OVERVIEW OF THE LAB

The Structured Query Language (SQL) is the de-facto query language for modern relational database management systems (RDBMS). All major, modern RDBMS support SQL. Database developers, administrators, and even software applications access RDBMS through the use of SQL. Familiarity with SQL is essential for working with and understanding modern RDBMS.

The first objective of this lab is to provide you with a practical exercise of using SQL to create and drop a table, and to insert, update, delete, and select a row in a table.

The second objective of this lab is to provide you with a practical exercise of using the Structured Query Language (SQL) to do the following:

- Add a NOT NULL constraint to a table column
- Add a PRIMARY KEY constraint to a table column
- Use dates
- Insert a NULL value into a table row
- Use a WHERE clause to limit the number of rows affected by the SELECT, UPDATE, and DELETE commands
- SELECT only a subset of columns in a result set

REQUIRED SOFTWARE

The examples in this lab will execute in modern versions of Oracle, Microsoft SQL Server, and PostgreSQL as is. If you have been approved to use a different RDBMS, you may need to modify the SQL for successful execution, though the SQL should execute as is if your RDBMS is ANSI compliant.

The screenshots in this lab display execution of SQL in the default SQL clients supported in the course – Oracle SQL Developer, SQL Server Management Studio, and pgAdmin. You are welcome, however, to use a SQL client other than these defaults if you prefer.

PREPARING FOR THE LAB

You will need to install a RDBMS prior to completing this lab. If you are using Oracle, it is highly recommended that you create and login as a non-system user, to avoid damaging the database. You can create a user with the following commands:

```
CREATE USER username IDENTIFIED BY password DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;
GRANT connect, resource TO username;
```

You will then be able to login as the new user.

If you are using Microsoft SQL Server, it is highly recommended that you create and use a database other than the Master database. You can do so with the following commands:

```
CREATE DATABASE database_name;
GO;
USE database_name;
```

If you are using PostgreSQL you can use the UI wizard to create a database or use the following script from the default databse created at time of installation:

```
CREATE DATABASE database_name;
```

SAVING YOUR DATA

If you choose to perform portions of the lab in different sittings, it is important to *commit* your data at the end of each session. This way, you will be sure to make permanent any data changes you have made in your current session, so that you can resume working without issue in your next session. To do so, simply issue this command:

```
COMMIT;
```

We will learn more about committing data in future weeks. For now, it is sufficient to know that data changes in one session will only be visible only in that session, unless they are committed, at which time the changes are made permanent in the database.

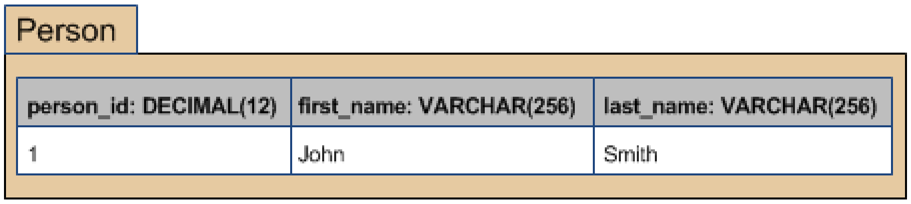
COMPLETING YOUR LAB

Use the submission template provided in the assignment inbox to complete this lab.

SECTION ONE

OVERVIEW

In Section One we will create the Person table illustrated below:



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 data types specified in the diagram above.

We will create this table, try out some data manipulation commands, and ultimately remove the table.

STEPS

1. Execute the following command to create the Person table. Make sure to type the command exactly as illustrated, including spaces, parentheses, and newlines.

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

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 identifer 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.

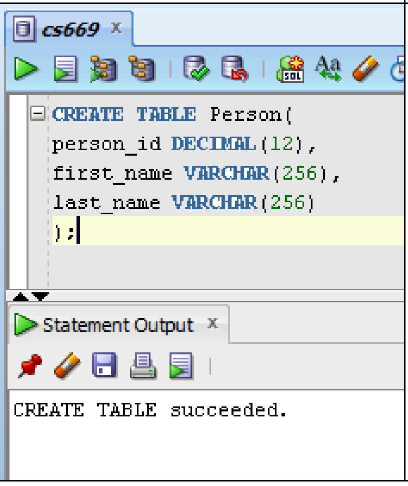
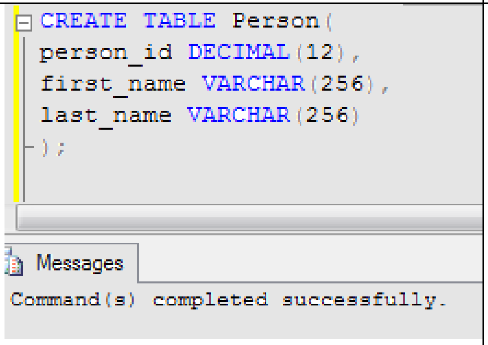
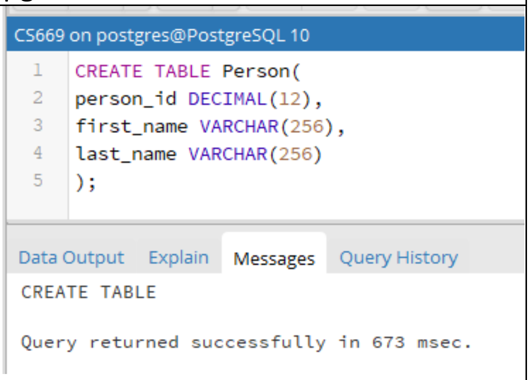
The VARCHAR datatype for the first_name column indicates that we are restricting that column’s values to character sequences, for example, “abcd”. The “(256)” means that the maximum number of bytes is 256. Some RDBMS allow you to change the meaning of this number from bytes to characters, so that the 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.

2. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	Microsoft SQL Server	pgAdmin

 `CREATE TABLE Person(`
`person_id DECIMAL(12),`
`first_name VARCHAR(256),`
`last_name VARCHAR(256)`
`);`
`CREATE TABLE succeeded.`
	 `CREATE TABLE Person(`
	`person_id DECIMAL(12),`
	`first_name VARCHAR(256),`
	`last_name VARCHAR(256)`
	`);`
	`Command(s) completed successfully.`
		 CS669 on postgres@PostgreSQL 10
		Data Output
		Explain
		Messages
		Query History `CREATE TABLE Person(`
		`person_id DECIMAL(12),`
		`first_name VARCHAR(256),`
		`last_name VARCHAR(256)`
		`);`
		`CREATE TABLE`
		`Query returned successfully in 673 msec.`

A complete screenshot contains the SQL command and the result of the SQL command. The screenshot will be 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.

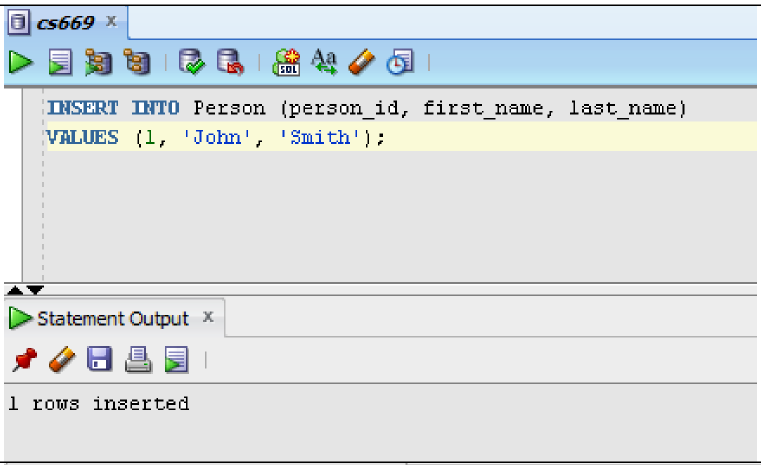
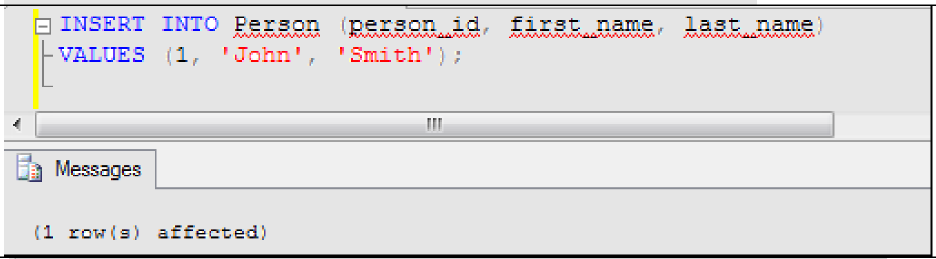
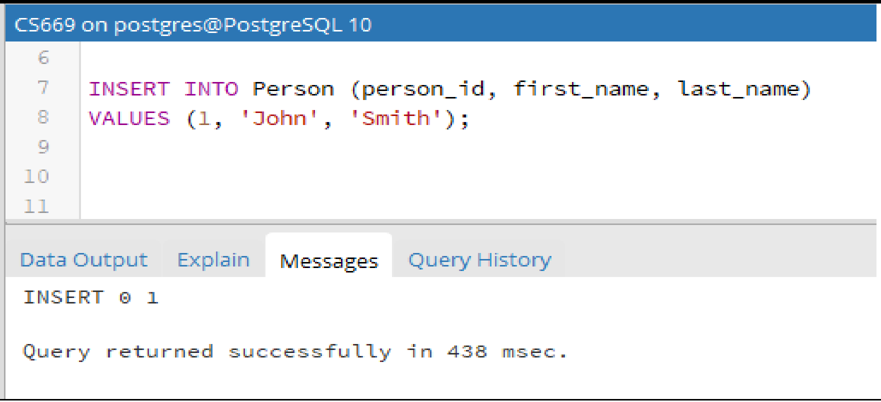
3. Insert a row with values person_id = 1,   first_name = John, and last_name = Smith by executing the following command.

```
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 to 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 RDMBS 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 (`).

4. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

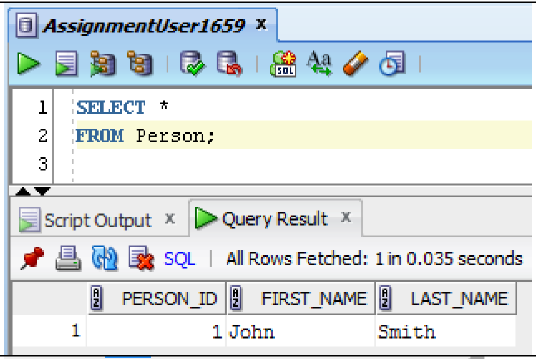
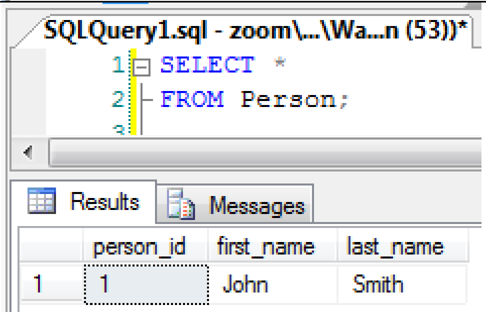
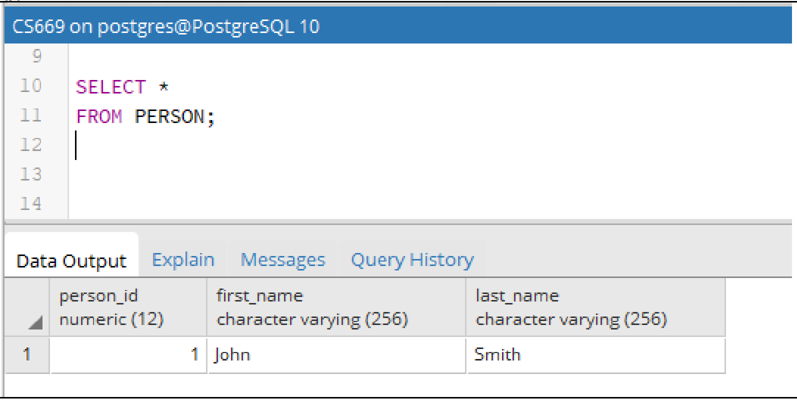
Oracle SQL Developer	 `INSERT INTO Person (person_id, first_name, last_name) VALUES (1, 'John', 'Smith');` Statement Output: `1 rows inserted`
Microsoft SQL Server Management Studio	 `INSERT INTO Person (person_id, first_name, last_name) VALUES (1, 'John', 'Smith');` Messages: `(1 row(s) affected)`
pgAdmin	 CS669 on postgres@PostgreSQL 10 `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.`

5. Select all rows in the 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.

6. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	
Microsoft SQL Server Management Studio	
pgAdmin	

Oracle SQL Developer:

