Project 1

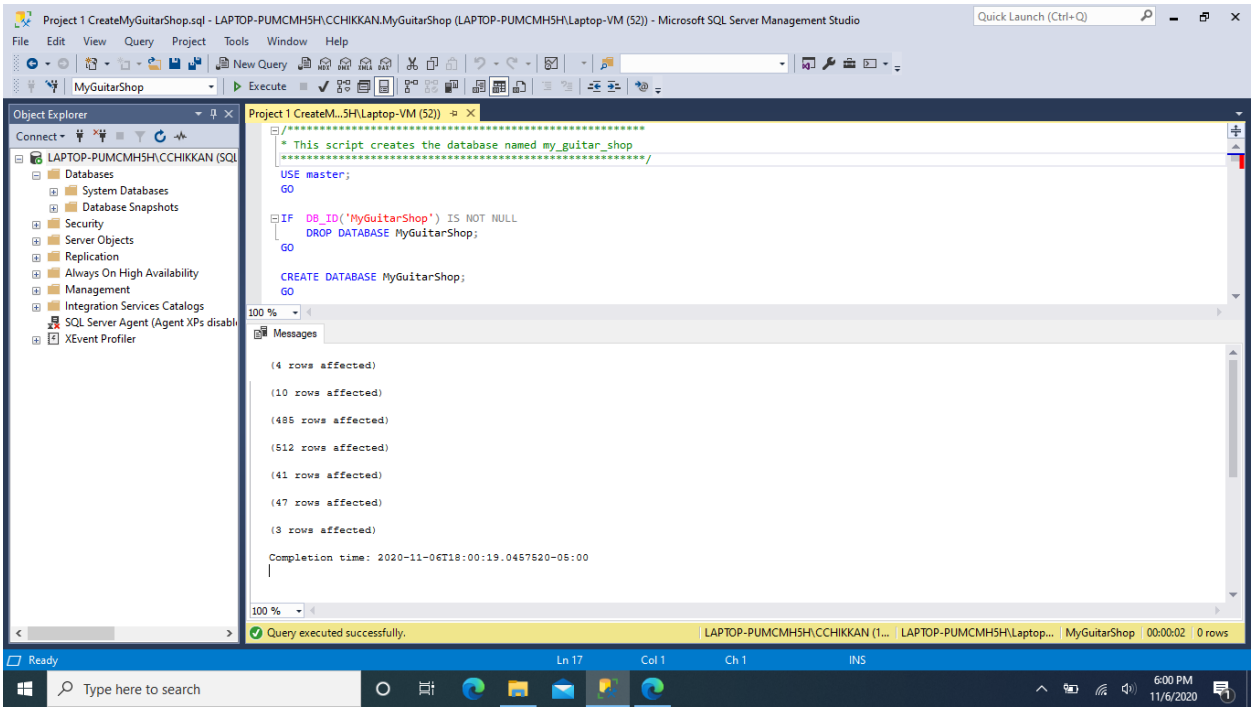
I. My Guitar Shop Database

A. Database Setup [2 pts.]

1. (1) Download CreateMyGuitarShop.sql from Project 1 directory on Blackboard and open it in SQL server management studio. Execute the entire script and show the message in the Message tab, indicating the script is executed successfully. A complete screenshot of execution result is required. Your screenshot should show your entire SQL server window. You are not allowed to crop out any part and follow this for all the questions in this project.

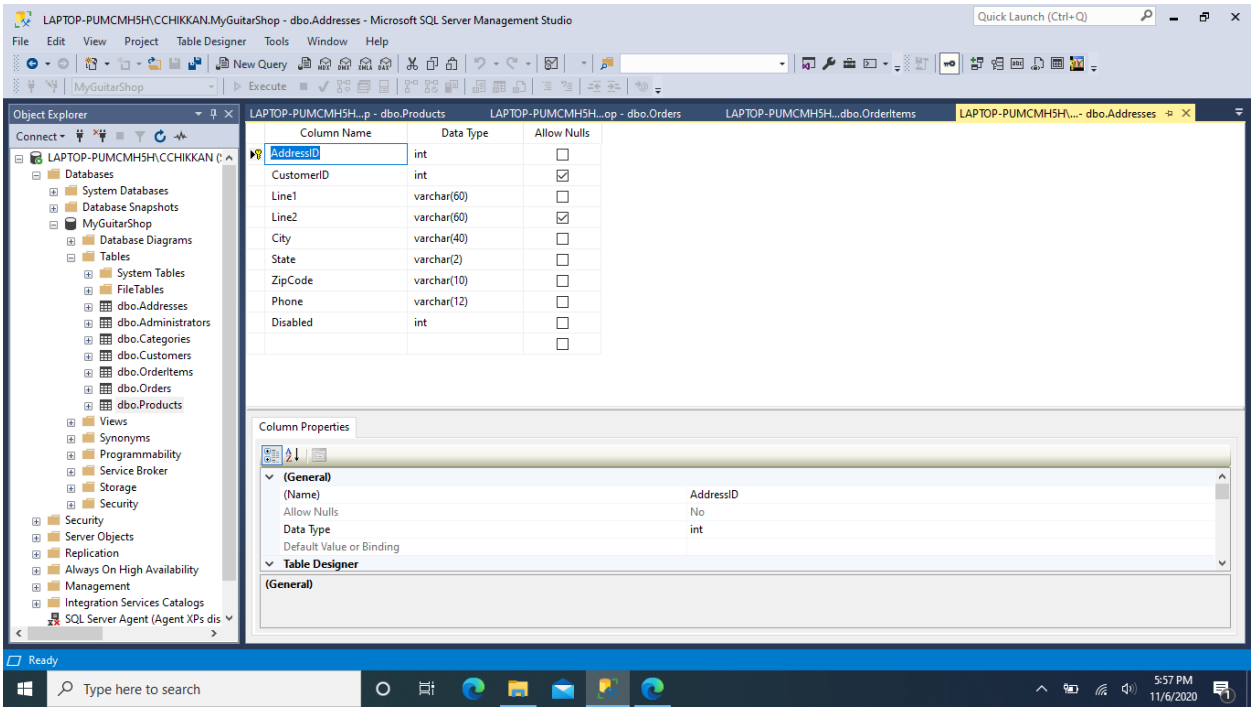
Solution:

The CreateMyGuitarShop sql file is downloaded from the Blackboard and opened in SQL server management studio. Once executed, a new database named **MyGuitarShop** is created in the management studio.

