developer	 <p>A screenshot of the Oracle SQL Developer interface. The SQL editor shows 'SELECT * FROM Person;'. The 'Query Result' tab is active, displaying an error: 'ORA-00942: table or view does not exist' with details on cause and action. The error occurred at Line 2, Column 5.</p>
Microsoft SQL Server Management Studio	 <p>A screenshot of the Microsoft SQL Server Management Studio interface. The SQL editor shows 'SELECT * FROM Person;'. The 'Messages' tab is active, displaying an error: 'Msg 208, Level 16, State 1, Line 1 Invalid object name 'Person'.'</p>



### ***Oracle Explanation***

The Oracle error message clearly states that the table or view being referenced in the command does not exist. The reason for the additional clause “or view” in the error message is that wherever a table is used in a command, a view can be used as well. We will study views in future weeks.

To diagnose the issue, the error message and SQL command need to be analyzed together. Either alone does not provide sufficient information. For example, in this case, the error message does not state that it was the reference to “Person” that was invalid. We must deduce that by reviewing our SQL command. The error message has one more piece of information that will help us do so, which is a line number and a column number. In this case, the error message is stating that the source of the error begins at line 2, column 5. If we review our command, we see that the word “Person” begins at line 2, column 5, and so that is the source of the error. The fact that Person does not exist is expected, since we previously dropped that table.

### ***Microsoft SQL Server Explanation***

The error message generated by SQL Server may be somewhat difficult to interpret from a relational point of view, because it does not use the familiar term “table” in the error message. SQL Server uses the generic term “object” to denote any durable entity that it supports, including tables and views. Therefore we can interpret the phrase “Invalid object name ‘Person’” to mean that we attempted to use the name “Person” as a reference to a durable entity in the SQL command, but no durable entity exists by that name.

Just as with the Oracle error messages, we need to use the SQL Server error message, in combination with the SQL command, to diagnose the issue. The error message does provide a line number indicating the start of the command with the issue. Since in this case we only have one command, it states that it begins on line 1. If we were to be executing multiple commands, the line number would be helpful.

When we see this error message, we need to think, “Somewhere in the command that begins on line 1 is a reference to ‘Person’ that does not exist.” We can then search through the command and find the invalid reference. In our simple case, we only have one reference to “Person”, and we attempted to reference it as a table, and so we know that the Person table does not exist. This is what we expected, since we previously dropped the Person table.

### ***PostgreSQL Explanation***

Using the provided error message along with the SQL command we are able to diagnose the issue.

The error message generated in PostgreSQL specifies a short description of the error message along with the line number in the script the error corresponds to and the more complex the SQL command the more beneficial the line number becomes. In this example the error message states that the “person” does not exist and points us to Line 2 where the “person” object is referenced in the FROM statement.

## Section Two – More Precise Data Handling

### Step 7 – Table Setup

In this section, we work with the Camera table illustrated below in order to provide you with examples you can follow. Note that the bolded columns represent those with a NOT NULL constraint.

Camera	
PK	<b>camera_id</b> DECIMAL(12)
	<b>model</b> VARCHAR(64) description VARCHAR(255) <b>date_manufactured</b> DATE

To create the Camera table, we use the following command.

```
CREATE TABLE Camera (  
  camera_id DECIMAL(12) PRIMARY KEY,  
  model VARCHAR(64) NOT NULL,  
  description VARCHAR(255),  
  date_manufactured DATE NOT NULL  
);
```

Though in the command above you see some familiar constructs, let us further explain the phrases “PRIMARY KEY” and “NOT NULL”. Because the phrase “PRIMARY KEY” is part of the camera\_id column definition (recall that the comma separates column definitions), the RDBMS knows to apply the PRIMARY KEY constraint to the camera\_id column, and not to any other column. Similarly, the phrase “NOT NULL” applies to both the model and date\_manufactured columns, since the phrase is part of those columns definitions.

It is important to remember two properties of constraints as described in the lecture and textbook. The first is that a constraint defines a condition that the data must satisfy. The second is that the RDBMS continually enforces the condition defined by an active constraint at all times. When we place a constraint on a single column as in the command above, then the value for the column must satisfy the condition, *for every row in the table at all times*. By default, the RDBMS will reject any SQL statement that would cause a constraint violation.

Let us briefly review the concept of NULL. When we assign a datatype to a column, we restrict the legal values for that column. For example, a VARCHAR datatype means that we allow sequences of characters for the column. A DATE datatype means that we allow date values for the column. But, what if we want to indicate that the column has no value at all? The NULL keyword is used for this purpose. When the RDBMS sees the NULL keyword, it knows that no value is to be placed into that column.

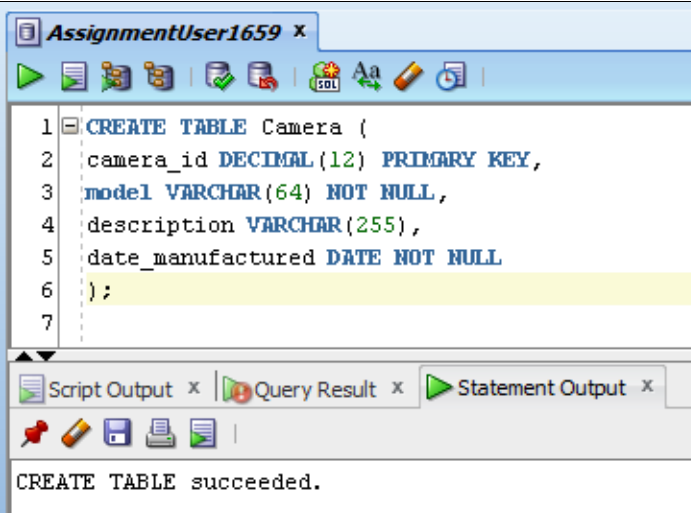
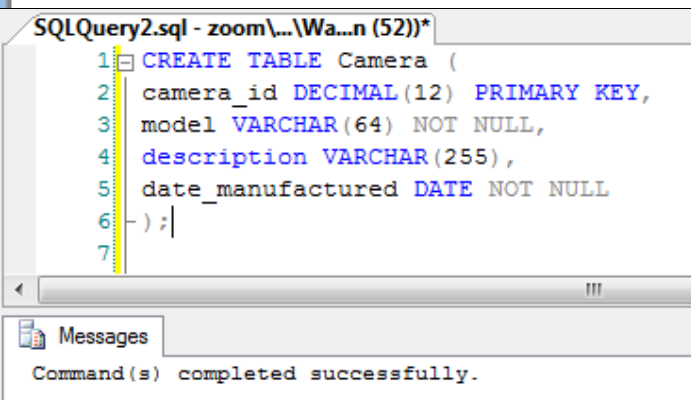


Recall that a NOT NULL constraint indicates that each row in the table must have a value for the column(s) covered by the constraint. Indicating no value for the covered column(s) is illegal. Further recall that a PRIMARY KEY constraint is a combination of NOT NULL and UNIQUE constraints. The value(s) covered by the PRIMARY KEY must always be present and unique. The benefit of a primary key value is that it can always be used to uniquely identify a row in a table.

By default columns are nullable, meaning that a value for that column is optional; any particular row may omit a value for that column. This default is why a NOT NULL constraint was used to ensure that both the model and date\_manufactured columns always have values. This default means that the description column is nullable, since the “NOT NULL” phrase is *not* part of the description column definition.

You may have noticed that the date\_manufactured column has a DATE datatype. A DATE datatype indicates that a year, month, and day may be stored in the column. Some database management systems also allow additional time information to be stored in a DATE column. For example, Oracle allows the additional storage of hours, minutes, and seconds in a DATE column. However, the standards for SQL specify that DATE columns only store the year, month, and day, and it is a best practice to use DATE columns to store only these fields, for portability.

Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer	 <p>The screenshot shows the Oracle SQL Developer interface. The main window displays the following SQL code:</p> <pre> 1 CREATE TABLE Camera ( 2   camera_id DECIMAL(12) PRIMARY KEY, 3   model VARCHAR(64) NOT NULL, 4   description VARCHAR(255), 5   date_manufactured DATE NOT NULL 6 ); 7 </pre> <p>Below the code editor, the 'Script Output' tab is active, showing the message: 'CREATE TABLE succeeded.'</p>
Microsoft SQL Server Management Studio	 <p>The screenshot shows the Microsoft SQL Server Management Studio interface. The main window displays the following SQL code:</p> <pre> 1 CREATE TABLE Camera ( 2   camera_id DECIMAL(12) PRIMARY KEY, 3   model VARCHAR(64) NOT NULL, 4   description VARCHAR(255), 5   date_manufactured DATE NOT NULL 6 ); 7 </pre> <p>Below the code editor, the 'Messages' tab is active, showing the message: 'Command(s) completed successfully.'</p>

pgAdmin

CS669 on postgres@PostgreSQL 10

6

CREATE TABLE Camera (

7

camera\_id DECIMAL(12) PRIMARY KEY,

8

model VARCHAR(64) NOT NULL,

9

description VARCHAR(255),

10

date\_manufactured DATE NOT NULL

11

);

Data Output

Explain

Messages

Query History

CREATE TABLE

Query returned successfully in 361 msec.

## Step 8 – Table Population

We start by inserting the following row into the Camera table.

**camera\_id** = 51  
**model** = ProShot 5000  
**description** = Useful for portraits  
**date\_manufactured** = 21-FEB-2008

```
INSERT INTO Camera (camera_id,  
                    model,  
                    description,  
                    date_manufactured)  
VALUES (51,  
        'ProShot 5000',  
        'Useful for portraits',  
        CAST('21-FEB-2008' AS DATE)  
);
```

You have already seen examples of inserting numbers and character sequences, but inserting dates is worth a more detailed look. If we want to insert a number, we simply type the number in the format we are used to, and the RDBMS knows that what we typed is a literal number. For example, we simply typed “51” as in the example above. If we want to type a character sequence, we simply enclose it between apostrophes ('), and the RDBMS knows that what we typed is a literal character sequence. For example, we typed 'ProShot 5000' above. Indicating literal date values in a portable way is not as simple.

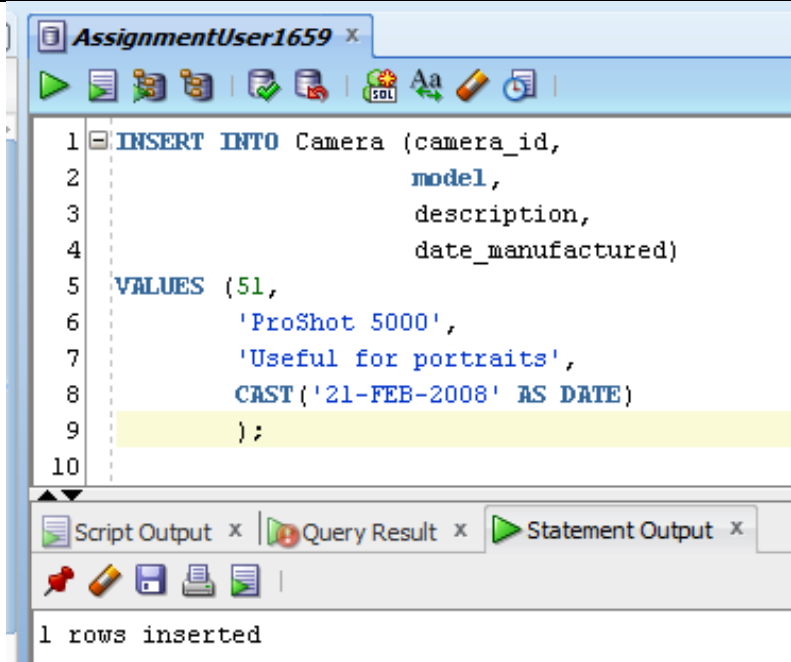
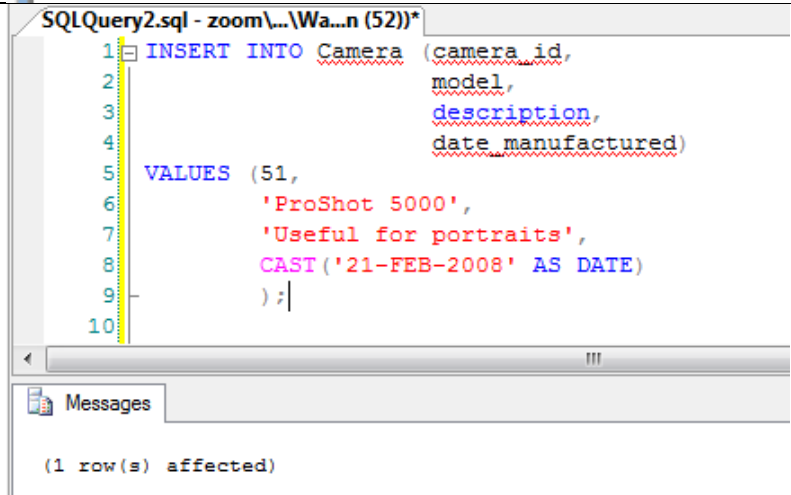
One common, portable way to specify a date literal is to type the date as a character sequence, then use the CAST operator to convert the character sequence to a date. The CAST operator is necessary because otherwise the RDBMS may not treat the character sequence value as a date. Each specific RDBMS has non-portable methods of specifying date literals as well. For example, Oracle offers a TO\_DATE function, and SQL Server implicitly converts standalone character sequences into dates if a date is expected. Though there are many portable and non-portable methods of specifying date literals, one commonality for most of these methods is character sequence representation of the date in the command. In our example, we used the character sequence “21-FEB-2008” to indicate February 21<sup>st</sup>, 2008.

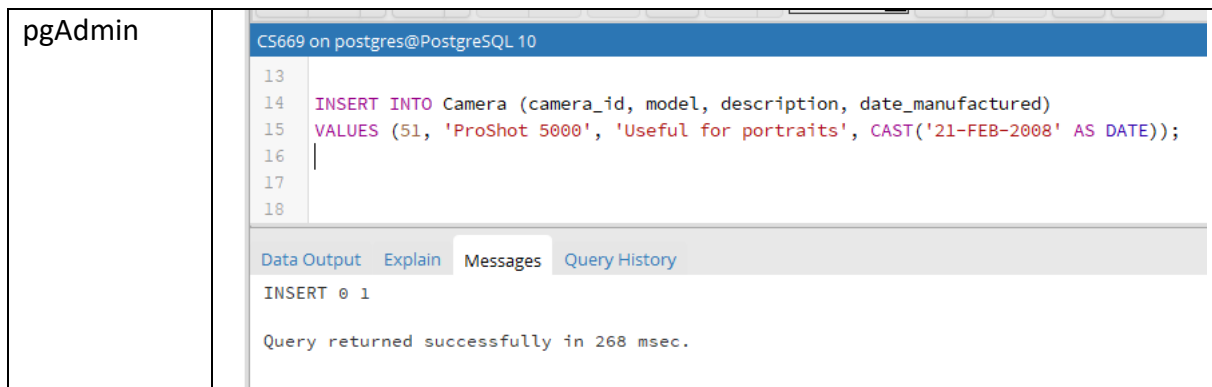
The format of the date character sequence determines portability between RDBMS using different date conventions. In the example above, we used the date format “dd-mmm-yyyy”, where “dd” represents two digits for the day of month, “mmm” represents the first three letters of the month, and “yyyy” represents the four digits of the year. If we had used “02/21/2008” in the date format typically used in the United States, European RDBMS would not have interpreted the date as we expected, since European date standards specify the day of month first, followed by the month, following by the year (“21/02/2008” for this example). If we had used the European date format, RDBMS in the United States would not have interpreted the date as we expect. For this reason, it is best to use the more portable specification “dd-mmm-yyyy”.

*Troubleshooting Note:* The region or language settings of your operating system determine the globalization options selected by your DBMS at the time of installation, which in turn determine whether these RDBMS

accepts the “dd-mmm-yyyy” format listed above. For Windows users, the operating system setting is under “Region and Language” in the control panel. If you find that your DBMS does not accept the format, this is likely because of your region or language, and you may enter the format accepted by your region and language. More information on which format in your region is acceptable may be found at <https://www.postgresql.org/docs/10/static/locale.html> [https://docs.oracle.com/cd/E11882\\_01/server.112/e10729/ch3globenv.htm#NLSPG003](https://docs.oracle.com/cd/E11882_01/server.112/e10729/ch3globenv.htm#NLSPG003). Alternatively, temporarily changing your Region and Language to “United States” may resolve the issue.