```
SELECT *
FROM Person;
```

All Rows Fetched: 1 in 0.035 seconds

	PERSON_ID	FIRST_NAME	LAST_NAME
1	1	John	Smith

Microsoft SQL Server Management Studio:

```
SELECT *
FROM Person;
```

	person_id	first_name	last_name
1	1	John	Smith

pgAdmin:

```
SELECT *
FROM PERSON;
```

	person_id numeric (12)	first_name character varying (256)	last_name character varying (256)
1	1	John	Smith

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

7. Update the last name of John Smith from “Smith” to “Glass” by executing the following command:

```
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'
```

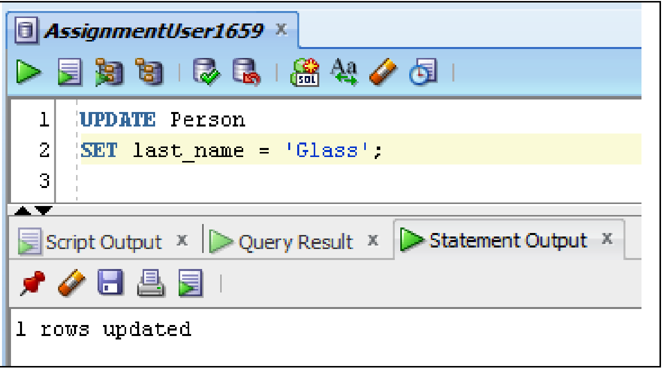
we are instructing the RDBMS to set the value of the last_name column to the value “Glass”. If we had instead used the phrase:

```
first_name = 'Jane'
```

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

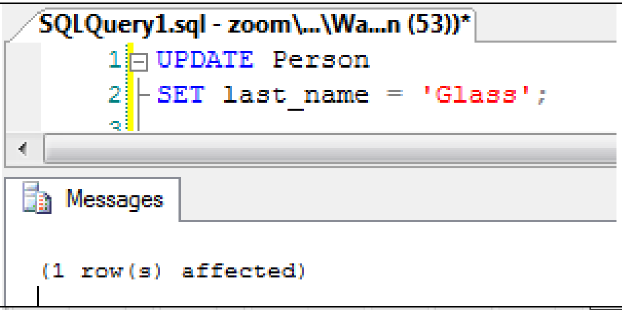
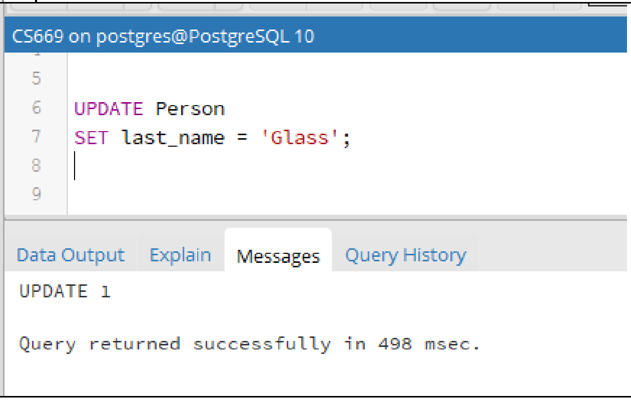
The UPDATE command above updates all rows in the Person table. If there were to be more than one row in the Person table, the last_name values for *all* of the rows would be updated. We will learn next week how to limit updates to a specific row or group of rows.

8. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	

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

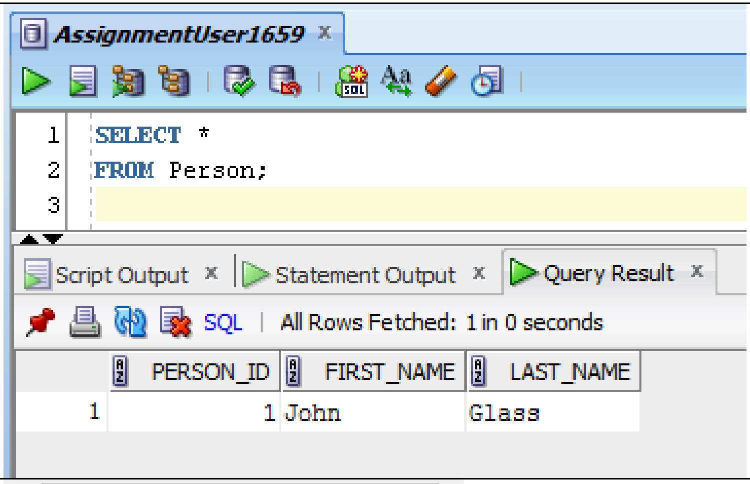
```
1 rows updated
```

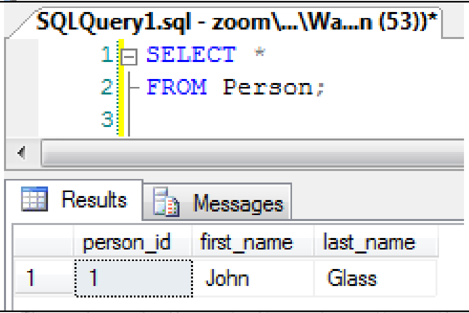
Microsoft SQL Server Management Studio	 SQLQuery1.sql - zoom\...\Wa...n (53))* `UPDATE Person` `SET last_name = 'Glass';` Messages `(1 row(s) affected)`
pgAdmin	 CS669 on postgres@PostgreSQL 10 `UPDATE Person` `SET last_name = 'Glass';` Data Output, Explain, Messages, Query History `UPDATE 1` `Query returned successfully in 498 msec.`

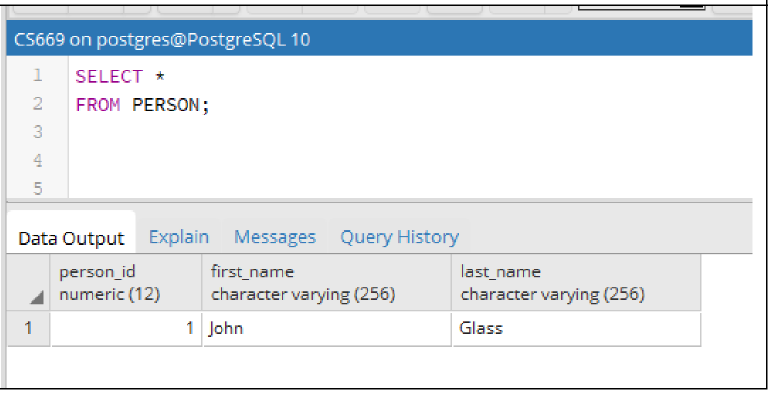
9. In order to see the results of our update, let us again select all rows in the table by executing the following command.

```
SELECT *
FROM Person;
```

10. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	 AssignmentUser1659
	`SELECT *`
	`FROM Person;`
	Script Output · Statement Output · Query Result
	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))*
	`SELECT *`
	`FROM Person;`
	Results · Messages
	person_id | first_name | last_name
	1 | 1 | John | Glass

pgAdmin	 CS669 on postgres@PostgreSQL 10
	`SELECT *`
	`FROM PERSON;`
	Data Output · Explain · Messages · Query History
	person_id numeric (12) | first_name character varying (256) | last_name character varying (256)
	1 | 1 | John | Glass

Notice that the last name is now “Glass”.

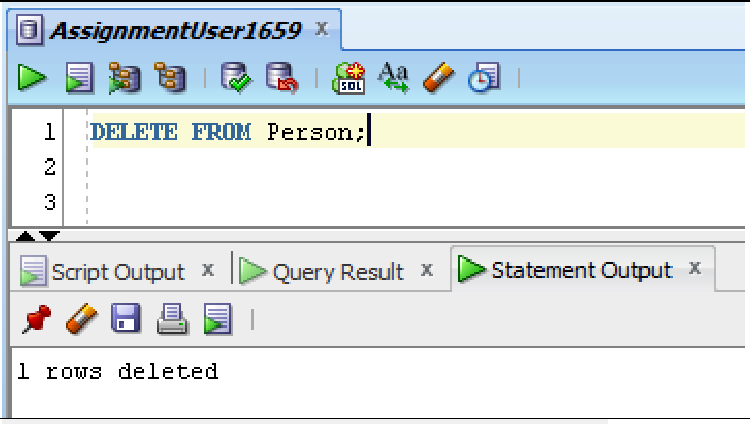
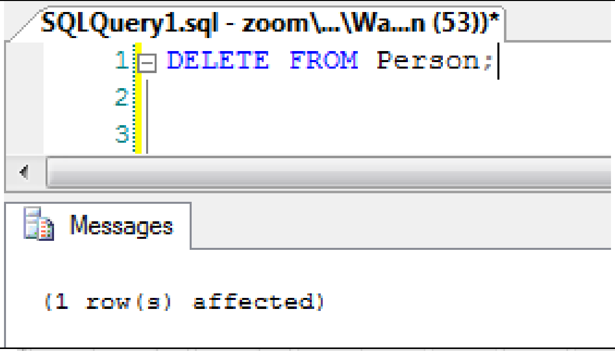
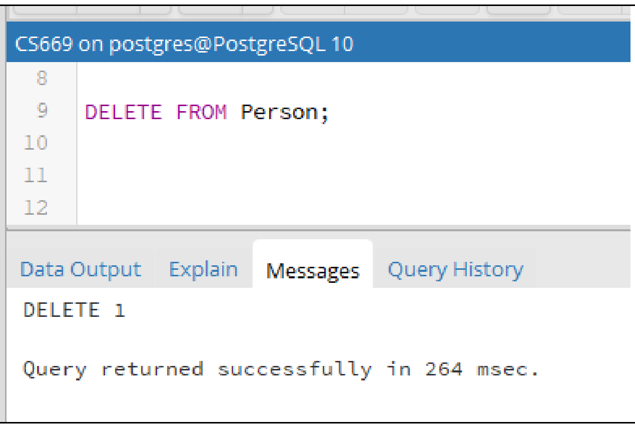
11. Next, we are going to remove all rows from the table. We do so with the following command.