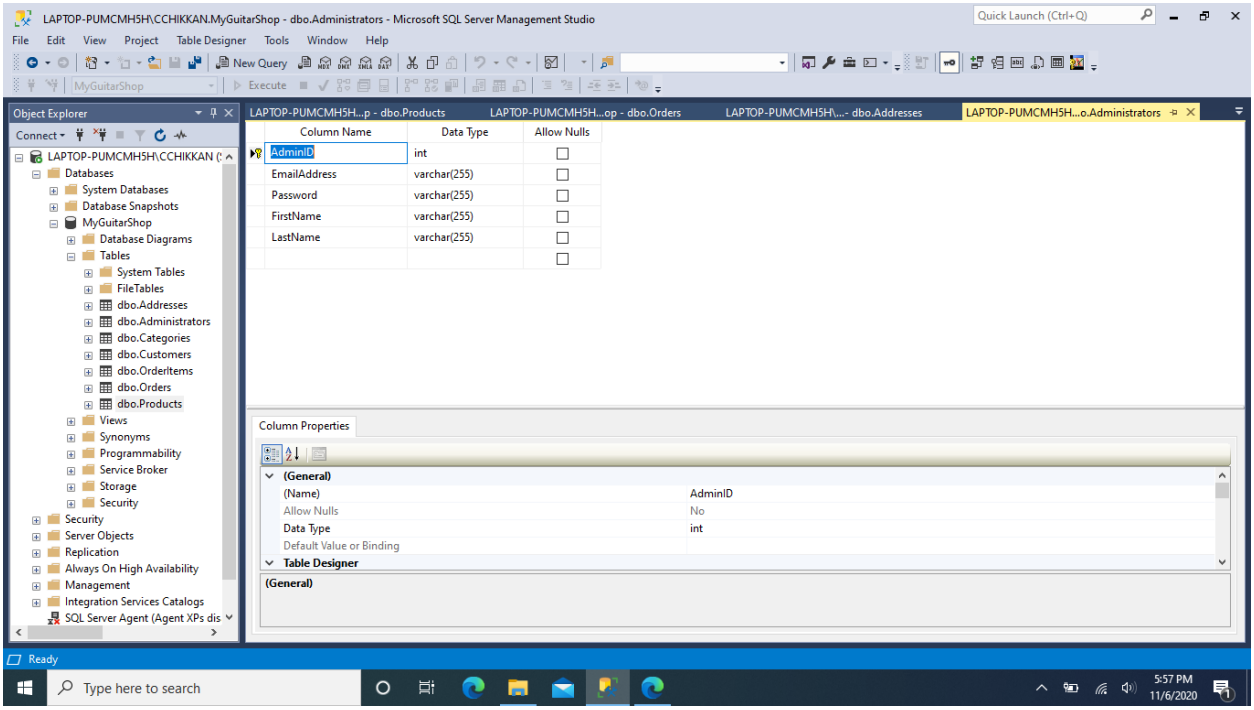


(2) Navigate through the database objects and view the column definitions for each table. Open a new Query Editor window. Show details in Orders table and OrdersItems table using SELECT statement. Full screenshots of execution results are required as mentioned.

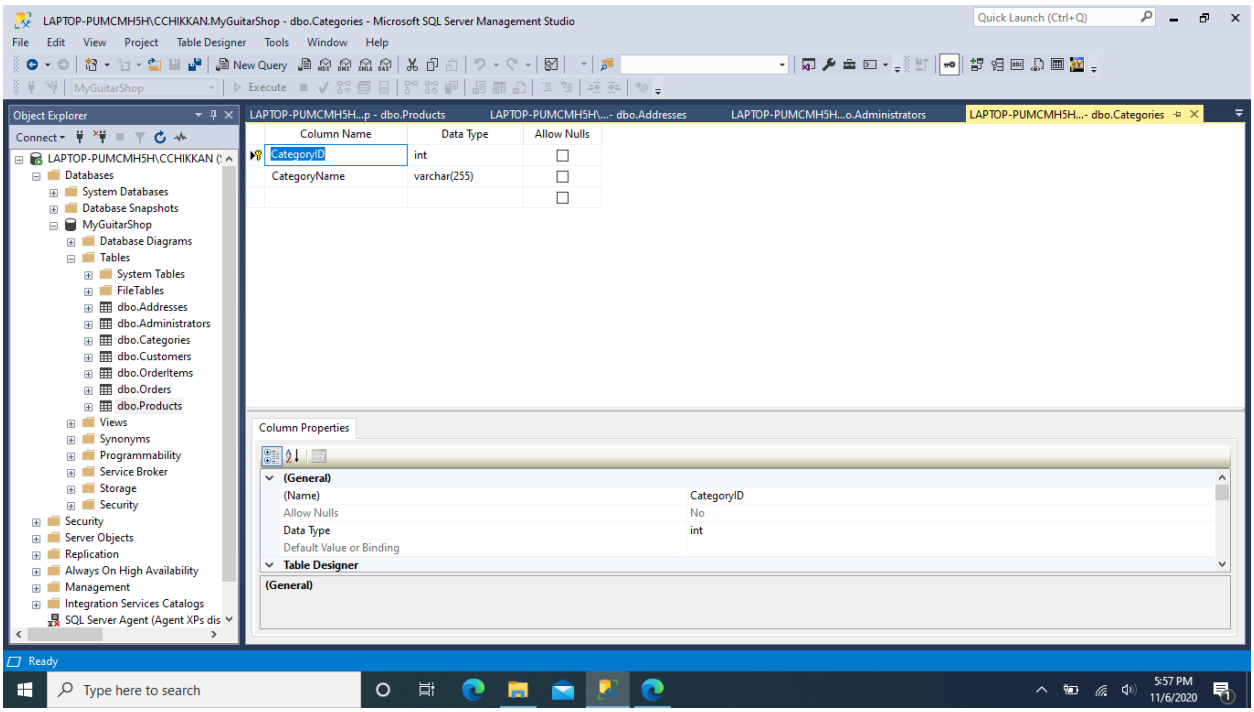
Solution:
The column definitions for Addresses Table:



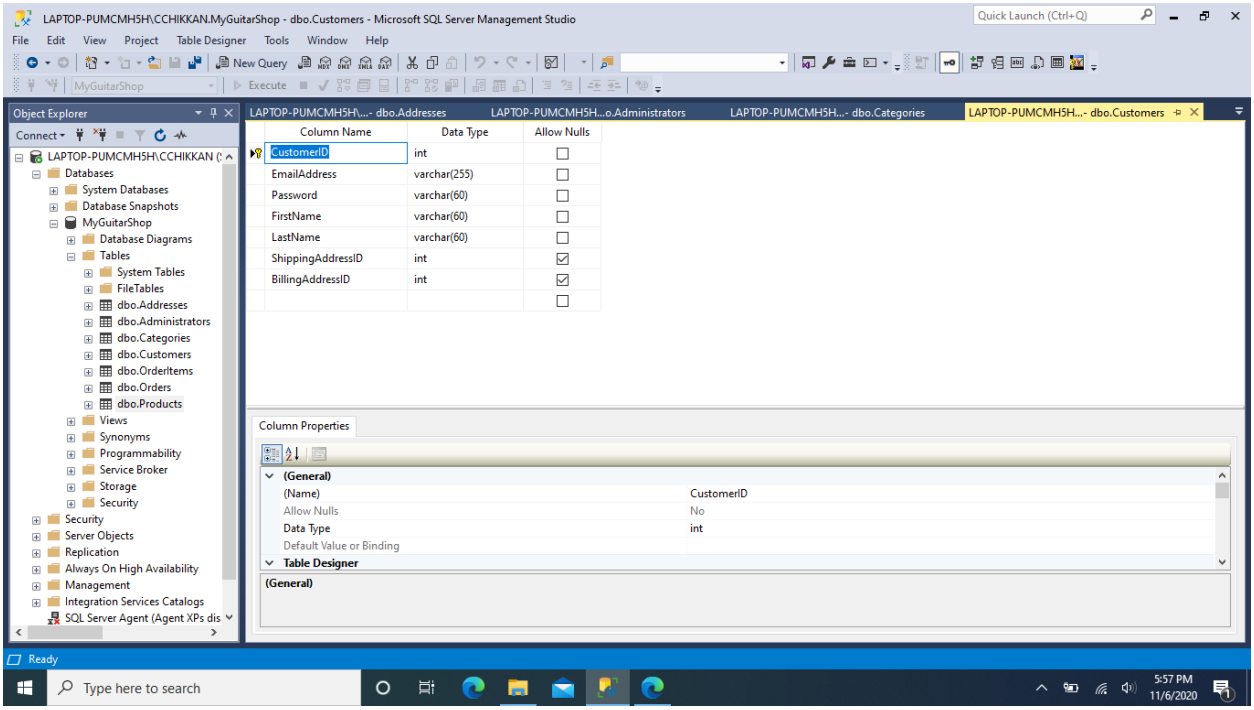
The column definitions for Administrators Table:



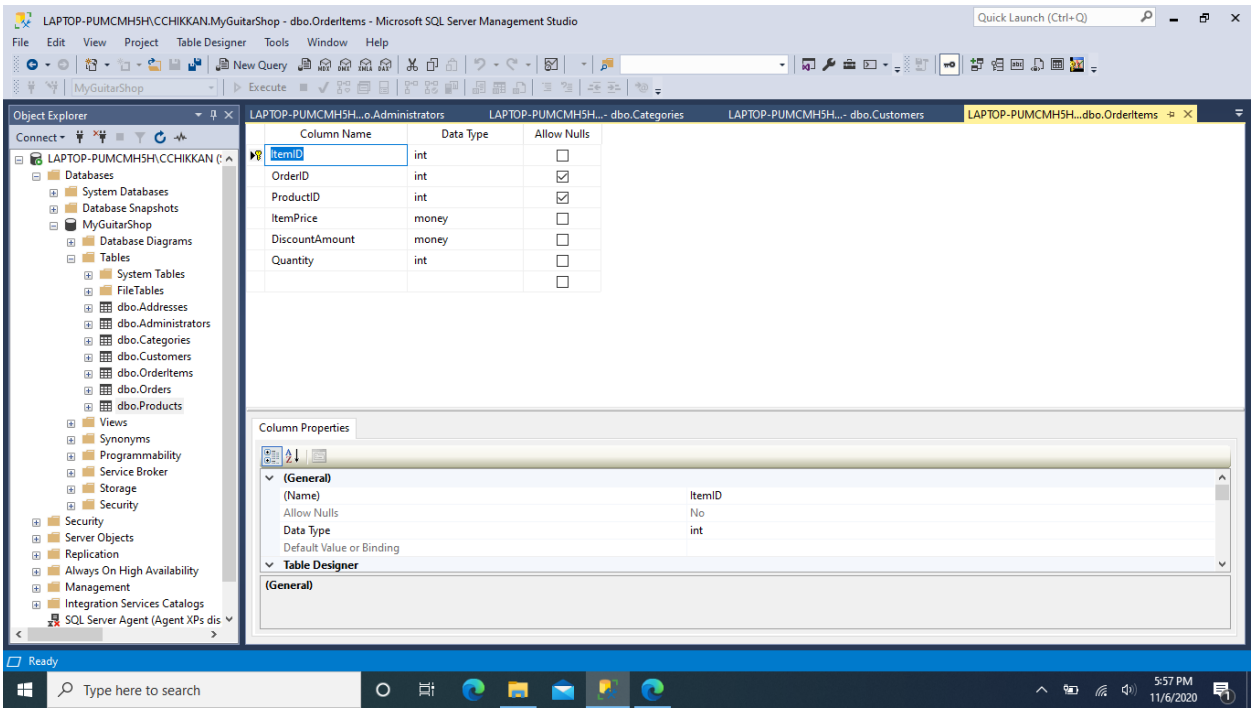
The column definitions for Categories Table:



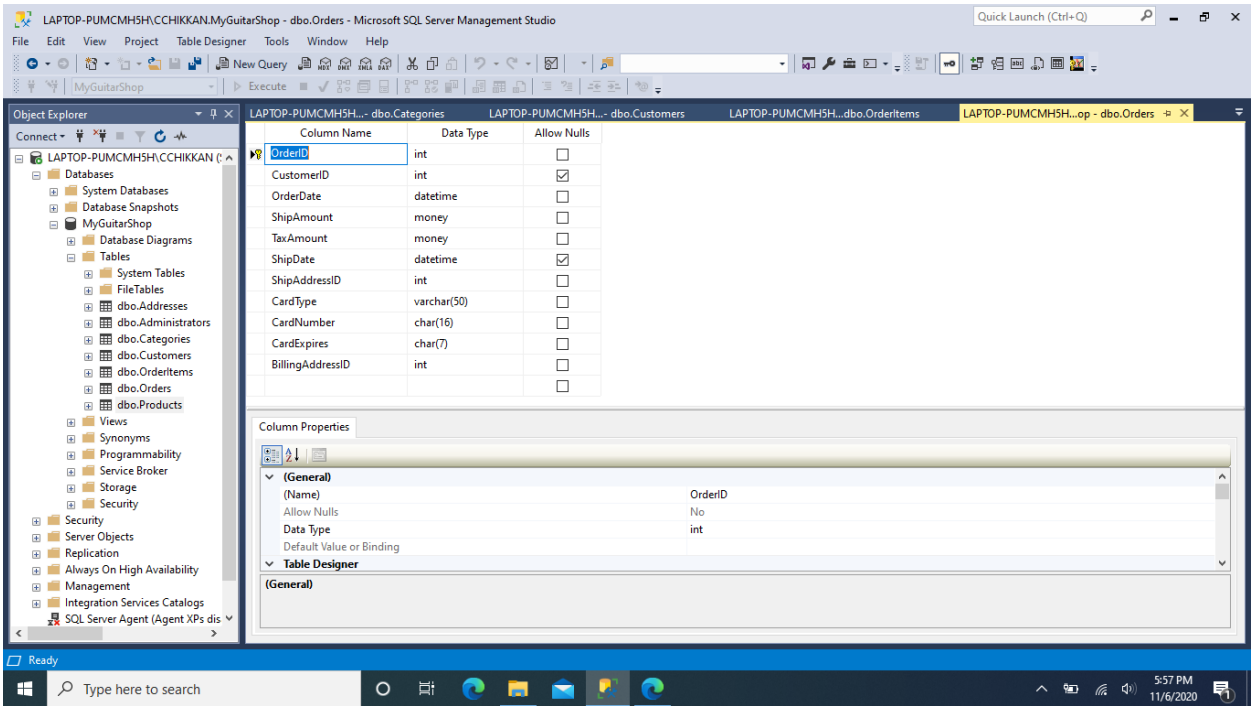
The column definitions for Customers Table:



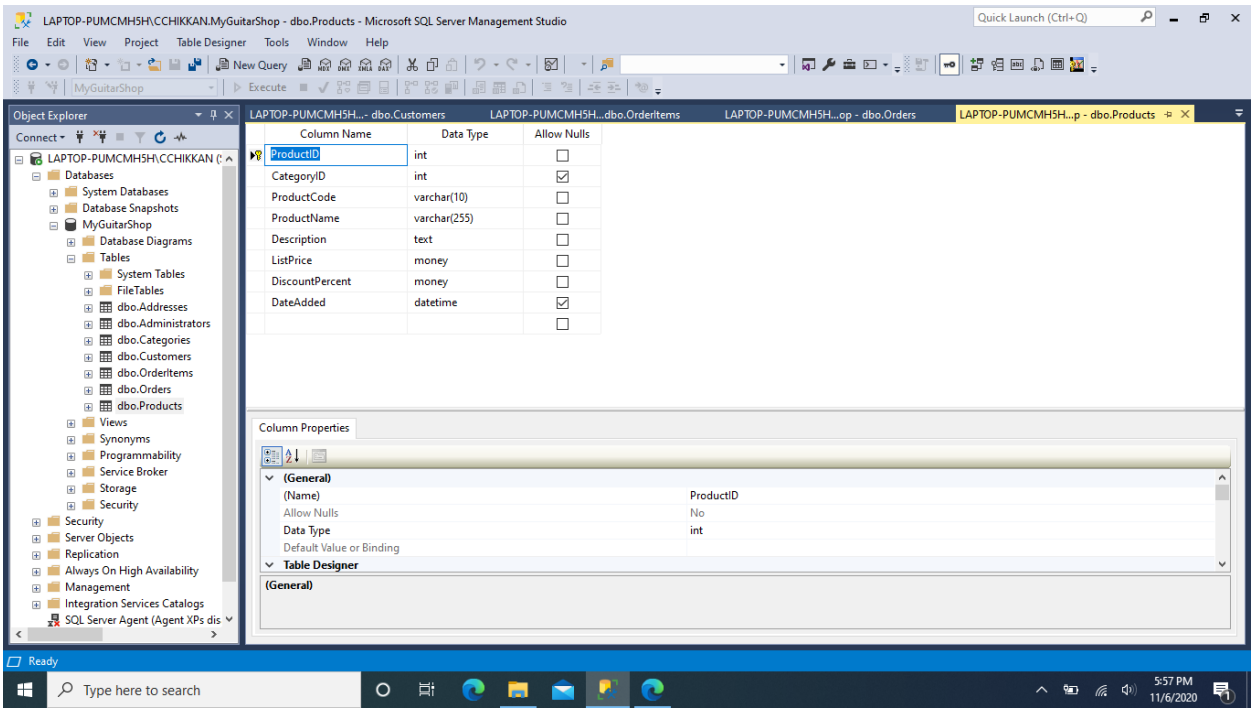
The column definitions for OrderItems Table:



The column definitions for Orders Table:

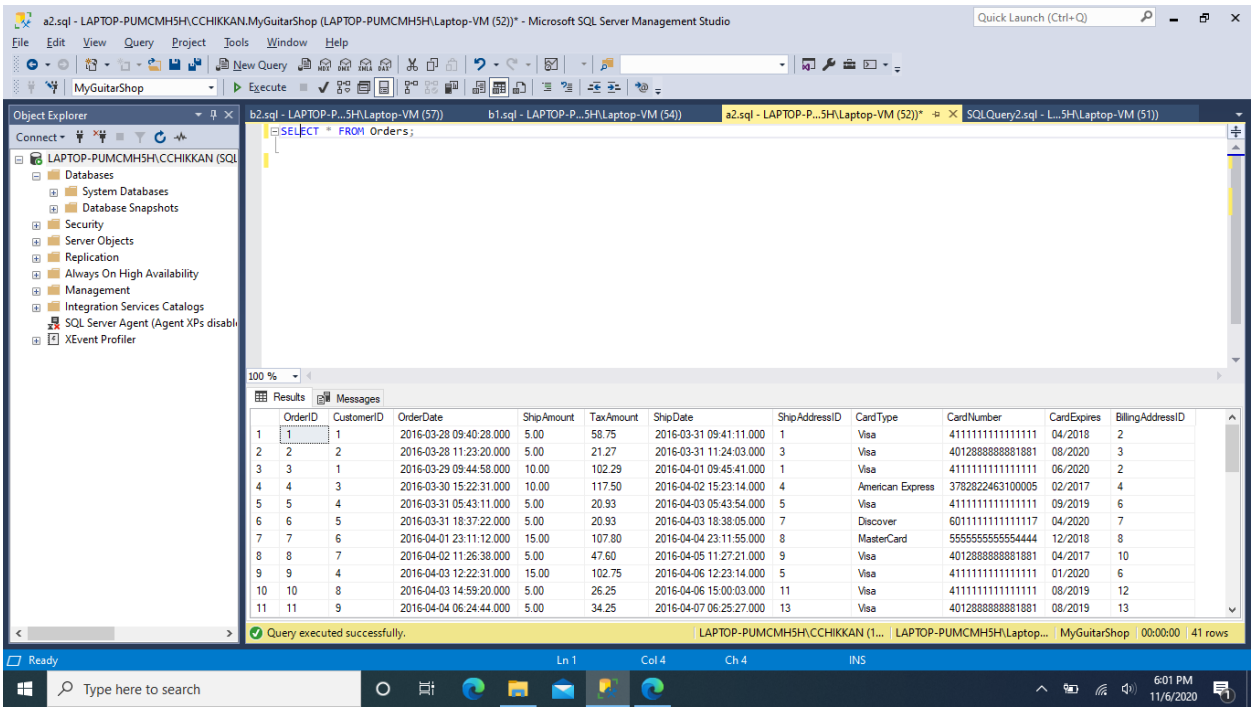


The column definitions for Products Table:

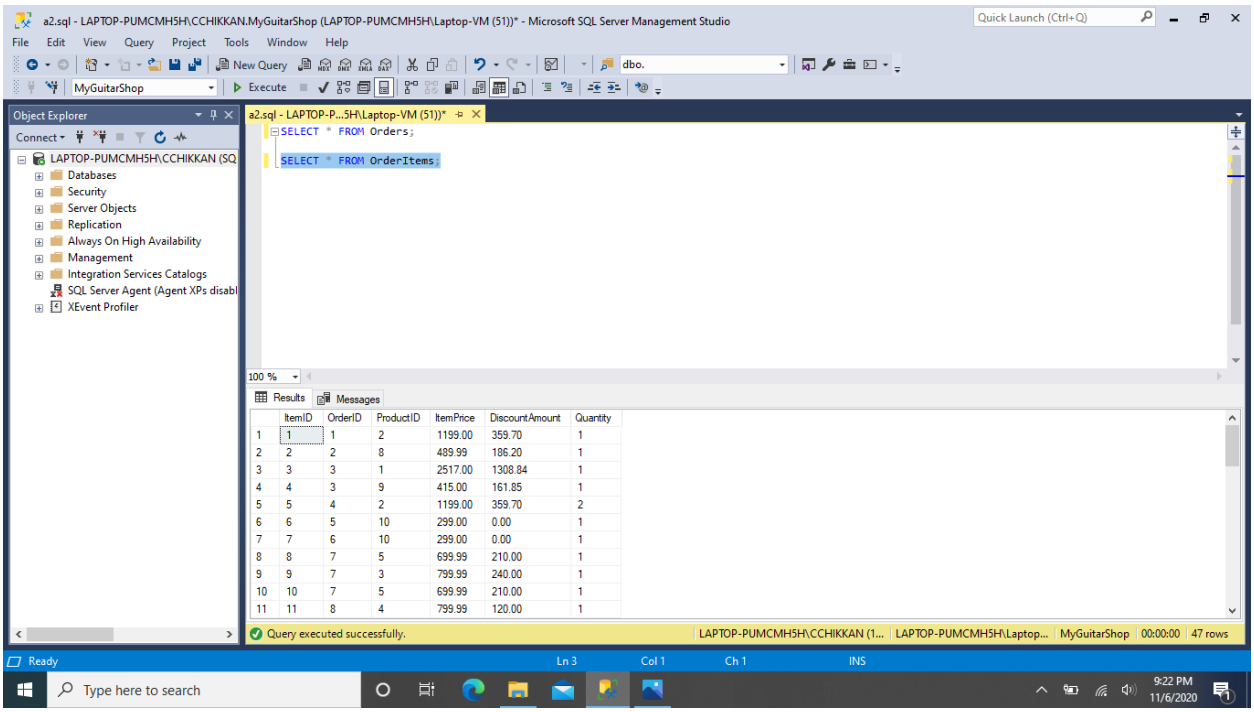


Query to show details in Orders table and OrdersItems table using SELECT statement:

SELECT * from MyGuitarShop.dbo.Orders;



SELECT * from MyGuitarShop.dbo.OrdersItems;