Below is a sample screenshot of the command execution in Oracle SQL Developer

Oracle SQL Developer	 <pre> 1  INSERT INTO Camera (camera_id, 2                        model, 3                        description, 4                        date_manufactured) 5  VALUES (51, 6          'ProShot 5000', 7          'Useful for portraits', 8          CAST('21-FEB-2008' AS DATE) 9          ); 10 </pre> <p>1 rows inserted</p>
Microsoft SQL Server Management Studio	 <pre> 1  INSERT INTO Camera (camera_id, 2                        model, 3                        description, 4                        date_manufactured) 5  VALUES (51, 6          'ProShot 5000', 7          'Useful for portraits', 8          CAST('21-FEB-2008' AS DATE) 9          ); 10 </pre> <p>(1 row(s) affected)</p>



We can see that each RDBMS notifies the number of rows affected/inserted.

We now insert another row.

**camera\_id** = 52

**model** = NaturalShot 300x

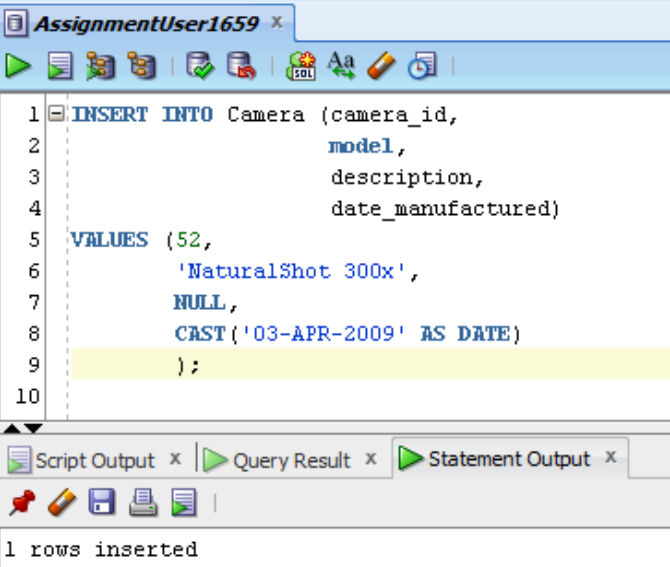
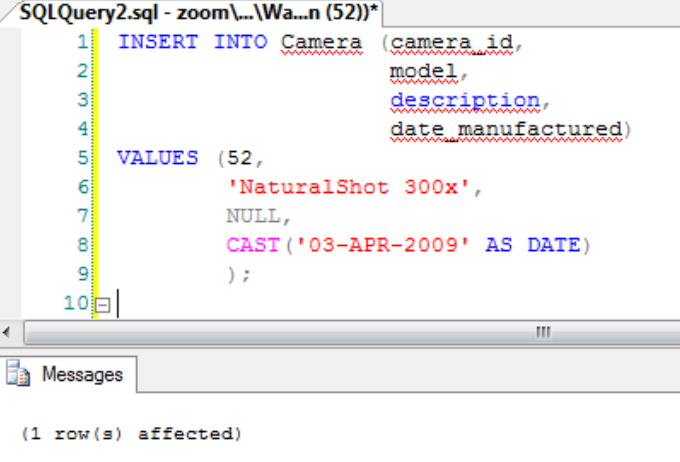
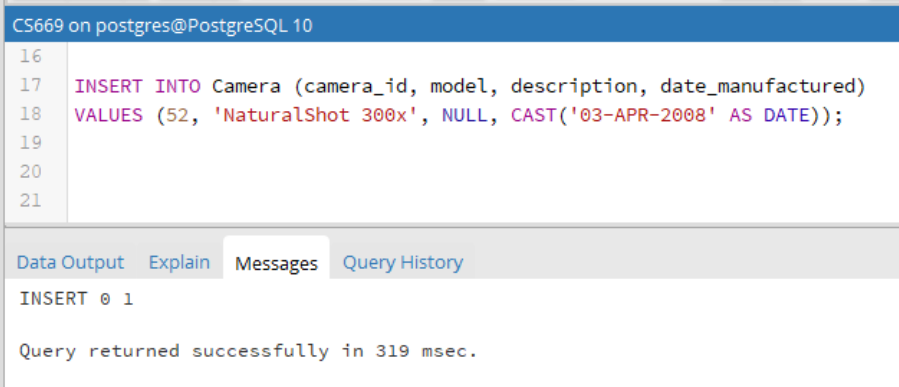
**description** = NULL

**date\_manufactured** = 03-APR-2009

```
INSERT INTO Camera (camera_id,
                    model,
                    description,
                    date_manufactured)
VALUES (52,
        'NaturalShot 300x',
        NULL,
        CAST('03-APR-2009' AS DATE)
);
```

Notice that we used the NULL keyword to indicate that the description column has no value for this row. Now that you understand the concept of null, and how to insert null values, you can see that it is really quite simple to indicate that a column does not have a value.

Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	 <pre> 1  INSERT INTO Camera (camera_id, 2                        model, 3                        description, 4                        date_manufactured) 5  VALUES (52, 6            'NaturalShot 300x', 7            NULL, 8            CAST('03-APR-2009' AS DATE) 9  ); 10 </pre> <p>1 rows inserted</p>
Microsoft SQL Server Management Studio	 <pre> 1  INSERT INTO Camera (camera_id, 2                        model, 3                        description, 4                        date_manufactured) 5  VALUES (52, 6            'NaturalShot 300x', 7            NULL, 8            CAST('03-APR-2009' AS DATE) 9  ); 10 </pre> <p>(1 row(s) affected)</p>
pgAdmin	 <pre> 16 17  INSERT INTO Camera (camera_id, model, description, date_manufactured) 18  VALUES (52, 'NaturalShot 300x', NULL, CAST('03-APR-2008' AS DATE)); 19 20 21 </pre> <p>INSERT 0 1</p> <p>Query returned successfully in 319 msec.</p>

The next row we insert is done so in a slightly different way.

**camera\_id** = 53

**model** = StillShot 2012

**date\_manufactured** = 05-JUN-2009

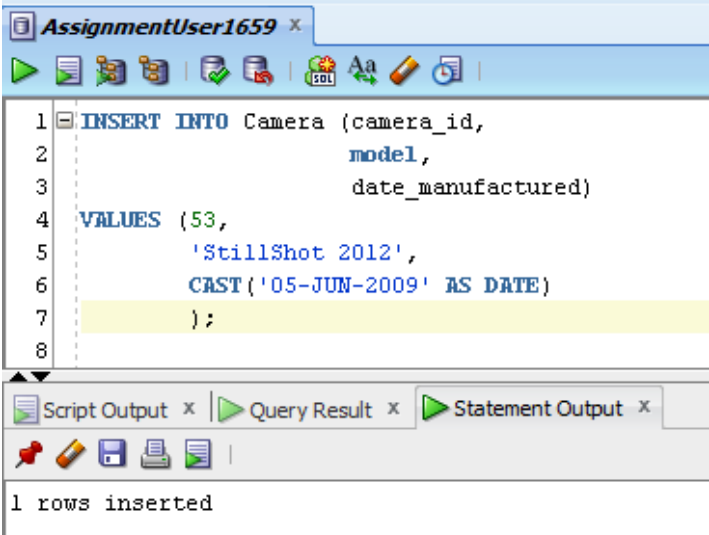
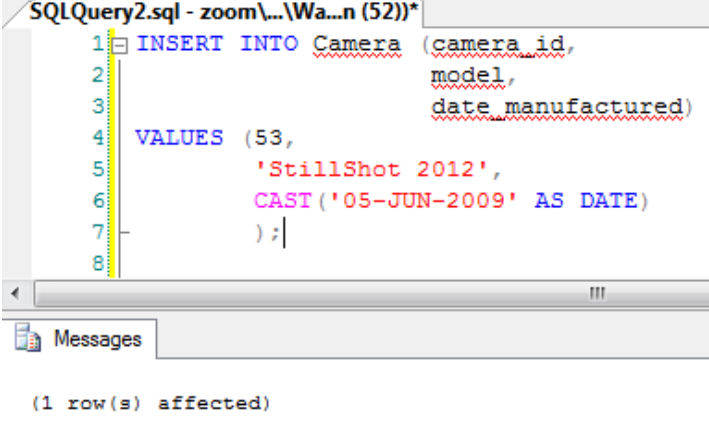
```

INSERT INTO Camera (camera_id,
                    model,
                    date_manufactured)
VALUES (53,
        'StillShot 2012',
        CAST('05-JUN-2009' AS DATE)
        );

```

Notice that we omitted the description column from column identification list and the column value list. When the RDBMS sees the column omitted, it knows by default to give the column no value. This command illustrates another way to avoid inserting a value for a column.

Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer	 <pre> AssignmentUser1659 x 1 INSERT INTO Camera (camera_id, 2                      model, 3                      date_manufactured) 4 VALUES (53, 5          'StillShot 2012', 6          CAST('05-JUN-2009' AS DATE) 7          ); 8 Script Output x   Query Result x   Statement Output x 1 rows inserted </pre>
Microsoft SQL Server Management Studio	 <pre> SQLQuery2.sql - zoom\...\Wa...n (52)* 1 INSERT INTO Camera (camera_id, 2                      model, 3                      date_manufactured) 4 VALUES (53, 5          'StillShot 2012', 6          CAST('05-JUN-2009' AS DATE) 7          ); 8 Messages (1 row(s) affected) </pre>

pgAdmin

CS669 on postgres@PostgreSQL 10

19

20

21

22

23

24

```
INSERT INTO Camera (camera_id, model, date_manufactured)
VALUES (53, 'StillShot 2012', CAST('05-JUN-2009' AS DATE));
```

Data Output

Explain

Messages

Query History

INSERT 0 1

Query returned successfully in 360 msec.

Now that we also have inserted three rows into our Camera table, let us see what we have in our table.

Oracle SQL Developer

AssignmentUser1659 x

1

2

3

```
SELECT *
FROM CAMERA;
```

Script Output x

Statement Output x

Query Result x

SQL

All Rows Fetched: 3 in 0.001 seconds

	CAMERA_ID	MODEL	DESCRIPTION	DATE_MANUFACTURED
1	51	ProShot 5000	Useful for portraits	21-FEB-08
2	52	NaturalShot 300x	(null)	03-APR-09
3	53	StillShot 2012	(null)	05-JUN-09

Microsoft SQL Server Management Studio

SQLQuery2.sql - zoom\...\Wa...n (52))\*

1

2

3

```
SELECT *
FROM CAMERA;
```

Results

Messages

	camera_id	model	description	date_manufactured
1	51	ProShot 5000	Useful for portraits	2008-02-21
2	52	NaturalShot 300x	NULL	2009-04-03
3	53	StillShot 2012	NULL	2009-06-05

pgAdmin

CS669 on postgres@PostgreSQL 10

23

24

25

26

27

```
SELECT *
FROM Camera;
```

Data Output

Explain

Messages

Query History

	camera_id numeric (12)	model character varying (64)	description character varying (255)	date_manufactured date
1	51	ProShot 5000	Useful for portraits	2008-02-21
2	52	NaturalShot 300x	[null]	2008-04-03
3	53	StillShot 2012	[null]	2009-06-05



Notice that although we used two different methods to insert the last two rows, both rows do not have a value for description column just the same. The RDBMS uses the NULL keyword to indicate a lack of a value. Further notice that Oracle and SQL Server display the dates differently when returned from the query. How the dates are rendered varies with different RDBMS and with different SQL clients.

## Step 9 – Invalid Insertion

To show you how this works, let us try to insert a row into the Camera table with no value for the model column. The insertion will fail because a NOT NULL constraint is present on the model column, with values as follows.

**camera\_id** = 54

**model** = NULL

**description** = Multi-purpose camera

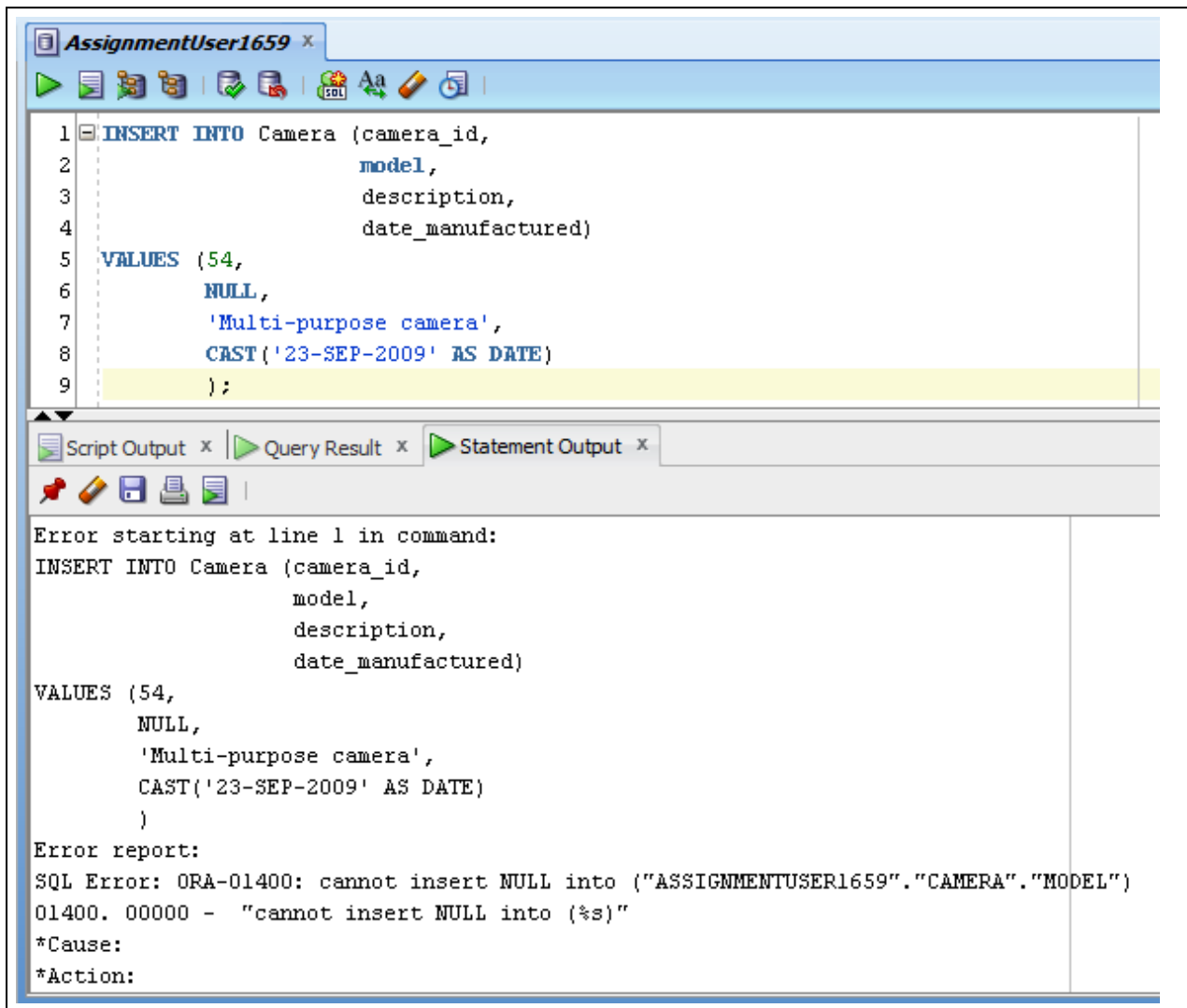
**date\_manufactured** = 23-SEP-2009

We do so with the following command.

```
INSERT INTO Camera (camera_id,  
                    model,  
                    description,  
                    date_manufactured)  
VALUES (54,  
        NULL,  
        'Multi-purpose camera',  
        CAST('23-SEP-2009' AS DATE)  
);
```

Notice that we use the keyword NULL in place of a value for the model column.

When we execute this in Oracle SQL Developer, we see the following.



Once we understand the concept of NULL, the error message from Oracle is fairly straightforward, though some parts need some interpretation. The message states that one cannot insert NULL into the model column for the Camera table, though you may wonder why the character sequence:

"ASSIGNMENTUSER1659"."CAMERA"."MODEL"

appears in the screenshot above, since it is somewhat difficult to read. You see the familiar "CAMERA" and "MODEL" keywords, which represent the Camera table and model column, respectively. They are separated with a period (.) because this is the convention for "drilling down" into a table. We use:

TableName.ColumnName

to reference a specific column in a table in an RDBMS.

The reason why ASSIGNMENTUSER1659 appears in the screenshot above is because that is the schema that was used when this assignment was created. We have not studied the concept of a schema yet, but briefly, a schema is a logical and physical storage location for durable entities. Modern RDBMS support multiple schemas so that multiple users and applications can keep their durable entities separate, in their own

containers. In Oracle, each user is assigned its own schema by default, so when the author of this assignment logged in with user ASSIGNMENTUSER1659, the tables were created by default in the ASSIGNMENTUSER1659 schema. That is why we see the ASSIGNMENTUSER1659 schema in the error message above.