```
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 next week how to limit a delete to a specific row or group of rows.

12. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

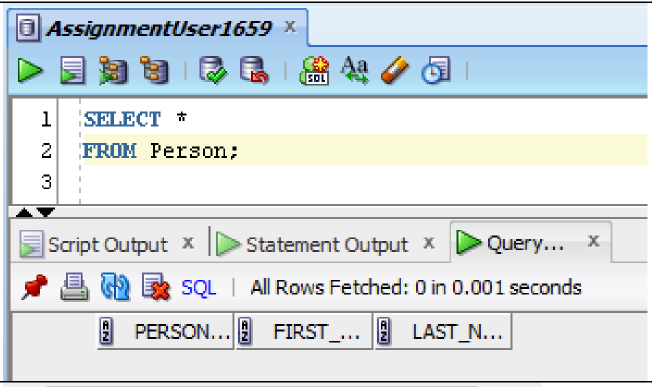
Oracle SQL Developer	 AssignmentUser1659 `DELETE FROM Person;` Script Output, Query Result, Statement Output `1 rows deleted`
Microsoft SQL Server Management Studio	 SQLQuery1.sql - zoom\...\Wa...n (53))* `DELETE FROM Person;` Messages `(1 row(s) affected)`
pgAdmin	 CS669 on postgres@PostgreSQL 10 `DELETE FROM Person;` Data Output, Explain, Messages, Query History `DELETE 1` `Query returned successfully in 264 msec.`

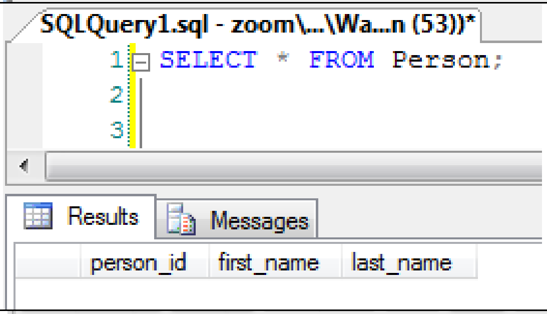
Notice that each RDBMS indicate the number of rows that were deleted.

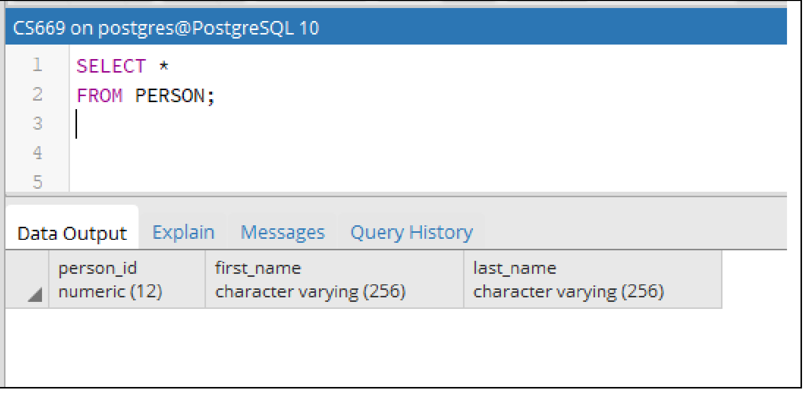
13. In order to see that the table now has no rows, let us again select all rows in the table by executing the following command.

```
SELECT *
FROM Person;
```

14. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	 AssignmentUser1659
	`SELECT *`
	`FROM Person;`
	Script Output · Statement Output · Query...
	SQL · All Rows Fetched: 0 in 0.001 seconds
	PERSON... · FIRST_... · LAST_N...

Microsoft SQL Server Management Studio	 SQLQuery1.sql - zoom\...\Wa...n (53))* `SELECT * FROM Person;` Results · Messages person_id · first_name · last_name

pgAdmin	 CS669 on postgres@PostgreSQL 10
	`SELECT *`
	`FROM PERSON;`
	Data Output · Explain · Messages · Query History
	person_id numeric (12) · first_name character varying (256) · last_name 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.

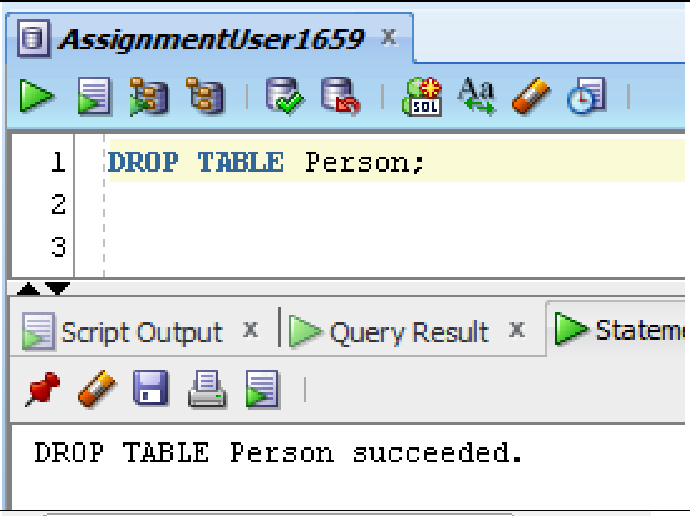
15. So that we know how to remove tables that we create, let us try that now.

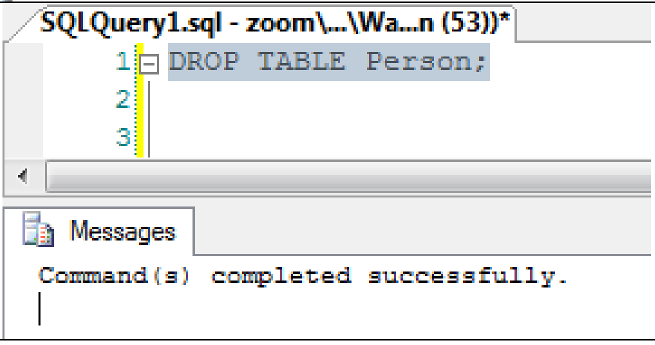
```
DROP TABLE Person;
```

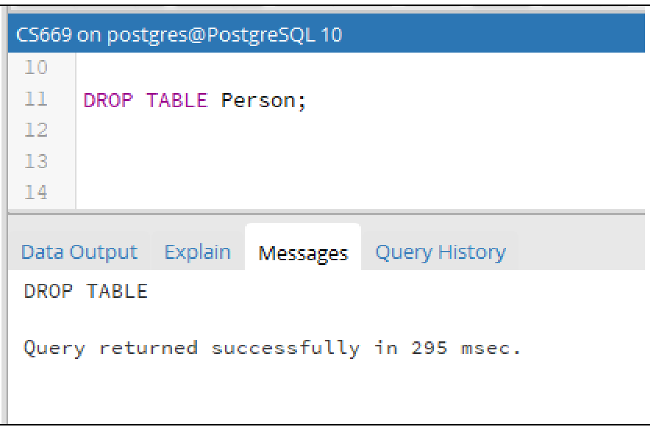
The technical word for removing a table is “drop”, and so the first two keywords DROP TABLE instruct the RDBMS to remove a table. The third word, “Person”, is the name of the table to be dropped, in this case, the Person table.

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 dropped, and the data cannot always be recovered if you later decide that you want to keep the data.

16. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	 AssignmentUser1659
	`DROP TABLE Person;`
	Script Output · Query Result · Statem
	`DROP TABLE Person succeeded.`

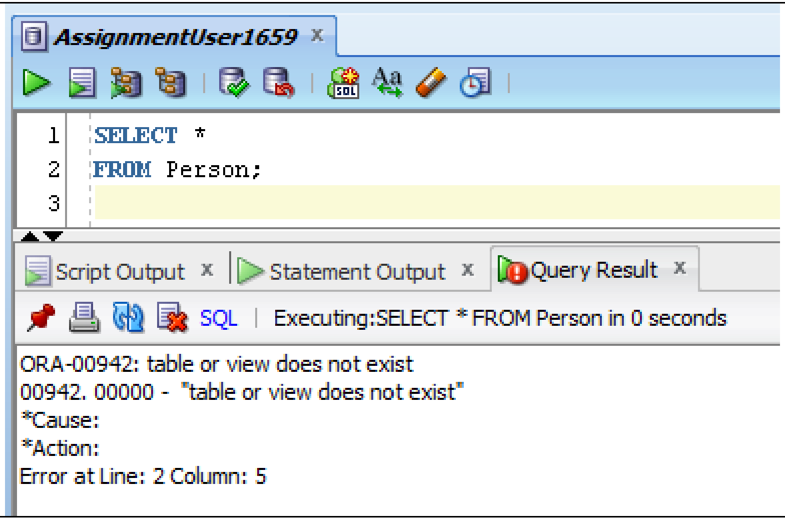
Microsoft SQL Server Management Studio	 SQLQuery1.sql - zoom\...\Wa...n (53))*
	`DROP TABLE Person;`
	Messages
	`Command(s) completed successfully.`

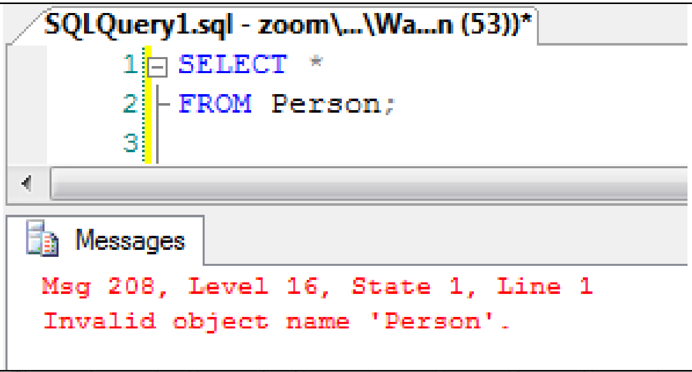
pgAdmin	 CS669 on postgres@PostgreSQL 10
	`DROP TABLE Person;`
	Data Output Explain Messages Query History
	`DROP TABLE`
	`Query returned successfully in 295 msec.`