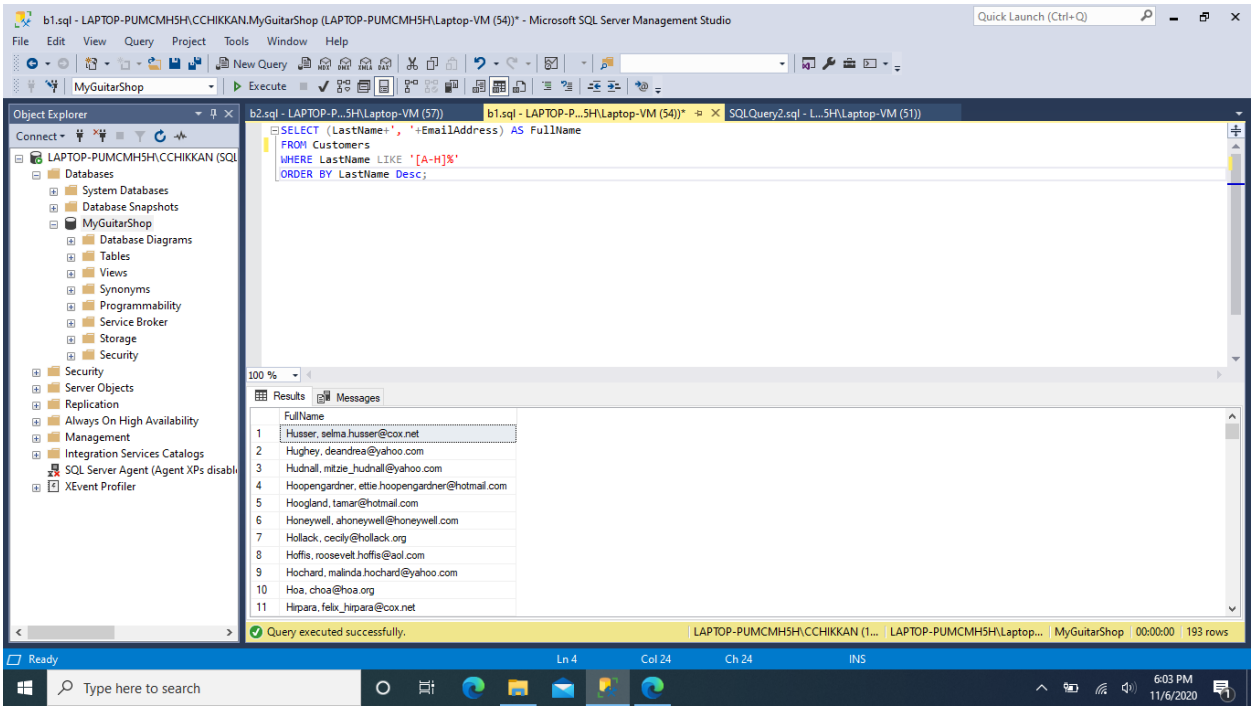


B. An Introduction to SQL [6 pts.]

1. [3] Write a SELECT statement that returns one column from the Customers table named FullName that joins the LastName and EmailAddress columns. Format this column with the last name, a comma, a space, and then Email Address like this: Abdallah, johnetta_abdallah@aol.com Add an ORDER BY clause to this statement that sorts the result set by last name in descending sequence. Return only the contacts whose last name begins with a letter from A to H.

Solution:
The below query returns the Full name of customers by combining the last name and email address columns and sorted by last name in descending order.

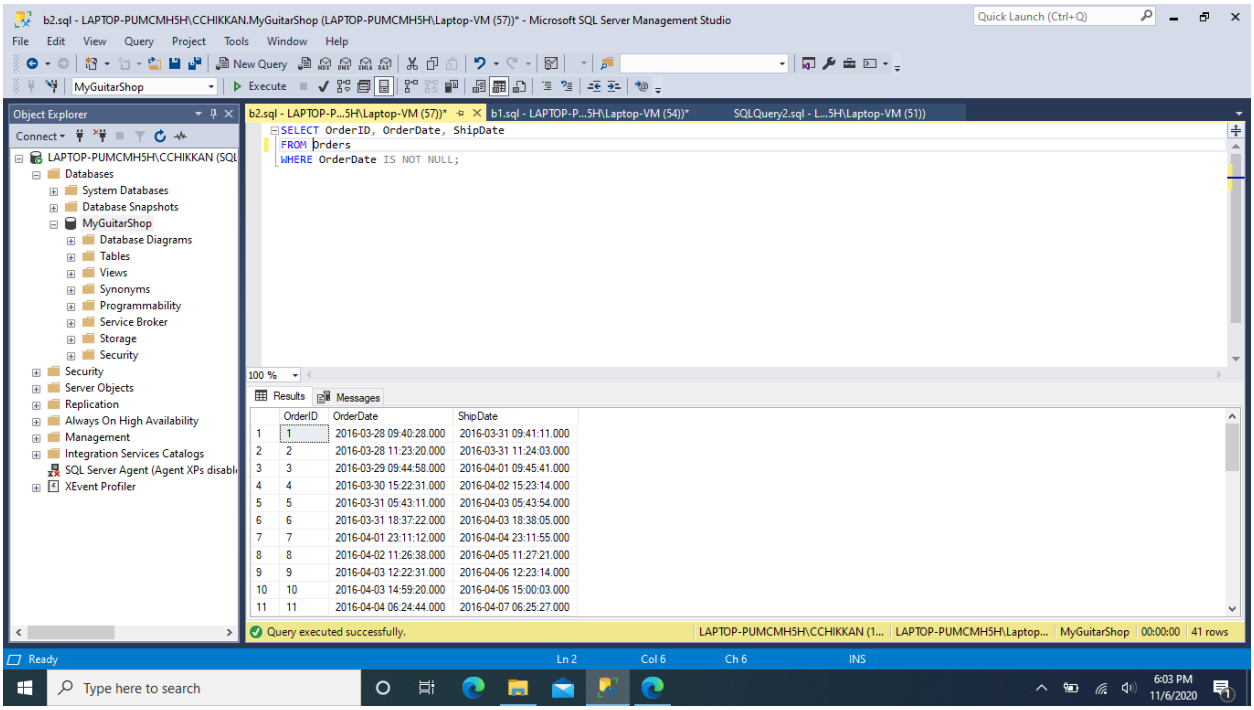
```
SELECT (LastName+', '+EmailAddress) AS FullName
FROM Customers
WHERE LastName LIKE '[A-H]%'
ORDER BY LastName Desc;
```



2. [3] Write a SELECT statement that returns these columns from the Orders table:
   OrderID The OrderID column
   OrderDate The OrderDate column
   ShipDate The ShipDate column
   Return only the rows where the OrderDate column does not contain a null value.

Solution:
The below query that returns OrderID, OrderDate and ShipDate columns and only those rows where ShipDate is not null.

SELECT OrderID, OrderDate, ShipDate
FROM Orders
WHERE OrderDate IS NOT NULL;