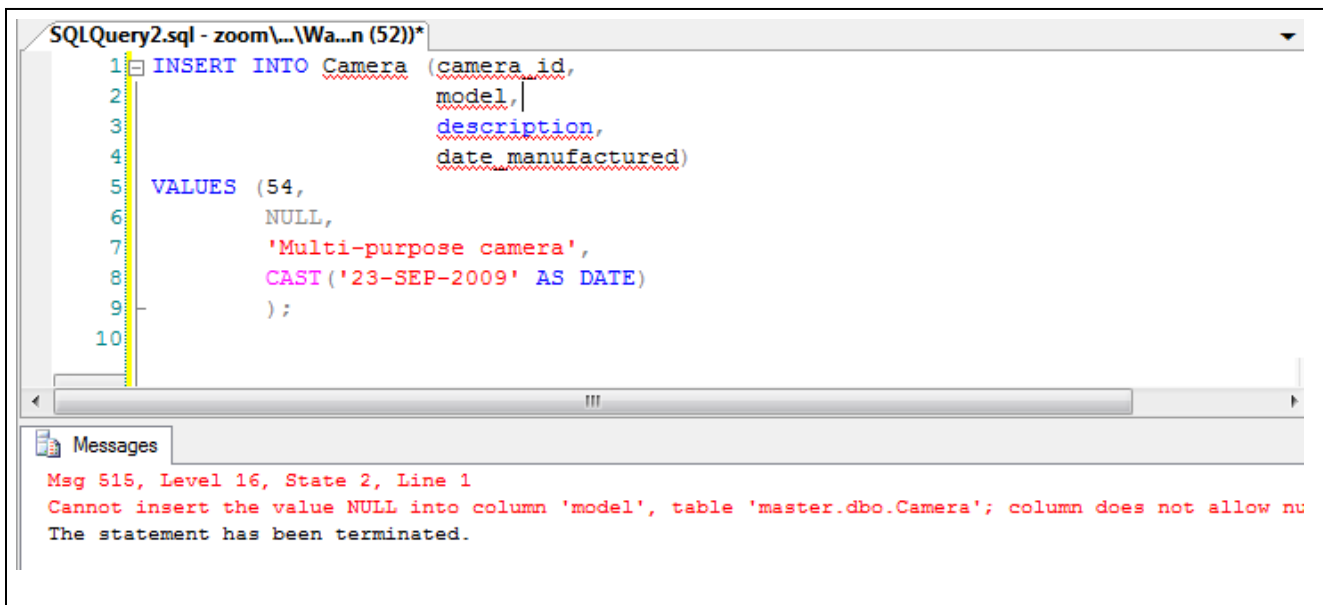
Now, it is unlikely that you chose the same username, and so you will see your own username in the screenshot here, instead of the author's username. What is important to note, however, is that we can also specify a schema name in front of any table name. We use:

SchemaName.TableName.ColumnName

to reference a column, if we also want to specify the schema, in an RDBMS.

The reason each entity is enclosed in quotes is because in order to support all durable entity names, including those which have spaces or other special characters, the RDBMS surrounds all names with quotes to avoid any ambiguity.

Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



Part of the error message was truncated in the screenshot, so let us show it here:

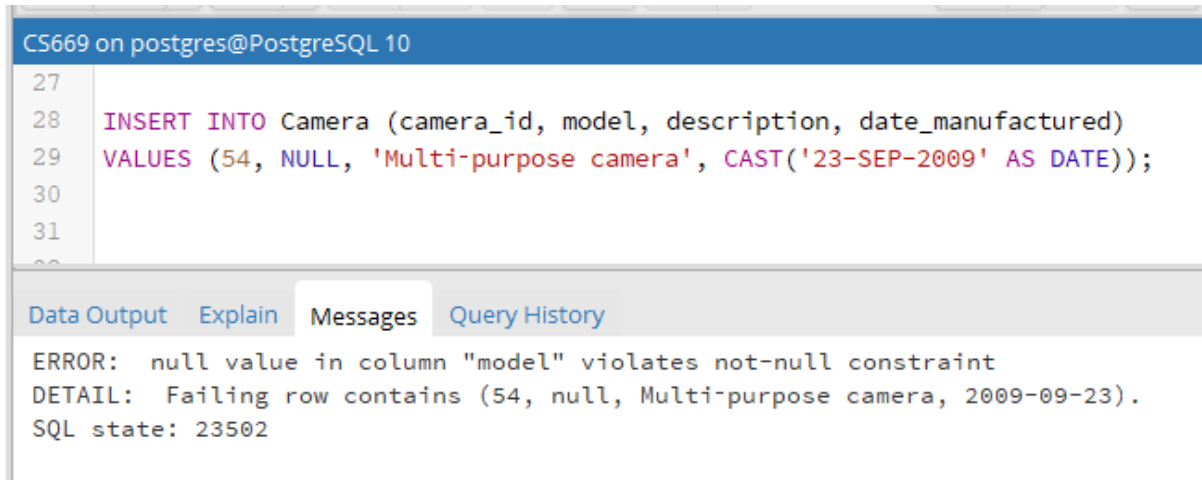
```
--  
Msg 515, Level 16, State 2, Line 1  
Cannot insert the value NULL into column 'model', table 'master.dbo.Camera'; column does not allow nulls.  
INSERT fails.  
The statement has been terminated.  
--
```

The SQL Server error message in this instance may be easier to interpret than the corresponding Oracle error message. It directly states that a NULL value cannot be inserted into the model column. Just as with Oracle, we see that there is further qualification of the table. In the screenshot above, it is “master.dbo.Camera”. In general form, it is:

DatabaseName.SchemaName.TableName

In the screenshot above, it is indicating that the Camera table resides in database “master”, in schema “dbo”. “Master” is the default database, and “dbo” is the default schema for a database user. If you followed the instructions to use a database other than “master”, you will see a different database name in your screenshot.

Below is a sample screenshot of the command execution in pgAdmin for PostgreSQL.

A screenshot of the pgAdmin interface. At the top, a blue header bar reads "CS669 on postgres@PostgreSQL 10". Below this is a text editor area with line numbers 27 through 33. Line 28 contains the SQL command: `INSERT INTO Camera (camera_id, model, description, date_manufactured)`. Line 29 contains the values: `VALUES (54, NULL, 'Multi-purpose camera', CAST('23-SEP-2009' AS DATE));`. Below the editor is a tabbed interface with four tabs: "Data Output", "Explain", "Messages", and "Query History". The "Messages" tab is selected, showing an error message: `ERROR: null value in column "model" violates not-null constraint`. Below this is the detail: `DETAIL: Failing row contains (54, null, Multi-purpose camera, 2009-09-23).` and the SQL state: `SQL state: 23502`.

The error message PostgreSQL (pgAdmin) is fairly straight forward as it clearly states NULL value in column “model” is not allowed and violates an established not null constraint. The message also provides information about the values in the row that was attempting to be inserted. Unlike Oracle and MS SQL the error message does not contain information about the table, schema or database that was involved.

Note that different clients or different database versions may give different error messages than those illustrated above, but they will still convey the same information.

## Step 10 – Valid Insertion

This step calls for a standard insert since no nulls are involved. The explanations in prior steps illustrates standard inserts.