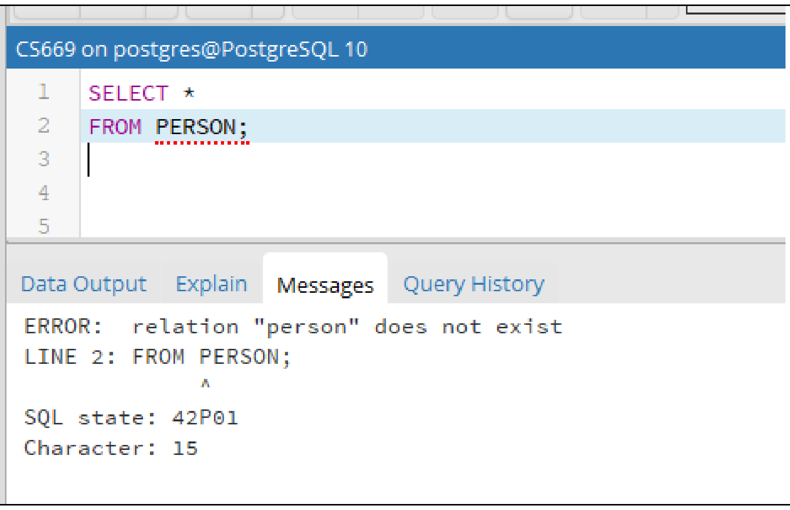
17. For good measure, let us attempt to retrieve data from the Person table we just deleted. Of course, now that the Person table has been dropped, the command will not be able to retrieve the results. It is helpful however to become accustomed to the error message indicating that the table does not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

```
SELECT *
FROM Person;
```

18. Capture a screenshot of the command the result of its execution. Below is a sample screenshot of the command execution in each RDBMS.

Oracle SQL Developer	 AssignmentUser1659
	`SELECT *`
	`FROM Person;`
	Script Output Statement Output Query Result
	SQL | Executing:SELECT * FROM Person in 0 seconds
	`ORA-00942: table or view does not exist`
	`00942. 00000 -  "table or view does not exist"`
	`*Cause:`
	`*Action:`
	`Error at Line: 2 Column: 5`

Microsoft SQL Server Management Studio	 SQLQuery1.sql - zoom\...\Wa...n (53))*
	 `SELECT *`
	 `FROM Person;`
	 Messages
	 `Msg 208, Level 16, State 1, Line 1`
	 `Invalid object name 'Person'.`

pgAdmin	 CS669 on postgres@PostgreSQL 10
	 `SELECT *`
	 `FROM PERSON;`
	 Data Output / Explain / Messages / Query History
	 `ERROR:  relation "person" does not exist`
	 `LINE 2: FROM PERSON;`
	 `             ^`
	 `SQL state: 42P01`
	 `Character: 15`

Oracle Error Messages:
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 Error Messages:

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 intepret 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, "Somehwere 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 (pgAdmin):

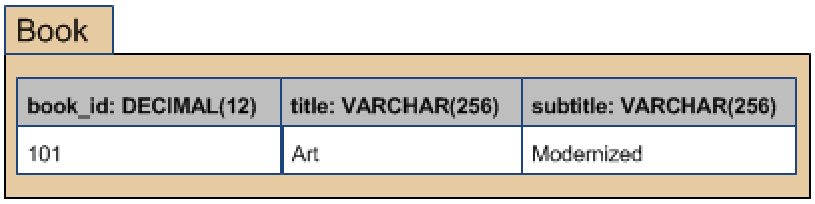
Using the provided error message along with the SQL command we are able to diagnose the issue.

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

OVERVIEW

The goal of Section Two is to apply what is learned in Section 1 to create and populate the following table:



Book

book_id: DECIMAL(12)	title: VARCHAR(256)	subtitle: VARCHAR(256)
101	Art	Modernized

This Book table has three columns -- book_id, title, and subtitle -- with data types specified in the diagram above.

STEPS

19. Execute the command to create the Book table. Capture a screenshot of the command and the result of its execution.
20. Execute the command to insert a row with values book_id = 101, title = Art, and subtitle = Modernized. Capture a screenshot of the command and the result of its execution.
21. To view the row just inserted, execute the command to select all rows in the table. Capture a screenshot of the command and the result of its execution.
22. Update the subtitle so that it has a value of “Revisited” instead of “Modernized”. Capture a screenshot of the command and the result of its execution.
23. To see the results of the update, select all rows in the table. Capture a screenshot of the command and the result of its execution.
24. Remove all rows from the table. Capture a screenshot of the command and the result of its execution.
25. To see the results of removing, select all rows from the table. Capture a screenshot of the command and the result of its execution.
26. Drop the table. Capture a screenshot of the command and the result of its execution.
27. Attempt to select all rows from the table. Capture a screenshot of the command and the result of its execution.
28. Explain how you would use the error message resulting from Step 27, in conjunction with the SQL command, to diagnose the issue.

SECTION THREE

OVERVIEW

In Section Three, we will work with the Camera table illustrated below. Note that the bolded columns represent those with a NOT NULL constraint.

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

STEPS

29. Execute the following command to create the Camera table. Make sure to type the command exactly as illustrated, including spaces, parentheses, and newlines.

```sql
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 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.

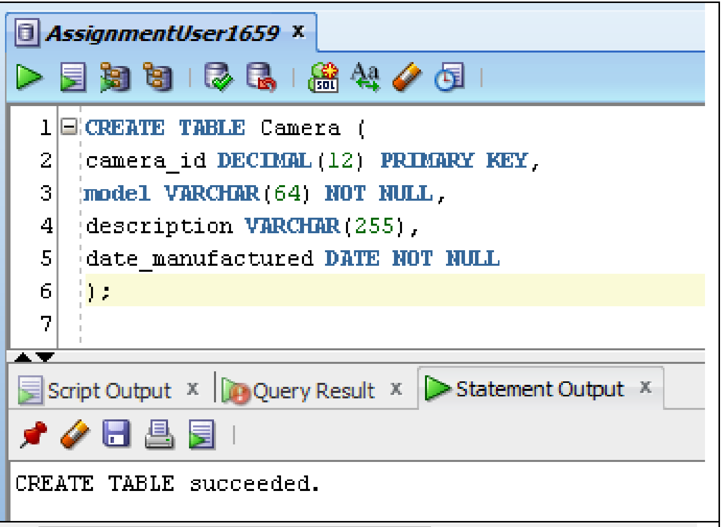
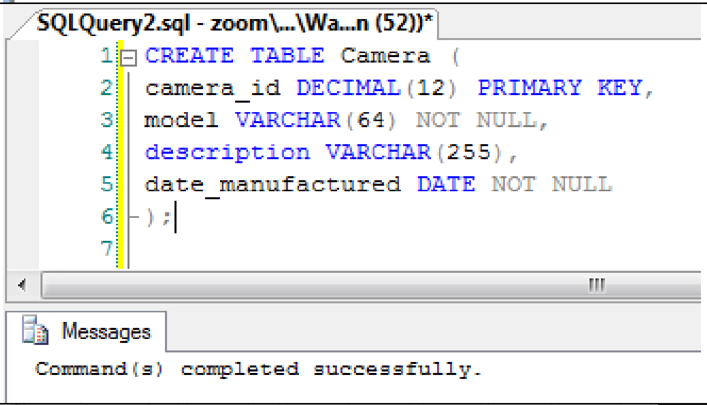
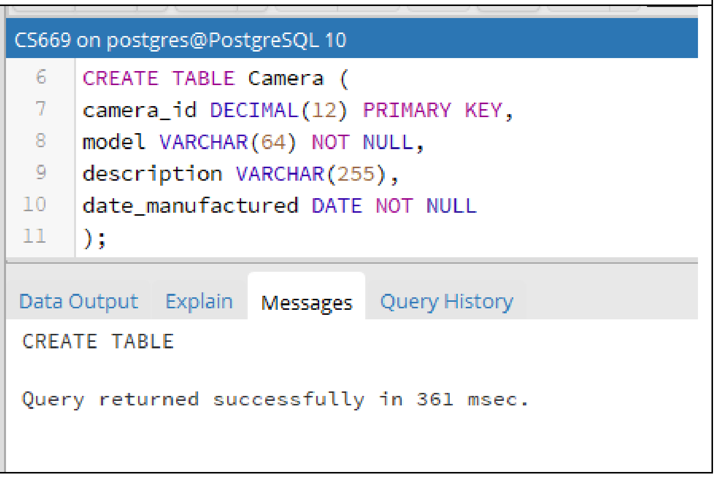
Let us briefly review the concept of NULL. When we assign a datatype to a column, we restrict the legal values for that column. For example, a VARCHAR datatype means that we allow sequences of characters for the column. A DATE datatype means that we allow date values for the column. But, what if we want 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.

30. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer	
Microsoft SQL Server Management Studio	
pgAdmin	

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

```
CREATE TABLE succeeded.
```

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

```
Command(s) completed successfully.
```

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

```
CREATE TABLE

Query returned successfully in 361 msec.
```

31. Execute the following command to insert a row with the values listed.
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)
        );
```

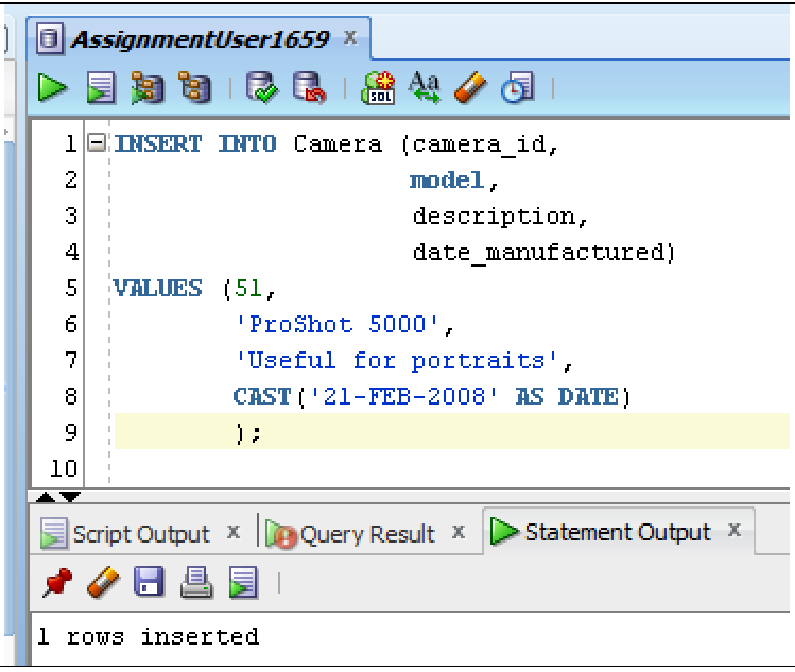
You have already seen examples of inserting numbers and character sequences, but inserting dates may be new to you and is worth a more detailed look. If we want to insert a number, we simply type the number in the format we are used to, and the RDBMS knows that what we typed is a literal number. For 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 21st, 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. Alternatively, temporarily changing your Region and Language to "United States" may resolve the issue.