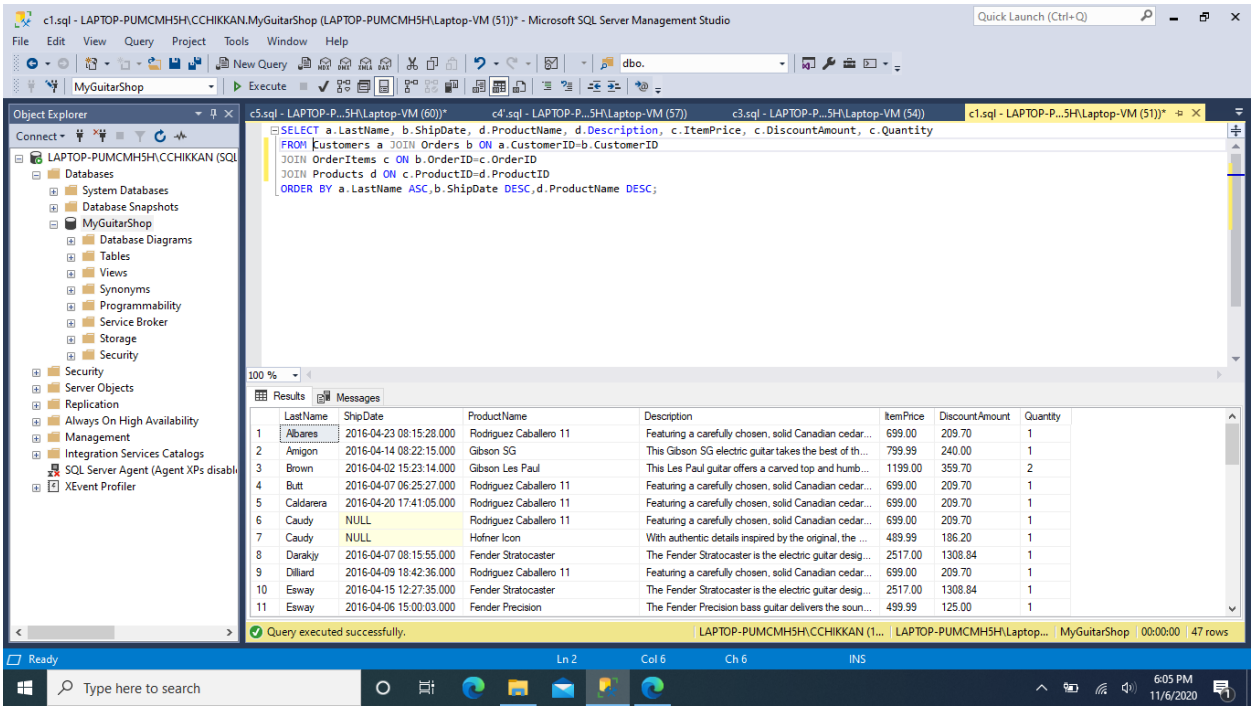


C. The essential SQL skills [44 pts.]

1. [4] Write a SELECT statement that joins the Customers, Orders, OrderItems, and Products tables. This statement should return these columns: LastName, ShipDate, ProductName, Description, ItemPrice, DiscountAmount, and Quantity. Use aliases for the tables. Sort the final result set by LastName in ascending order, and in descending order for ShipDate, and ProductName

Solution:
The above query is the SELECT statement that returns LastName, ShipDate, ProductName, Description, ItemPrice, DiscountAmount, and Quantity columns from Customers, Orders, OrderItems, and Products tables and also sorting the result set by LastName, ShipDate, and ProductName.

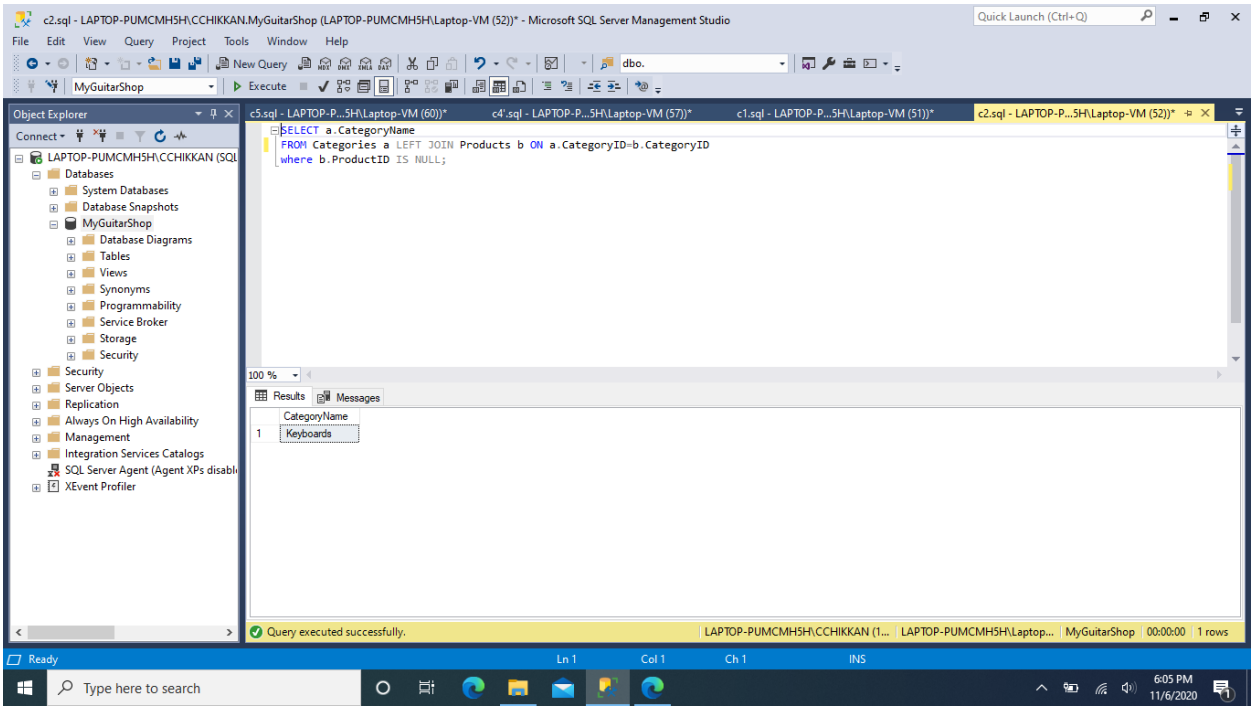
```
SELECT a.LastName, b.ShipDate, d.ProductName, d.Description, c.ItemPrice,
c.DiscountAmount, c.Quantity
FROM Customers a JOIN Orders b ON a.CustomerID=b.CustomerID
JOIN OrderItems c ON b.OrderID=c.OrderID
JOIN Products d ON c.ProductID=d.ProductID
ORDER BY a.LastName ASC,b.ShipDate DESC,d.ProductName DESC;
```



2. [4] Write a SELECT statement that returns CategoryName column from the Categories table. Return one row for each category that has never been used. (Hint: Use an outer join and only return rows where the ProductID column contains a null value.)

Solution:
The below query returns one row for each category that has never been used with CategoryName column.

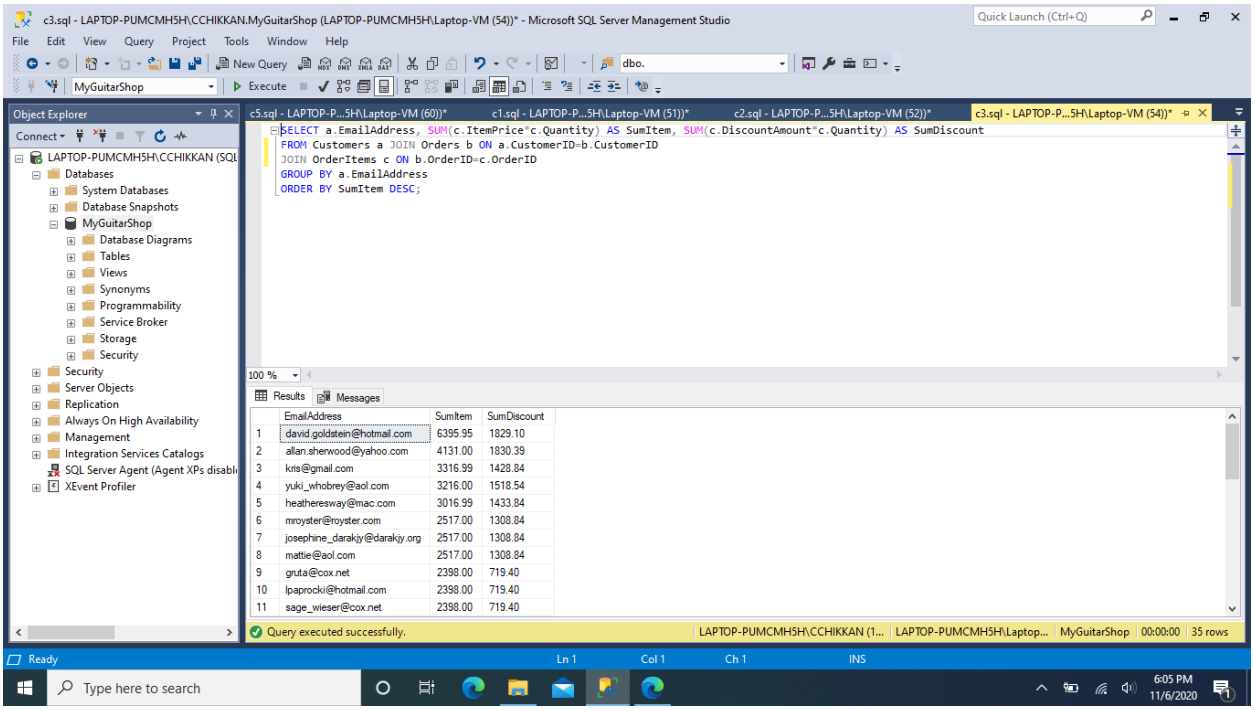
SELECT a.CategoryName
FROM Categories a LEFT JOIN Products b
ON a.CategoryID=b.CategoryID
WHERE b.ProductID IS NULL;



3. [4] Write a SELECT statement that returns one row for each customer that has orders with these columns:
   a) The EmailAddress column from the Customers table
   b) The sum of the ItemPrice in the OrderItems table multiplied by the quantity in the OrderItems table
   c) The average of the DiscountAmount column in the OrderItems table multiplied by the quantity in the OrderItems table.
   Sort the result set in descending sequence by the item price total for each customer.