## Step 11 – Filtered Results

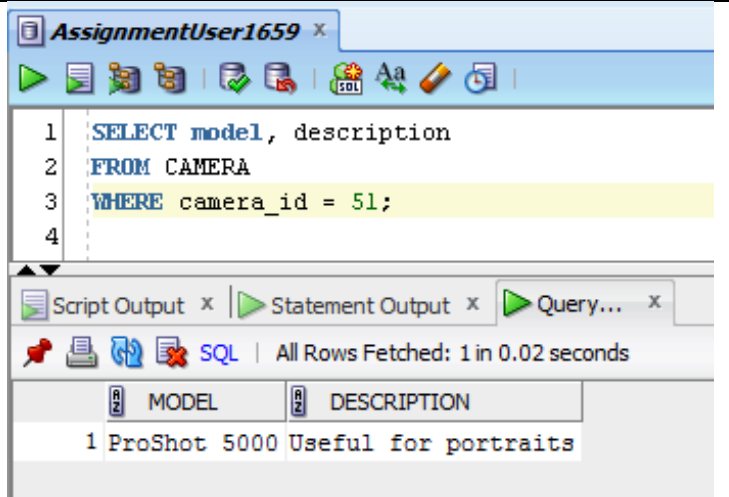
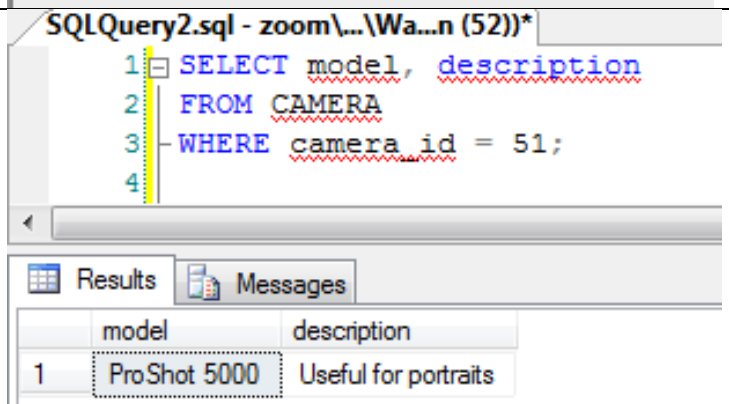
To show you how to do this, let us practice doing the same on the Camera table. We can limit the number of rows by its primary key, and also restrict the columns returned, with the following command.

```
SELECT model, description
FROM CAMERA
WHERE camera_id = 51;
```

By indicating the names of the columns that we would like to see next to the SELECT keyword, where each column name is separated by a comma, we are omitting any columns not in the list. In this example, we have indicated that we would like to see the model and description columns, but no other columns.

By introducing a WHERE clause at the end of the SELECT statement, and using a Boolean expression that indicates that we only want to see rows where the camera\_id value is 51, we have limited the number of rows returned in our result set. Try the command and see what happens.

Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer	 <p>AssignmentUser1659 x</p> <pre>1 SELECT model, description 2 FROM CAMERA 3 WHERE camera_id = 51; 4</pre> <p>Script Output x Statement Output x Query... x</p> <p>SQL   All Rows Fetched: 1 in 0.02 seconds</p> <table border="1"><thead><tr><th>MODEL</th><th>DESCRIPTION</th></tr></thead><tbody><tr><td>1 ProShot 5000</td><td>Useful for portraits</td></tr></tbody></table>	MODEL	DESCRIPTION	1 ProShot 5000	Useful for portraits		
MODEL	DESCRIPTION						
1 ProShot 5000	Useful for portraits						
Microsoft SQL Server Management Studio	 <p>SQLQuery2.sql - zoom\...\Wa...n (52))*</p> <pre>1 SELECT model, description 2 FROM CAMERA 3 WHERE camera_id = 51; 4</pre> <p>Results Messages</p> <table border="1"><thead><tr><th></th><th>model</th><th>description</th></tr></thead><tbody><tr><td>1</td><td>ProShot 5000</td><td>Useful for portraits</td></tr></tbody></table>		model	description	1	ProShot 5000	Useful for portraits
	model	description					
1	ProShot 5000	Useful for portraits					

pgAdmin

CS669 on postgres@PostgreSQL 10

31

32

33

34

35

```
SELECT model, description
FROM Camera
WHERE camera_id = 51;
```

Data Output

Explain

Messages

Query History

	model character varying (64)	description character varying (255)
1	ProShot 5000	Useful for portraits

Notice that each screenshot, only the row where camera\_id is 51 is returned, and only the model and description columns are shown.

Why not always select all rows and all columns from the table? Why bother restricting either? The answer is efficiency. If we have a production table with a billion rows and thirty columns, but only need one column's value for one of those rows, it would be time and resource inefficient for the database to obtain all column's value from all billion rows. It would also be time and resource inefficient for the end-user or application to search through the billion results for the match. If properly configured, an RDBMS will very efficiently retrieve the limited results needed, even if the number of rows and columns in the table is large.

The Boolean expression in a WHERE clause need not only indicate one row. It can indicate many rows. In fact, there are a variety of Boolean and mathematical operators that can be present in these expressions, in virtually endless combinations. Exploring all operators and combinations is beyond the scope of this lab. If you are curious, you can explore these in the lecture and textbook, and in other resources.



## Step 12 – Updating a Column

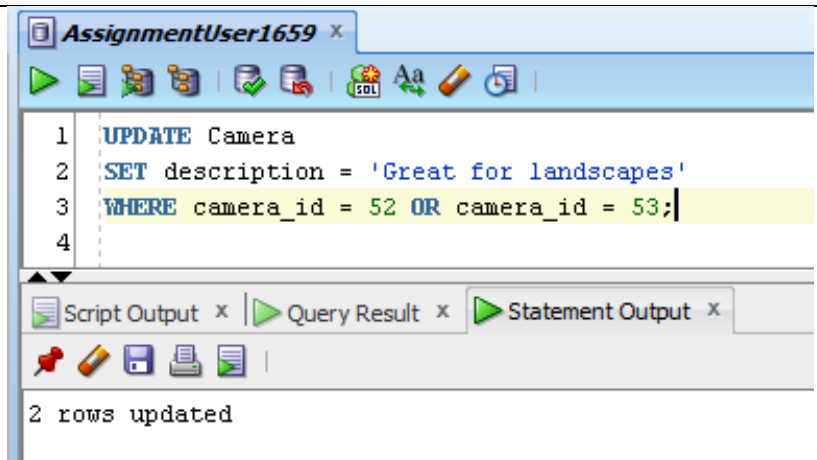
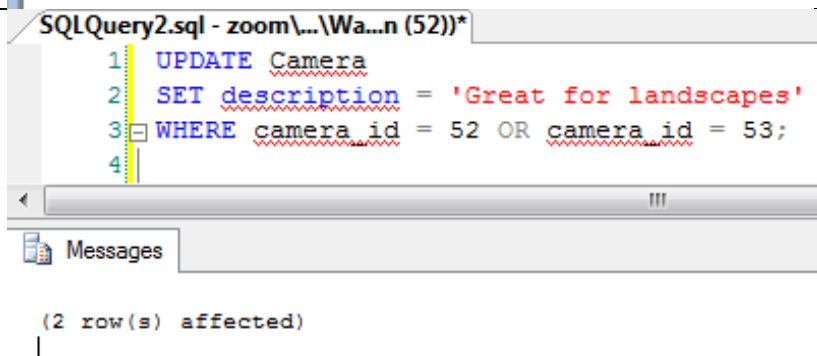
The good news is that we do not need to learn three different ways to limit the rows affected in the SELECT, UPDATE, and DELETE FROM commands. We use the same general WHERE clause for all three commands. So what you learned in the previous step can also be applied here.

To illustrate this, imagine that we want to add the same description to both camera rows that have no description (where camera\_id = 52 and 53), but want to avoid updating the row that already has a description. We can do so with the following command.

```
UPDATE Camera
SET description = 'Great for landscapes'
WHERE camera_id = 52 OR camera_id = 53;
```

Notice that we used the OR Boolean operator to combine two expressions. The UPDATE applies to the row that matches either expression.

Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	
Microsoft SQL Server Management Studio	

pgAdmin

C5669 on postgres@PostgreSQL 10

36

37

38

39

40

UPDATE Camera

SET description = 'Great for landscapes'

WHERE camera\_id = 52 OR camera\_id = 53;

Data Output

Explain

Messages

Query History

UPDATE 2

Query returned successfully in 342 msec.

Notice that each screenshot indicate two rows were affected, which is what we expect. We did not update all rows in the table, but only those that matched the condition expressed in the WHERE clause.

## Step 13 – Updating to Null

Doing this step requires the same constructs as used in prior steps, except that NULL is used in place of a value.