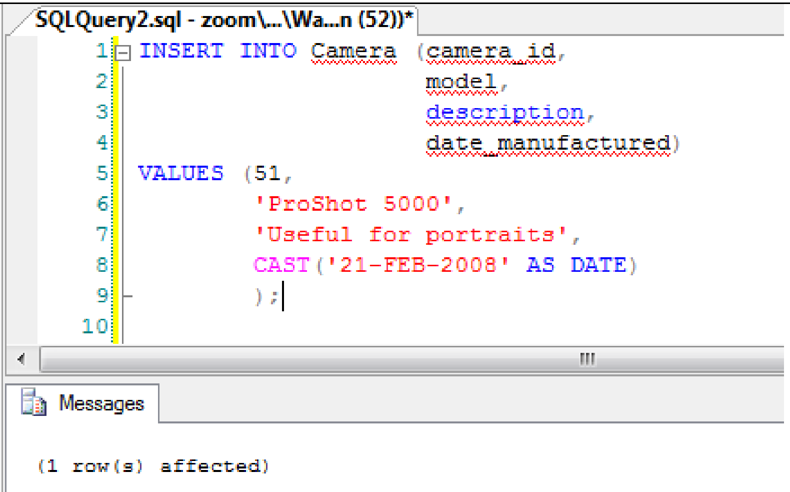
32. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer

Oracle SQL Developer	

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

```
1 rows inserted
```

Microsoft SQL Server Management Studio



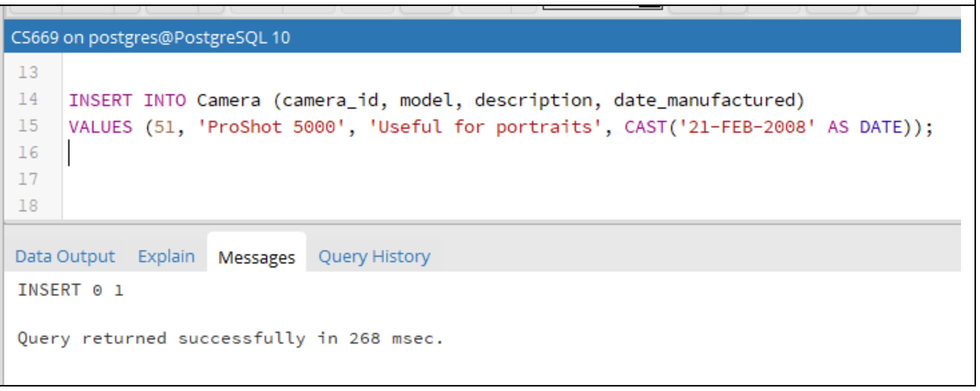
SQLQuery2.sql - zoom\...\Wa...n (52))*

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

Messages

```
(1 row(s) affected)
```

pgAdmin



CS669 on postgres@PostgreSQL 10

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

```
INSERT 0 1

Query returned successfully in 268 msec.
```

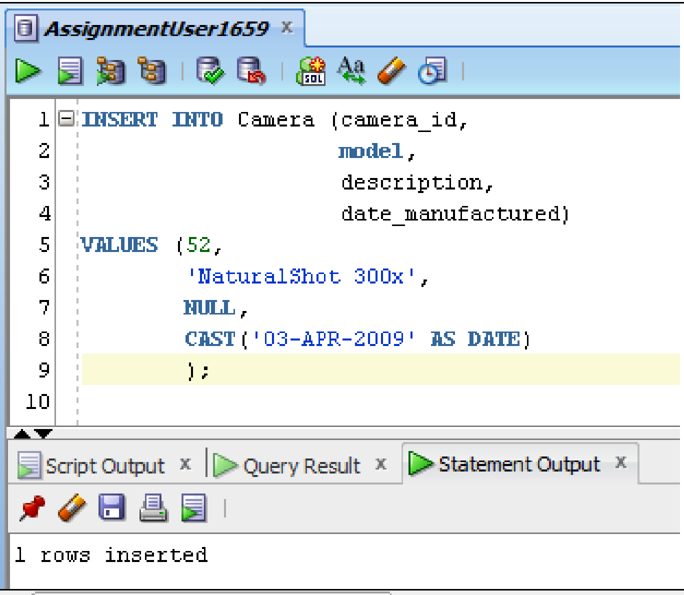
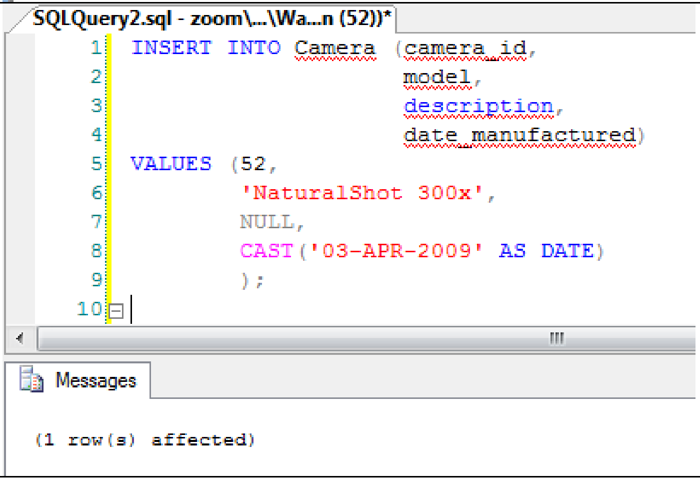
We can see that each RDBMS notifies the number of rows affected/inserted.

33. Execute the following command to insert a row with the values listed.
    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.

34. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer

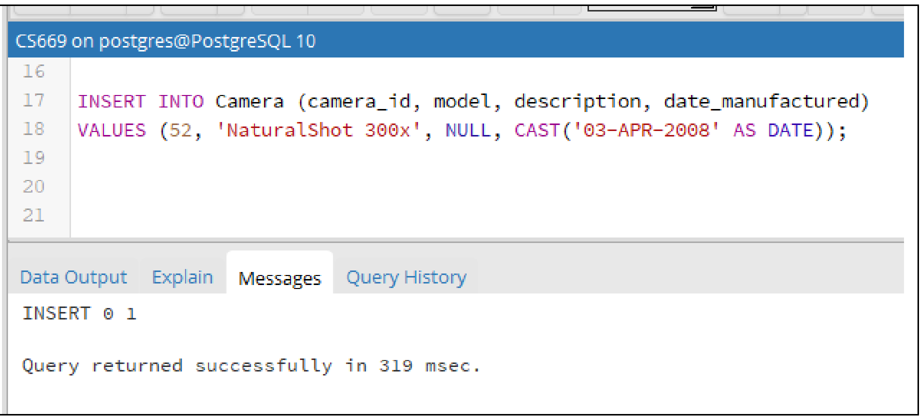
Oracle SQL Developer	
Microsoft SQL Server Management Studio	

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

```
1 rows inserted
```

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

```
(1 row(s) affected)
```

pgAdmin	

```
INSERT INTO Camera (camera_id, model, description, date_manufactured)
VALUES (52, 'NaturalShot 300x', NULL, CAST('03-APR-2008' AS DATE));
```

```
INSERT 0 1

Query returned successfully in 319 msec.
```

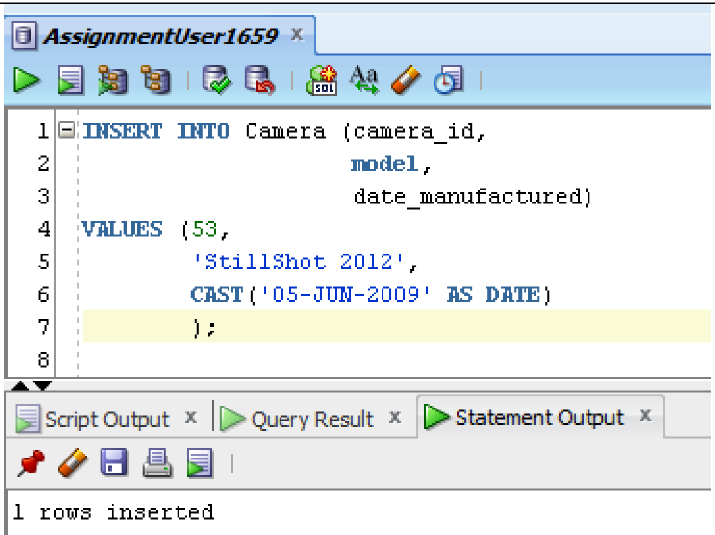
35. The next command we will execute indicates no value for the description column in a different way.
    camera_id = 53
    model = StillShot 2012
    date_manufactured = 05-JUN-2009

```
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.

36. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

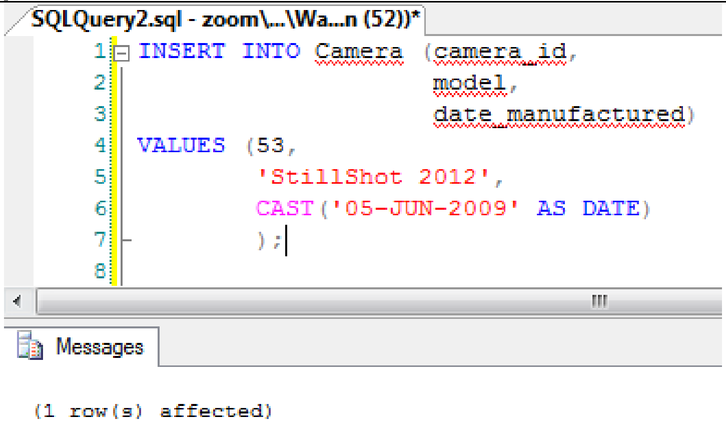
Oracle SQL Developer



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

```
1 rows inserted
```

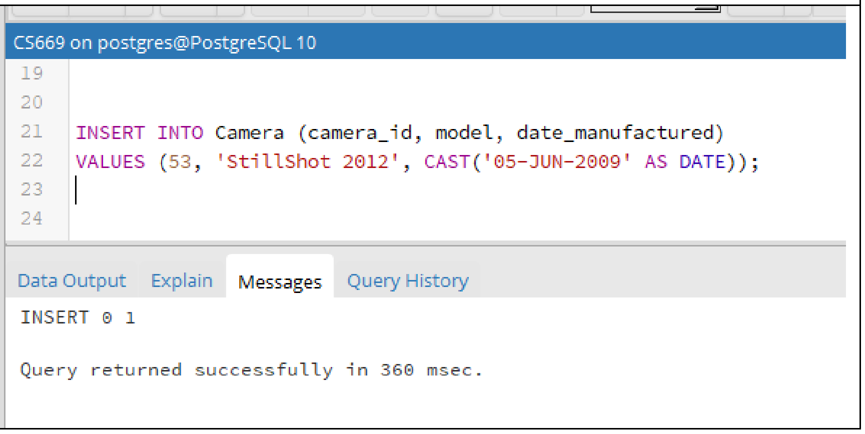
Microsoft SQL Server Management Studio



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