Solution:
The below query is a SELECT statement that returns one row for each customer that has orders with the columns mentioned in the above question.

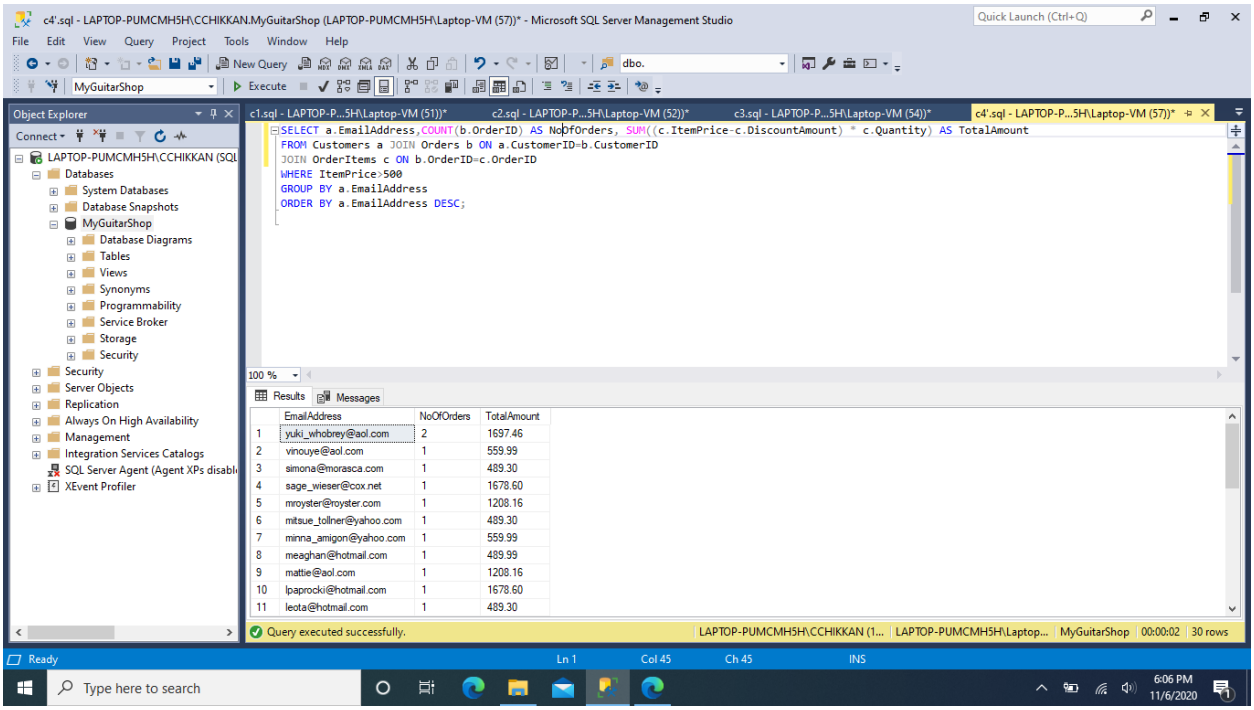
```
SELECT a.EmailAddress, SUM(c.ItemPrice*c.Quantity) AS SumItem,
SUM(c.DiscountAmount*c.Quantity) AS SumDiscount
FROM Customers a JOIN Orders b ON a.CustomerID=b.CustomerID
JOIN OrderItems c ON b.OrderID=c.OrderID
GROUP BY a.EmailAddress
ORDER BY SumItem DESC;
```



4. [4] Write a SELECT statement that returns one row for each customer that has orders with these columns:
   a) The EmailAddress column from the Customers table
   b) A count of the number of orders
   c) The total amount for each order (Hint: First, subtract the discount amount from the price. Then, multiply by the quantity)
   Return only those rows where items have a more than 500 ItemPrice value. Sort the result set in descending order of EmailAddress column.

Solution:
The below query is a SELECT statement that returns one row for each customer that has orders with the columns mentioned above.

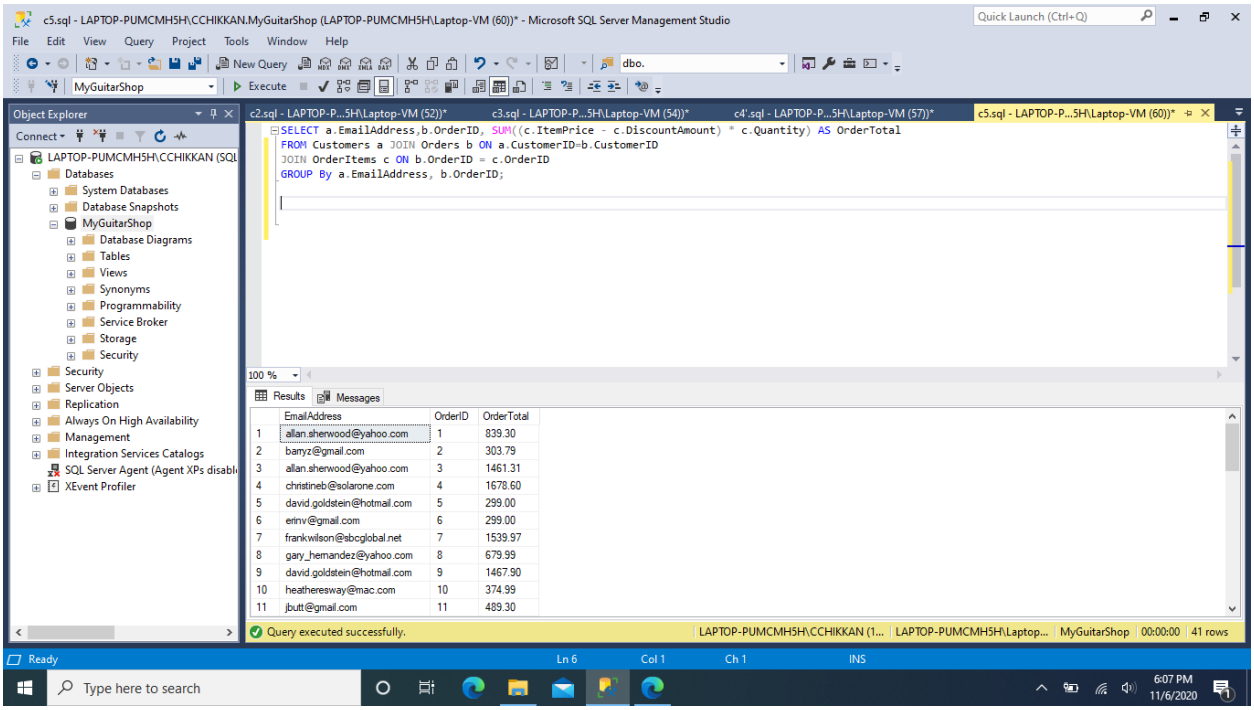
```
SELECT a.EmailAddress,COUNT(b.OrderID) AS NoOfOrders,
SUM((c.ItemPrice-c.DiscountAmount) * c.Quantity)
AS TotalAmount
FROM Customers a JOIN Orders b ON a.CustomerID=b.CustomerID
JOIN OrderItems c ON b.OrderID=c.OrderID
WHERE ItemPrice>500
GROUP BY a.EmailAddress
ORDER BY a.EmailAddress DESC;
```



5. [4] (1) Write a SELECT statement that returns three columns: EmailAddress, OrderID, and the order total amount for each customer. To do this, you can group the result set by the EmailAddress and OrderID columns. In addition, you must calculate the order total amount from the columns in the OrderItems table.