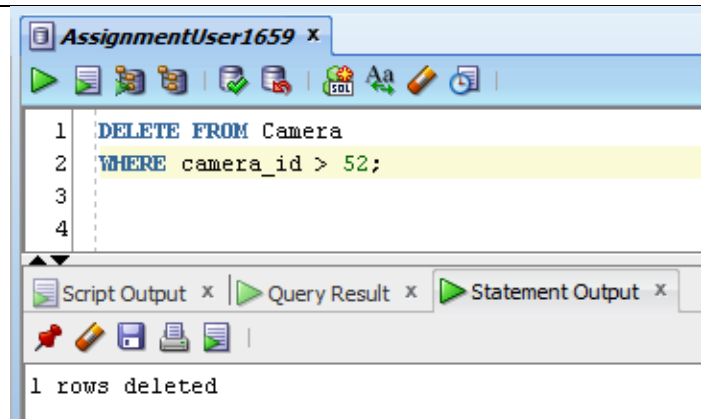
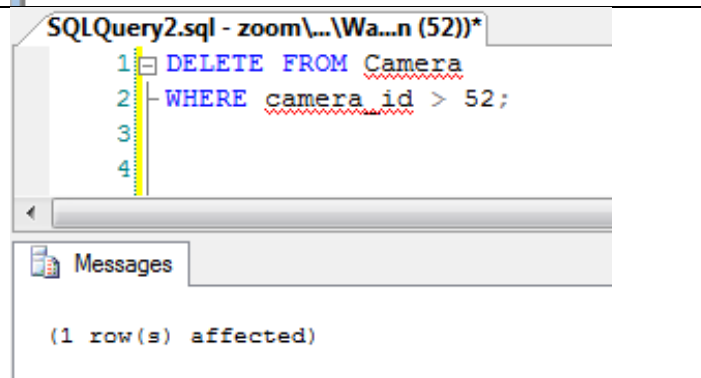
## Step 14 – Targeted Deletion

We can append a general WHERE clause to a DELETE FROM command in order to delete only rows matching a specific condition. For example, to delete the row where camera\_id = 53. We will do so using a different Boolean expression.

```
DELETE FROM Camera
WHERE camera_id > 52;
```

Notice that we used the greater than (>) operator, and we can translate this statement to English as, “Delete all rows from the Camera table that have a camera\_id value greater than 52”. In our case, we only had one row with a camera\_id value greater than 52, and that was the row where camera\_id = 53. If there had been additional rows with higher camera\_id values, they would have also been deleted.

Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	 The screenshot shows the Oracle SQL Developer interface. The main window displays a SQL script with the following text: <pre>1 DELETE FROM Camera 2 WHERE camera_id &gt; 52; 3 4</pre> The script is highlighted in yellow. Below the script, the 'Query Result' tab is active, showing the message '1 rows deleted'.
Microsoft SQL Server Management Studio	 The screenshot shows the Microsoft SQL Server Management Studio interface. The main window displays a SQL script with the following text: <pre>1 DELETE FROM Camera 2 WHERE camera_id &gt; 52; 3 4</pre> The script is highlighted in yellow. Below the script, the 'Messages' tab is active, showing the message '(1 row(s) affected)'.

pgAdmin

CS669 on postgres@PostgreSQL 10

45

46 `DELETE FROM Camera`

47 `WHERE camera_id > 52;`

48

49

Data Output

Explain

Messages

Query History

DELETE 1

Query returned successfully in 412 msec.

Now let us view the contents of the Camera table, so that we can see the results of both the UPDATE and DELETE commands we executed previously. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer

AssignmentUser1659 x

Run Script (F5)

1

2 `FROM CAMERA;`

3

4

Script Output x

Statement Output x

Query... x

SQL | All Rows Fetched: 2 in 0.001 seconds

CAMERA_ID	MODEL	DESCRIPTION	DATE_MANUFACTURED
1	51 ProShot 5000	Useful for portraits	21-FEB-08
2	52 NaturalShot 300x	Great for landscapes	03-APR-09

Microsoft SQL Server Management Studio

SQLQuery2.sql - zoom\...\Wa...n (52))\*

1 `SELECT *`

2 `FROM CAMERA;`

3

4

Results Messages

	camera_id	model	description	date_manufactured
1	51	ProShot 5000	Useful for portraits	2008-02-21
2	52	NaturalShot 300x	Great for landscapes	2009-04-03

pgAdmin

CS669 on postgres@PostgreSQL 10

41

42 `SELECT *`

43 `FROM Camera;`

44

45

Data Output

Explain

Messages

Query History

	camera_id numeric (12)	model character varying (64)	description character varying (255)	date_manufactured date
1	51	ProShot 5000	Useful for portraits	2008-02-21
2	52	NaturalShot 300x	Great for landscapes	2008-04-03

Notice that in each screenshot, the row where camera\_id = 53 has been deleted, and the row where camera\_id = 52 now has the description of “Great for landscapes”. This is what we expected.

## Section Three – Concepts Demonstration

### Step 15 – Data Anomalies

When the same data repeats multiple times, anomalies can result. One such anomaly occurs when data that is already present is added again with some different values. The original values and the new values disagree, creating a question as to which of the values is accurate. For example, imagine a Pizza table contains the price for a Veggie Lovers pizza.

PizzaName	PizzaPrice
Veggie Lovers	\$10.99

What happens if the pizza shop lowers the price of the pizza? If they are not careful, a second row may be inserted with the same pizza, but a different price, like the below.

PizzaName	PizzaPrice
Veggie Lovers	\$10.99
Veggie Lovers	\$9.99

The Veggie Lovers pizza now appears twice with two different prices. When an application, report, or person retrieves this data, how will they know which price is correct from this table? There may be some tricks that can be performed to try and determine the accurate price for today, such as using timestamps or row ids, but in general those tricks do not work for every situation, and add tremendous complexity to handling of the data. We want our data to be *unambiguous and definitive*, but the data in the table above is neither.

In practice, if data is duplicated, the table could end up with other pizzas with the same issue, like the below.

PizzaName	PizzaPrice
Veggie Lovers	\$10.99
Veggie Lovers	\$9.99
Inferno	\$9.59
Plain	\$8.99
Inferno	\$9.29
Veggie Lovers	\$10.59

Now the Inferno pizza also has two prices, and the Veggie Lovers pizza has a third price! Imagine that this table is used for years in the pizza shop. It would end up with possibly dozens of pizzas, each with many rows of duplication, with a new row every time a price is changed. It would be near impossible to determine which price is accurate. Hopefully it is becoming very clear to why this kind of duplication must be avoided.

You might argue the issue above is academic. Why not simply update the price rather than insert? That is correct in our simple example, so let us expand our example to something more real world. Imagine that the pizza shop uses a table to track purchases, like so:

PizzaName	PizzaPrice	Customer Name	Date
Veggie Lovers	\$10.99	Bob Smith	10/21/2022
Veggie Lovers	\$9.99	Elaine Yang	11/1/2022
Inferno	\$9.59	Elaine Yang	11/1/2022
Plain	\$8.99	Prasad Santhana	11/12/2022
Inferno	\$9.29	David Wolfe	11/13/2022
Veggie Lovers	\$10.59	David Wofe	11/13/2022

If proper design techniques are not used, this table may contain the *only* record of the price of a pizza, the customer's name, and the pizza name. Essentially, what should be three different tables -- Pizza, Customer, and Purchase -- are combined into a single table. From this example, we may not be able to determine the accurate price for a pizza at any given time. The price will be repeated for every purchase. The same goes for the pizza name and the customer's name. With this example, it's straightforward to see how multiple copies of data can be inserted in a real-world situation.

This type of anomaly occurs when the same data is *added again with different* values, such as inserting the price of the same pizza repeatedly. This can also occur when the same data is present multiple times, and an update of some but not all of the data causes the values to disagree. Continuing with our pizza shop example, imagine that two purchases of the Plain pizza initially reflect the same price, highlighted in the below.

PizzaName	PizzaPrice	Customer Name	Date
Veggie Lovers	\$10.99	Bob Smith	10/21/2022
Veggie Lovers	\$9.99	Elaine Yang	11/1/2022
Inferno	\$9.59	Elaine Yang	11/1/2022
Plain	\$8.99	Prasad Santhana	11/12/2022
Inferno	\$9.29	David Wolfe	11/13/2022
Veggie Lovers	\$10.59	David Wofe	11/13/2022
Plain	\$8.99	Batman	11/27/2022

Although that Plain information is duplicated, there is no anomaly because they are in agreement. What happens if the pizza shop realizes that they actually sold the second Plain pizza for \$9.99, but mistakenly entered it as \$8.99? They will need to update the existing row, resulting the below with the mismatched prices highlighted.

PizzaName	PizzaPrice	Customer Name	Date
Veggie Lovers	\$10.99	Bob Smith	10/21/2022
Veggie Lovers	\$9.99	Elaine Yang	11/1/2022
Inferno	\$9.59	Elaine Yang	11/1/2022
Plain	\$8.99	Prasad Santhana	11/12/2022
Inferno	\$9.29	David Wolfe	11/13/2022
Veggie Lovers	\$10.59	David Wofe	11/13/2022
Plain	\$9.99	Batman	11/27/2022

In this case, the update of the data caused the anomaly rather than the original adding of the data. Nevertheless, the same problem still occurs, disagreement between multiple copies of the same information. The difference between an update and insert anomaly therefore focuses around which type of operation caused the anomaly, and both anomalies result in the same problems.