```
(1 row(s) affected)
```

pgAdmin



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

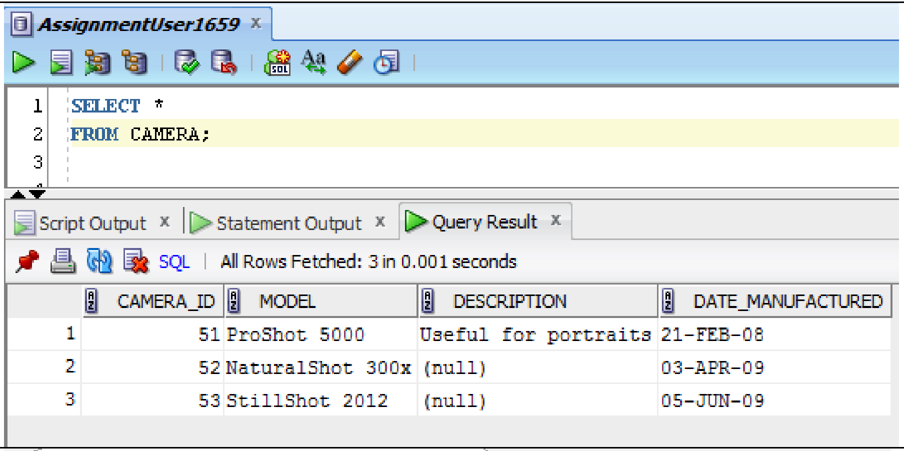
```
INSERT 0 1

Query returned successfully in 360 msec.
```

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

```
SELECT *
FROM CAMERA;
```

38. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer



AssignmentUser1659

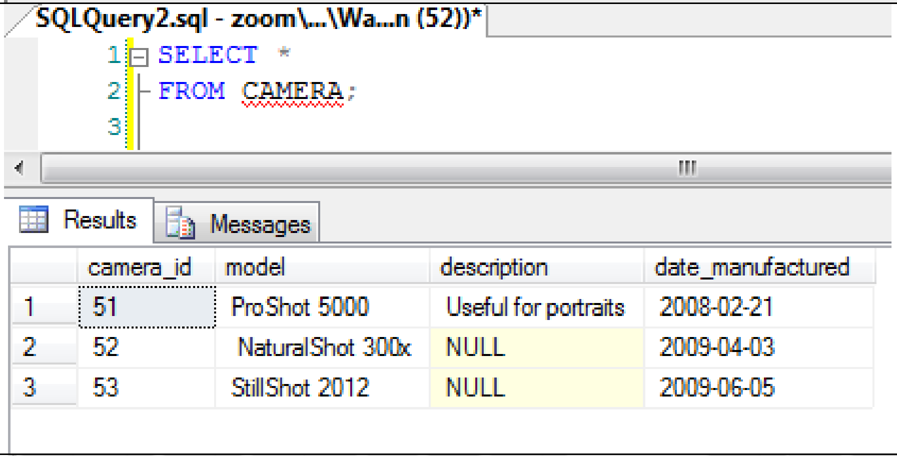
```
SELECT *
FROM CAMERA;
```

Script Output | Statement Output | Query Result

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



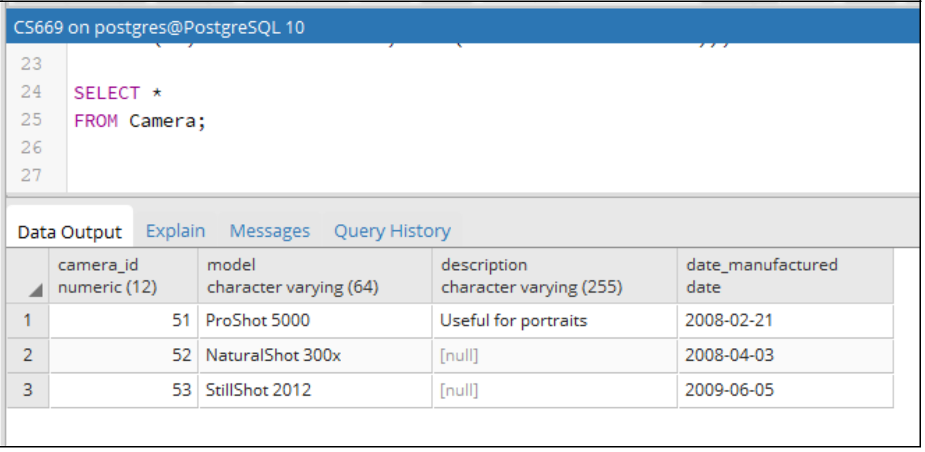
SQLQuery2.sql - zoom\...\Wa...n (52))*

```
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

```
SELECT *
FROM Camera;
```

Data Output | Explain | Messages | Query History

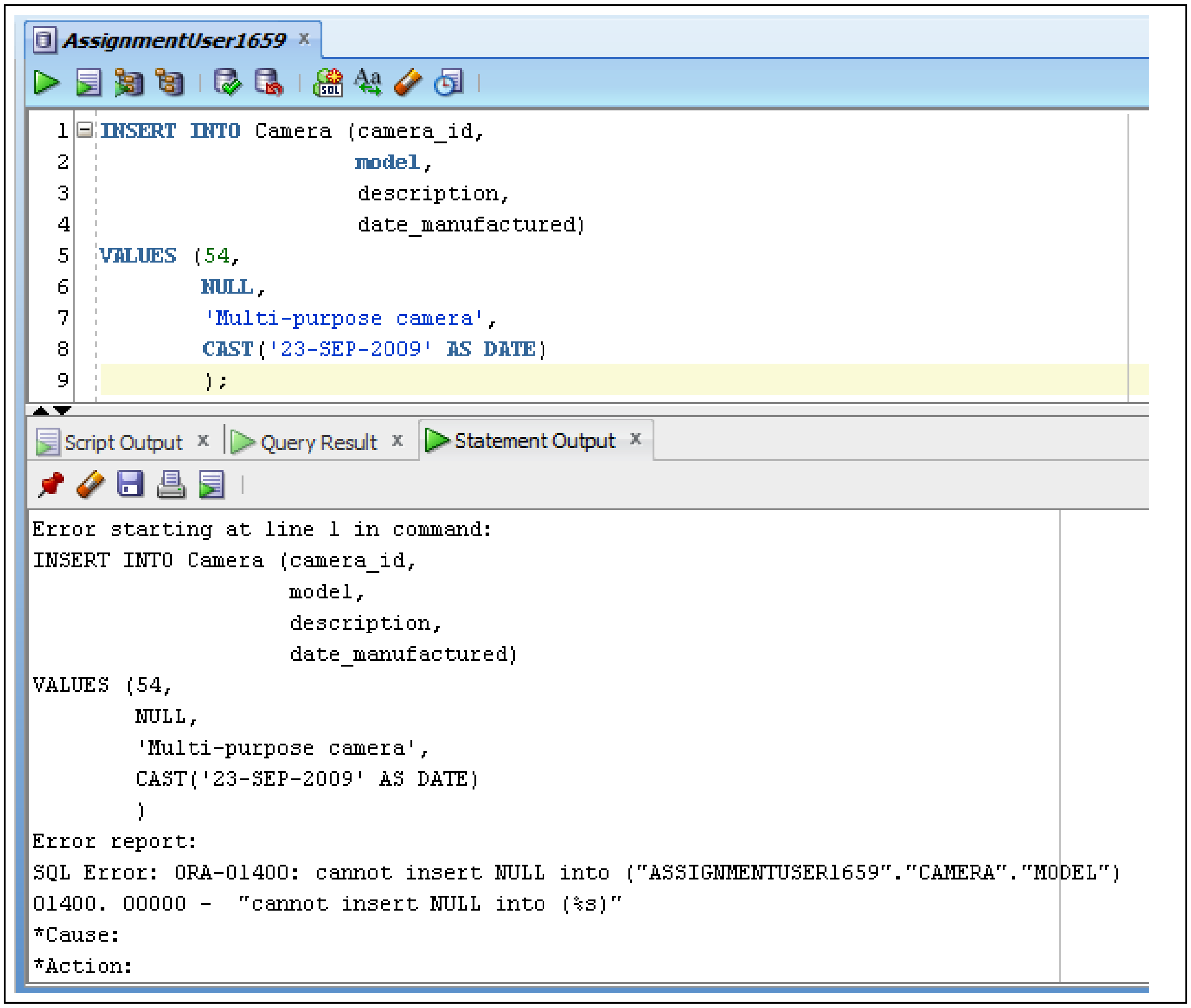
```
  camera_id     model                   description              date_manufactured
  numeric (12)  character varying (64)  character varying (255)  date
1            51 ProShot 5000            Useful for portraits     2008-02-21
2            52 NaturalShot 300x        [null]                   2008-04-03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

39. So that we can familiarize ourselves with another common error message, let us attempt insert a row without a value for the model column. The insertion will fail because a NOT NULL constraint is present on the model column.
camera_id = 54
model = NULL
description = Multi-purpose camera
date_manufactured = 23-SEP-2009

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

40. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer



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