A *deletion anomaly* occurs when the deletion of data inadvertently deletes other associated data. Continuing with the pizza shop example, imagine that both orders for the Inferno pizza happened over the phone, and neither person came to pick up the order (orders highlighted below).



PizzaName	PizzaPrice	Customer Name	Date
Veggie Lovers	\$10.99	Bob Smith	10/21/2022
Veggie Lovers	\$9.99	Elaine Yang	11/1/2022
Inferno	\$9.59	Elaine Yang	11/1/2022
Plain	\$8.99	Prasad Santhana	11/12/2022
Inferno	\$9.29	David Wolfe	11/13/2022
Veggie Lovers	\$10.59	David Wofe	11/13/2022
Plain	\$9.99	Batman	11/27/2022

The pizza shop later decides to remove the purchases from their table since they were never completed and paid for. After the delete, the table looks as follows.

PizzaName	PizzaPrice	Customer Name	Date
Veggie Lovers	\$10.99	Bob Smith	10/21/2022
Veggie Lovers	\$9.99	Elaine Yang	11/1/2022
Plain	\$8.99	Prasad Santhana	11/12/2022
Veggie Lovers	\$10.59	David Wofe	11/13/2022
Plain	\$9.99	Batman	11/27/2022

What is important about this deletion is that it *deleted every record of the Inferno pizza in addition to deleting the customer information for Elaine Yang*. To state this another way, removing the purchases themselves was a valid thing for the pizza shop to do, but the anomaly is that in doing so, the pizza shop lost all information about the Inferno pizza and Elaine Yang. This is the crux of what makes a deletion anomaly. Multiple tables -- in this case, Pizza, Purchase, and Customer -- are combined into a single table, and therefore deletes from that table result in permanent loss of information. The pizza shop intended to delete only the purchase, but inadvertently deleted the pizza and customer as well.

## Step 16 – File and Database Table Comparison

We will continue with the pizza shop example used in #15. As a reminder, below is the last iteration of the table.

PizzaName	PizzaPrice	Customer Name	Date
Veggie Lovers	\$10.99	Bob Smith	10/21/2022
Veggie Lovers	\$9.99	Elaine Yang	11/1/2022
Plain	\$8.99	Prasad Santhana	11/12/2022
Veggie Lovers	\$10.59	David Wofe	11/13/2022
Plain	\$9.99	Batman	11/27/2022

We could represent the same information in XML, for example the below.

```
<PizzaShop>
  <Purchase>
    <PizzaName>Veggie Lovers</PizzaName>
    <PizzaPrice>10.99</PizzaPrice>
    <CustomerName>Bob Smith</CustomerName>
    <Date>10/21/2022</Date>
  </Purchase>
</Purchase>
  <Purchase>
    <PizzaName>Veggie Lovers</PizzaName>
    <PizzaPrice>9.99</PizzaPrice>
    <CustomerName>Elaine Yang</CustomerName>
    <Date>11/1/2022</Date>
  </Purchase>
</Purchase>
  <Purchase>
    <PizzaName>Plain</PizzaName>
    <PizzaPrice>8.99</PizzaPrice>
    <CustomerName>Prasad Santhana</CustomerName>
    <Date>11/12/2022</Date>
  </Purchase>
</Purchase>
  <Purchase>
    <PizzaName>Veggie Lovers</PizzaName>
    <PizzaPrice>10.59</PizzaPrice>
    <CustomerName>David Wofe</CustomerName>
    <Date>11/13/2022</Date>
  </Purchase>
</Purchase>
  <Purchase>
    <PizzaName>Plain</PizzaName>
    <PizzaPrice>9.99</PizzaPrice>
    <CustomerName>Batman</CustomerName>
    <Date>11/27/2022</Date>
  </Purchase>
</Purchase>
</PizzaShop>
```

XML always has a root element, which we named `PizzaShop` since we are dealing with pizza shop information. Each row of information is contained with a `Purchase` element since each row represents a purchase. Then, each of the columns are represented with the `PizzaName`, `PizzaPrice`, `CustomerName`, and `Date` elements.

Alternatively, we could use JSON representation, such as the below. There are many valid JSON representations; we selected one of them.

```
[
  {
    "PizzaName": "Veggie Lovers",
    "PizzaPrice": "10.99",
    "CustomerName": "Bob Smith",
    "Date": "10/21/2022"
  },
  {
    "PizzaName": "Veggie Lovers",
    "PizzaPrice": "9.99",
    "CustomerName": "Elaine Yang",
    "Date": "11/1/2022"
  },
  {
    "PizzaName": "Plain",
    "PizzaPrice": "8.99",
    "CustomerName": "Prasad Santhana",
```

```

    "Date": "11/12/2022"
  },
  {
    "PizzaName": "Veggie Lovers",
    "PizzaPrice": "10.59",
    "CustomerName": "David Wofe",
    "Date": "11/13/2022"
  },
  {
    "PizzaName": "Plain",
    "PizzaPrice": "9.99",
    "CustomerName": "Batman",
    "Date": "11/27/2022"
  }
]

```

You can spot the attributes quickly by their name – PizzaName, PizzaPrice, CustomerName, and Date, nested within JSON brackets in a way that represents multiple rows of information.

We could have also used a custom text format with labels and data, like the below.

```

PizzaName: Veggie Lovers
PizzaPrice: 10.99
CustomerName: Bob Smith
Date: 10/21/2022
PizzaName: Veggie Lovers
PizzaPrice: 9.99
CustomerName: Elaine Yang
Date: 11/1/2022
PizzaName: Plain
PizzaPrice: 8.99
CustomerName: Prasad Santhana
Date: 11/12/2022
PizzaName: Veggie Lovers
PizzaPrice: 10.59
CustomerName: David Wofe
Date: 11/13/2022
PizzaName: Plain
PizzaPrice: 9.99
CustomerName: Batman
Date: 11/27/2022

```

The four attributes -- PizzaName, PizzaPrice, CustomerName, and Date – are repeated for each row of information.

We could also have used binary, fixed or variable width flatfile formats, or any other format.

If we write an application to access the table (we'll call it TableApp), and another to access the file (we'll call it FileApp), we can spot one major difference at a high level. FileApp must know the full path to the file, such as C:\PizzaShop.xml, C:\PizzaShop.json, C:\PizzaShop.txt. If the file is moved to a different directory, drive, or machine, the application may break. On the other hand, if the database decides to move its data file, it won't affect the TableApp. TableApp will continue to access the PizzaShop table through the database and the database can move the file around where it needs to.

We say that the relational database is *structurally independent* because applications do not rely on the *location* of the file (as explained previously), in addition to the *structure* of the file. For example, if we comma separate the labels instead of putting them on new lines, the new format could look like this:

PizzaName: Veggie Lovers, PizzaPrice: 10.99, CustomerName: Bob Smith, Date: 10/21/2022  
PizzaName: Veggie Lovers, PizzaPrice: 9.99, CustomerName: Elaine Yang, Date: 11/1/2022

This will undoubtedly break FileApp, because it was expecting to find four lines for each record, and now there is only one line per record. However, if the database adjusts how it stores the data in its data file, it will not break TableApp. TableApp does not depend upon the structure of a file; it only depends upon the structure of the table itself. It does not affect TableApp if the database changes file representation. There are many other scenarios where changing the structure of a file will break FileApp, but will not break TableApp, and the scenario above gives you the basic idea. FileApp depends upon the location and structure of the file used to represent the data; TableApp depends upon the structure of the table, not the file.

Another issue is that of scalability. Locating one record in a file with millions of records could take minutes, hours, or days. However, relational databases can easily support tables with millions of rows in them. Applications can typically pull back information in less than a second out of such tables, assuming they are properly designed and indexed. So, FileApp might take a long time to pull back one record out of millions, while TableApp can pull it back in less than a second.

Another issue is that of security. File systems support security on the file itself, but not on the data within the file. Someone can be given read or write access to the entire file, but not pieces of the file. Relational databases on the other hand support row and field level security natively. People can be granted access to specific rows, and to specific fields within those rows, if needed. So, access to individual rows can be granted easily to TableApp, but FileApp would need to implement its own complex logic to handle record-level security.