```
Error starting at line 1 in command:
INSERT INTO Camera (camera_id,
                    model,
                    description,
                    date_manufactured)
VALUES (54,
        NULL,
        'Multi-purpose camera',
        CAST('23-SEP-2009' AS DATE)
        )
Error report:
SQL Error: ORA-01400: cannot insert NULL into ("ASSIGNMENTUSER1659"."CAMERA"."MODEL")
01400. 00000 -  "cannot insert NULL into (%s)"
*Cause:
*Action:
```

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.

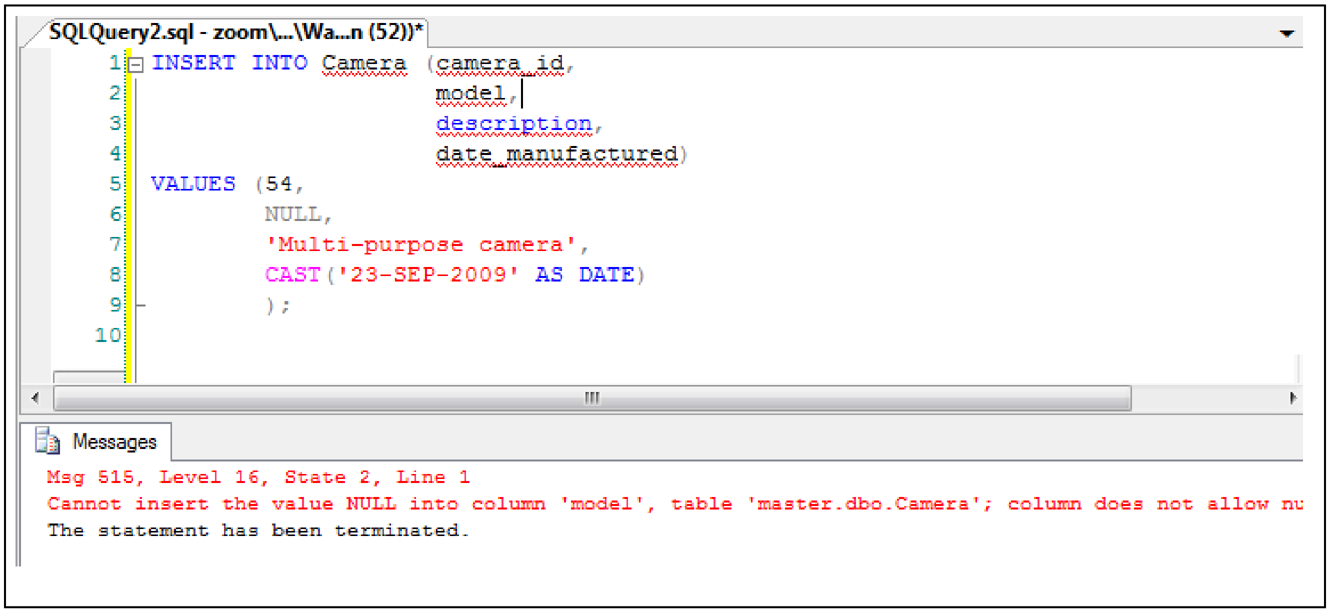
Now, it is unlikely that you chose the same username, and so you will see your own username in the screenshot here, instead of the author's username. What is important to note, however, is that we can also specify a schema name in front of any table name. We use:

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.



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

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

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

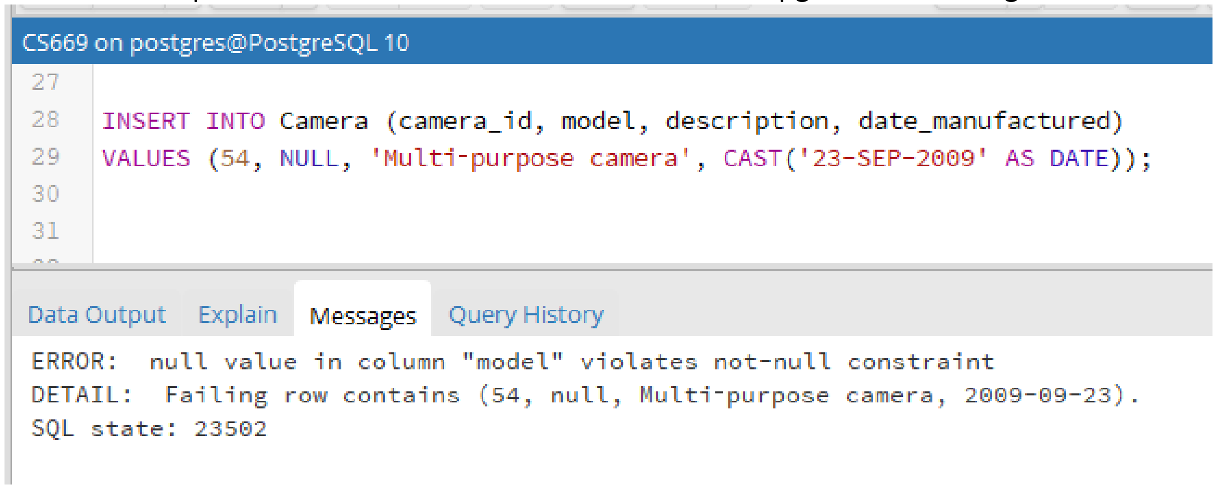
```
--
Msg 515, Level 16, State 2, Line 1
Cannot insert the value NULL into column 'model', table 'master.dbo.Camera';
column does not allow nulls. INSERT fails.
The statement has been terminated.
--
```

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.



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

```
ERROR:  null value in column "model" violates not-null constraint
DETAIL:  Failing row contains (54, null, Multi-purpose camera, 2009-09-23).
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 versions of Oracle and SQL Server may give different error messages than those illustrated above.

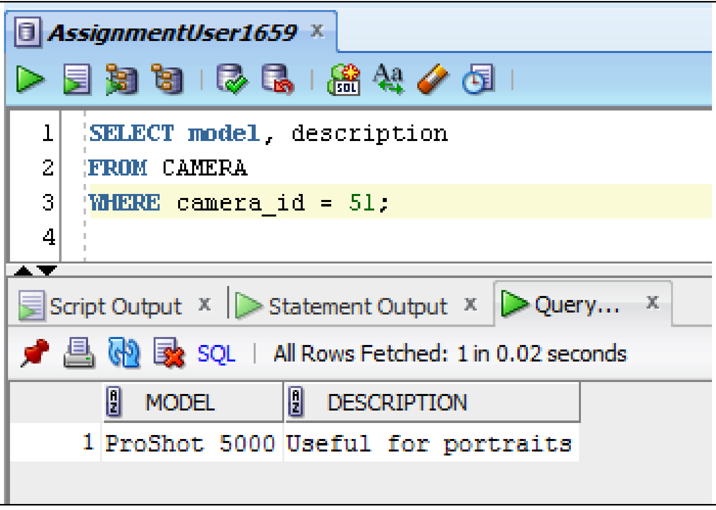
41. Let us practice limiting the number of rows and columns returned by a SELECT statement.

```
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.

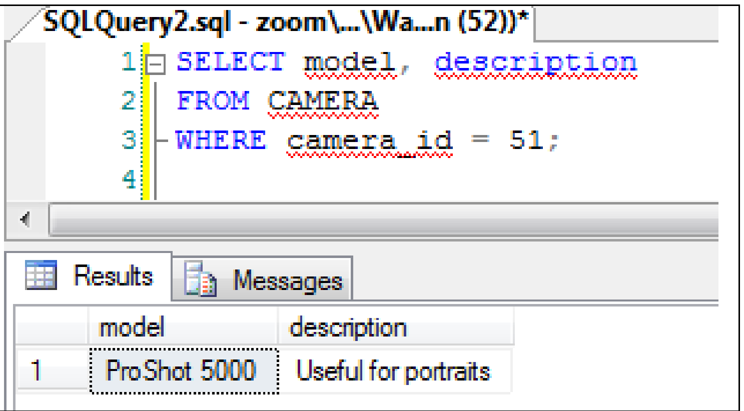
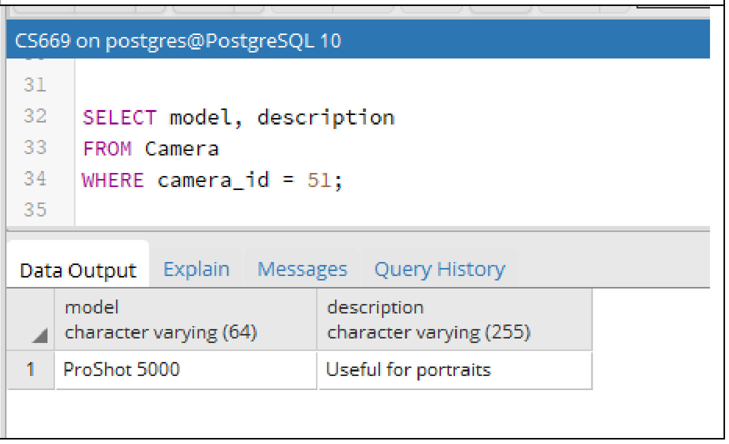
By introducing a WHERE clause at the end of the SELECT statement, and using a Boolean expression that indicates that we only want to see rows where the camera_id value is 51, we have limited the number of rows returned in our result set. Try the command and see what happens.

42. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer	

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

```
All Rows Fetched: 1 in 0.02 seconds

    MODEL         DESCRIPTION
  1 ProShot 5000  Useful for portraits
```

Microsoft SQL Server Management Studio	
pgAdmin	

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

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

Results | Messages

	model	description
1	ProShot 5000	Useful for portraits

CS669 on postgres@PostgreSQL 10

```
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 billon 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 assignment, but if you are curious, you can explore these in the lecture and textbook, and in other resources.

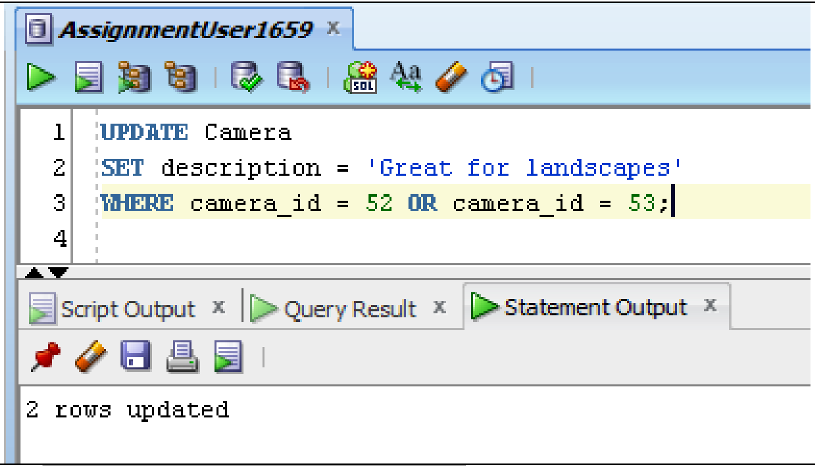
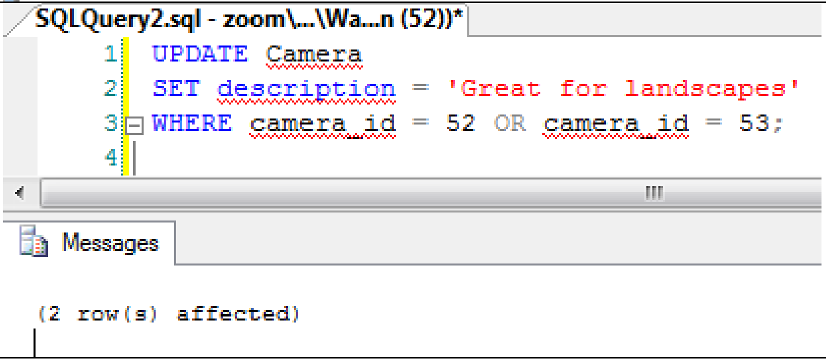
43. The good news is that we do not need to learn three different ways to limit the rows affected in the SELECT, UPDATE, and DELETE FROM commands. We use the same general WHERE clause for all three commands. Imagine that we want to add the same description to both rows that have no description (where 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.

44. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

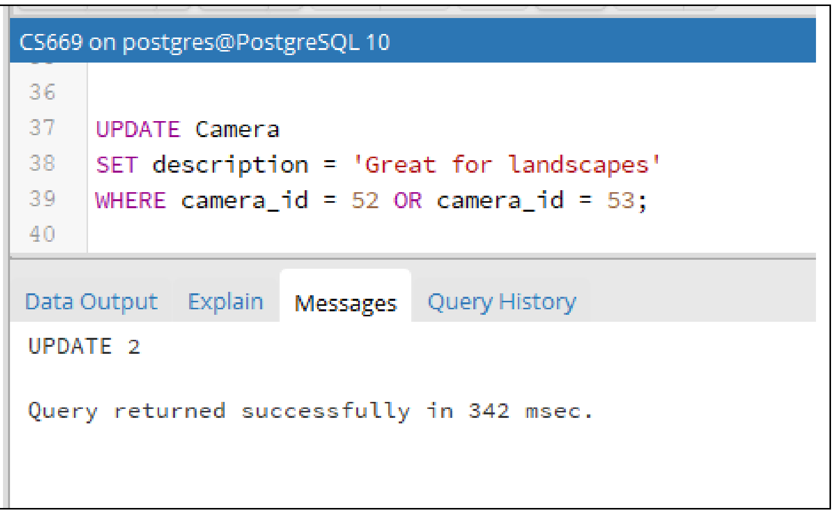
Oracle SQL Developer	
Microsoft SQL Server Management Studio	

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

```
2 rows updated
```

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

```
(2 row(s) affected)
```

pgAdmin	 CS669 on postgres@PostgreSQL 10

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

```
UPDATE 2

Query returned successfully in 342 msec.
```

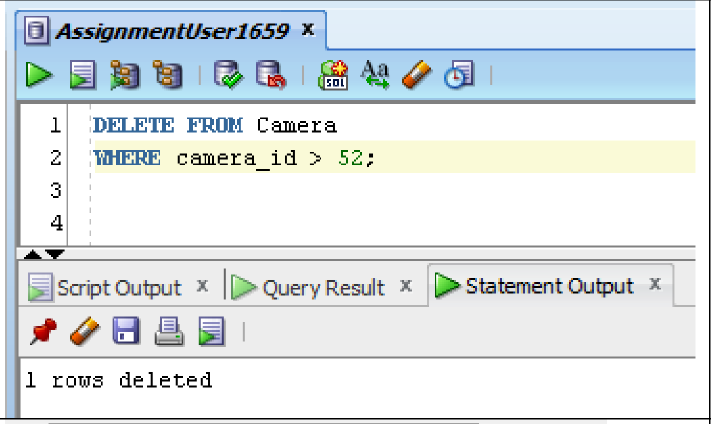
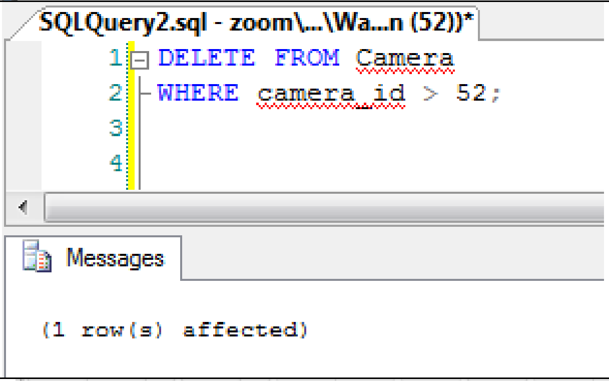
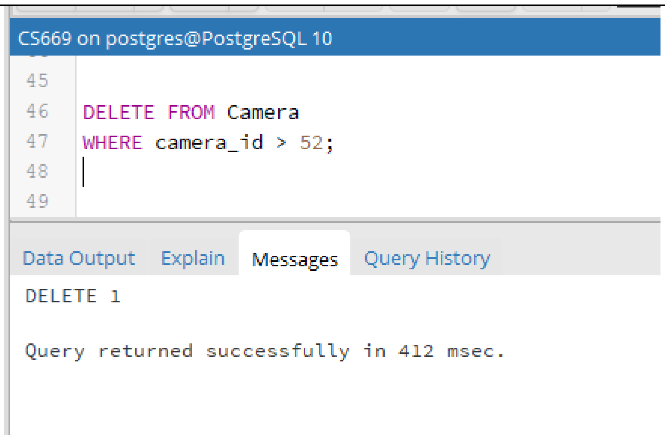
Notice that each screenshot indicate two rows were affected, which is what we expect. We did not update all rows in the table, but only those that matched the condition expressed in the WHERE clause.

45. We can append a general WHERE clause to a DELETE FROM command 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.

46. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

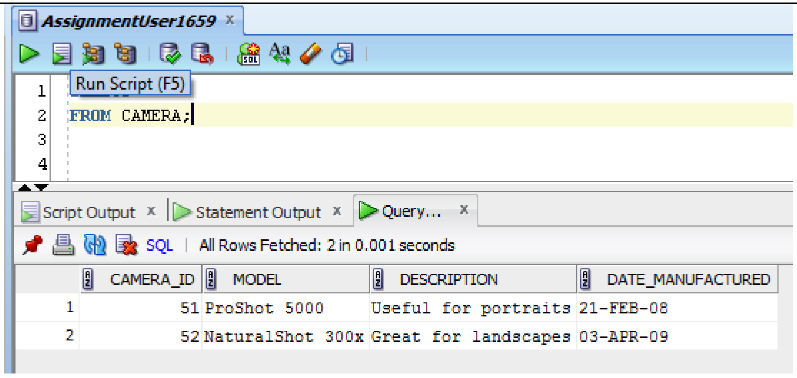
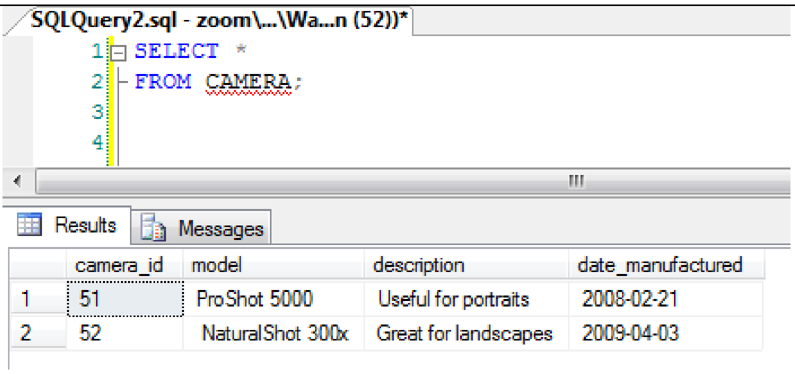
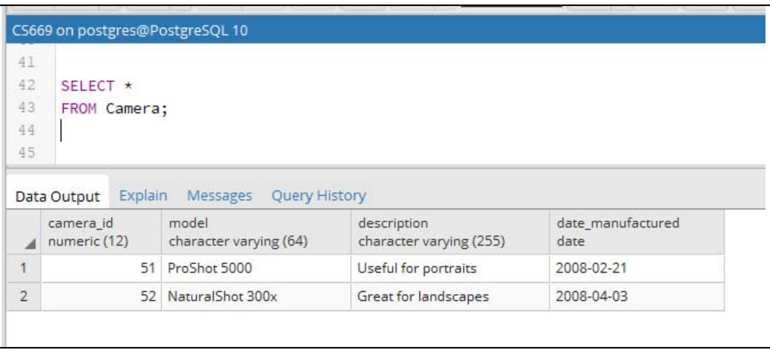
Oracle SQL Developer	 AssignmentUser1659 `DELETE FROM Camera WHERE camera_id > 52;` Script Output, Query Result, Statement Output: `1 rows deleted`
Microsoft SQL Server Management Studio	 SQLQuery2.sql - zoom\...\Wa...n (52))* `DELETE FROM Camera WHERE camera_id > 52;` Messages: `(1 row(s) affected)`
pgAdmin	 CS669 on postgres@PostgreSQL 10 `DELETE FROM Camera WHERE camera_id > 52;` Data Output, Explain, Messages, Query History: `DELETE 1` `Query returned successfully in 412 msec.`

Notice that each screenshot indicates one row was affected.

47. 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.

```
SELECT *
FROM CAMERA;
```

48. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in each RDBMS

Oracle SQL Developer	
Microsoft SQL Server Management Studio	
pgAdmin	

AssignmentUser1659

Run Script (F5)

```
FROM CAMERA;
```

Script Output | Statement Output | Query...

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

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

```
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	Great for landscapes	2009-04-03

CS669 on postgres@PostgreSQL 10

```
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	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 FOUR

OVERVIEW

In Section Four, you will apply what you learned in Section Three to a new table, without being given the commands and screenshots. This will help to solidify your learning. You will work with the Ocean_resort table diagrammed below. Note that the bolded columns represent those with a NOT NULL constraint.

	Ocean_resort	
PK	**ocean_resort_id**	**DECIMAL(12)**
	name	**VARCHAR(64)**
	description	VARCHAR(255)
	date_opened	**DATE**

STEPS

49. Create the Ocean_resort table with all of its columns, datatypes, and not null constraints. Capture a screenshot of the command and the result of its execution.

50. Insert a row with the following values.
    ocean_resort_id = 151
    name = Light of the Ocean Resort
    description = Gorgeous setting
    date_opened = 05-May-2001

    Capture a screenshot of the command and the result of its execution.

51. Insert another row with the following values. Make sure that the comma-separated list of column names contains the description column.
    ocean_resort_id = 152
    name = Breathtaking Bahamas Resort
    description = NULL
    date_opened = 23-Mar-1979

    Capture a screenshot of the command and the result of its execution.

52. Insert another row with the following values. Make sure that the comma-separated list of column names *does not* contain the description column.
    ocean_resort_id = 153
    name = Luxurious London Resort
    date_opened = 07-Sep-1910

    Capture a screenshot of the command and the result of its execution.

53. Select all rows from the Ocean_resort table. Capture a screenshot of the command and the result of its execution.

54. Attempt to insert a row with the following values. The insert command will fail because the name column must have a value. You can use either method to attempt to insert the NULL value.
    ocean_resort_id = 154
    name = NULL
    description = Experience The Netherlands No Other Way
    date_opened = 03-Jan-1999

    Capture a screenshot of the command and the result of its execution.

55. Explain how you would interpret the error message illustrated in Step 54 to conclude that the name column is missing a required value.

56. Retrieve only the name and description values for the one row where ocean_resort_id is 151. Capture a screenshot of the command and the result of its execution.

57. Explain why it is useful to limit the number of rows and columns returned from a SELECT statement, such as the statement performed in Step 56.

58. Two rows have no description. With a single command, update the description for both of them so that the new value is “The Most Relaxing Place on Earth”. Capture a screenshot of the command and the result of its execution.

59. With a single command, delete all rows that have an ocean_resort_id greater than 152. Capture a screenshot of the command and the result of its execution.

60. Retrieve all rows and columns from the Ocean_resort table. Capture a screenshot of the command and the result of its execution.