Solution:
The below query is a SELECT statement that returns three columns: EmailAddress, OrderID, and the OrderTotal for each customer by grouping EmailAddress and OrderID.

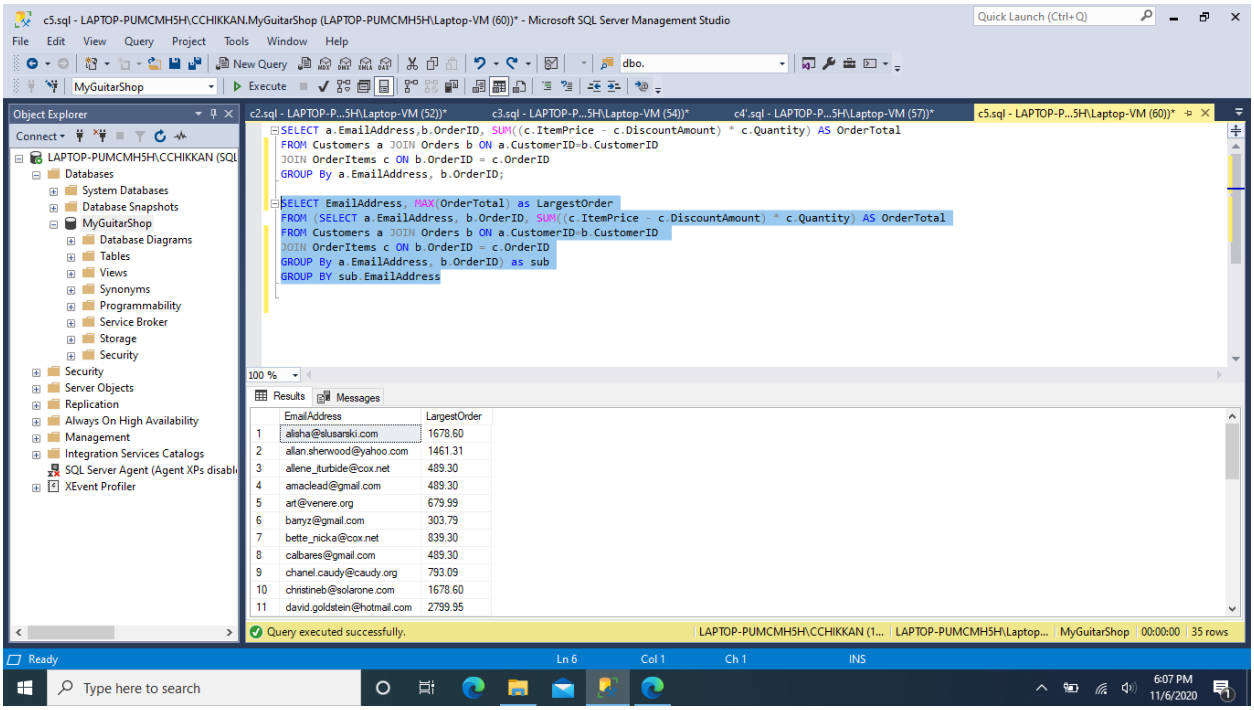
SELECT a.EmailAddress,b.OrderID, SUM((c.ItemPrice - c.DiscountAmount) * c.Quantity) AS OrderTotal
FROM Customers a JOIN Orders b ON a.CustomerID=b.CustomerID
JOIN OrderItems c ON b.OrderID = c.OrderID
GROUP By a.EmailAddress, b.OrderID;



(2) Write a second SELECT statement that uses the first SELECT statement in its FROM clause. The main query should return two columns: the customer's email address and the largest order for that customer. To do this, you can group the result set by the EmailAddress column.

Solution:
The below query is a SELECT statement that uses the SELECT statement which is written above in its FROM clause that returns two columns i.e. customer's email address and the largest order for that customer by grouping result set by the EmailAddress column.

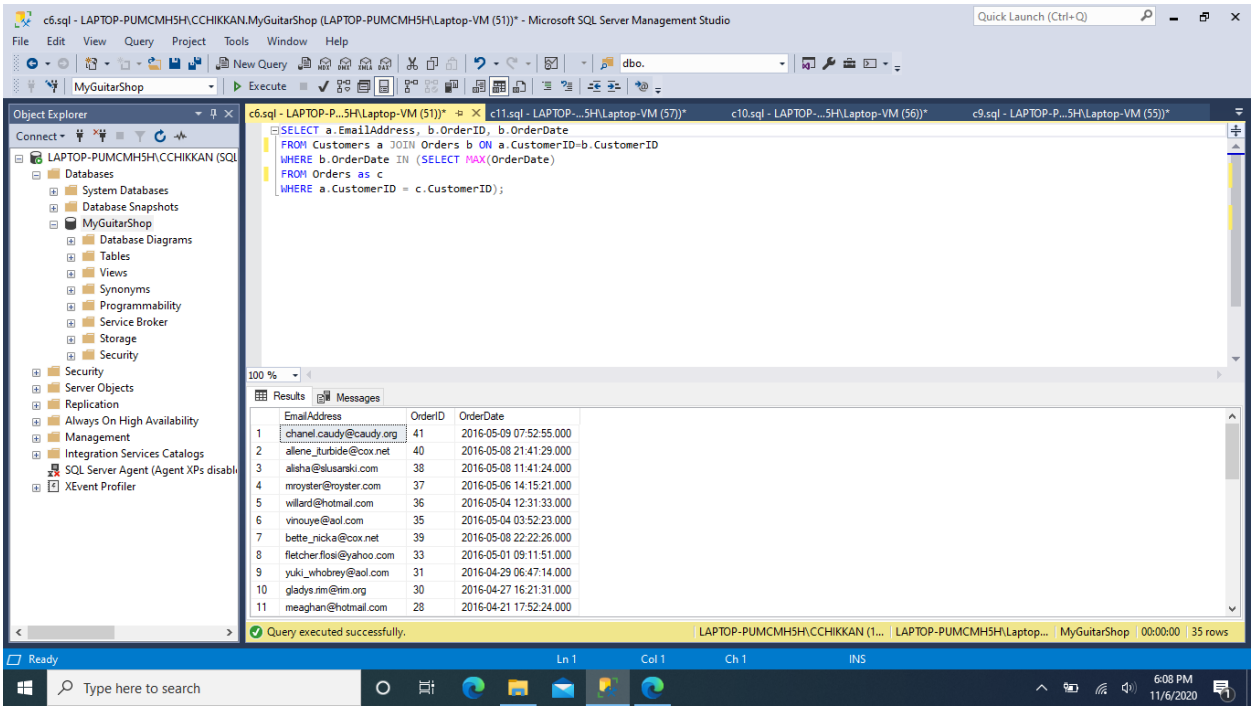
```
SELECT EmailAddress, MAX(OrderTotal) as LargestOrder
FROM (SELECT a.EmailAddress, b.OrderID, SUM((c.ItemPrice - c.DiscountAmount) *
c.Quantity) AS OrderTotal
FROM Customers a JOIN Orders b ON a.CustomerID=b.CustomerID
JOIN OrderItems c ON b.OrderID = c.OrderID
GROUP By a.EmailAddress, b.OrderID) as sub
GROUP BY sub.EmailAddress
```



6. [4] Use a correlated subquery to return one row per customer, representing the customer's newest order (the one with the latest date). Each row should include these three columns: EmailAddress, OrderID, and OrderDate.

Solution:
The below query is a correlated subquery to return one row per customer, representing the customer's newest order

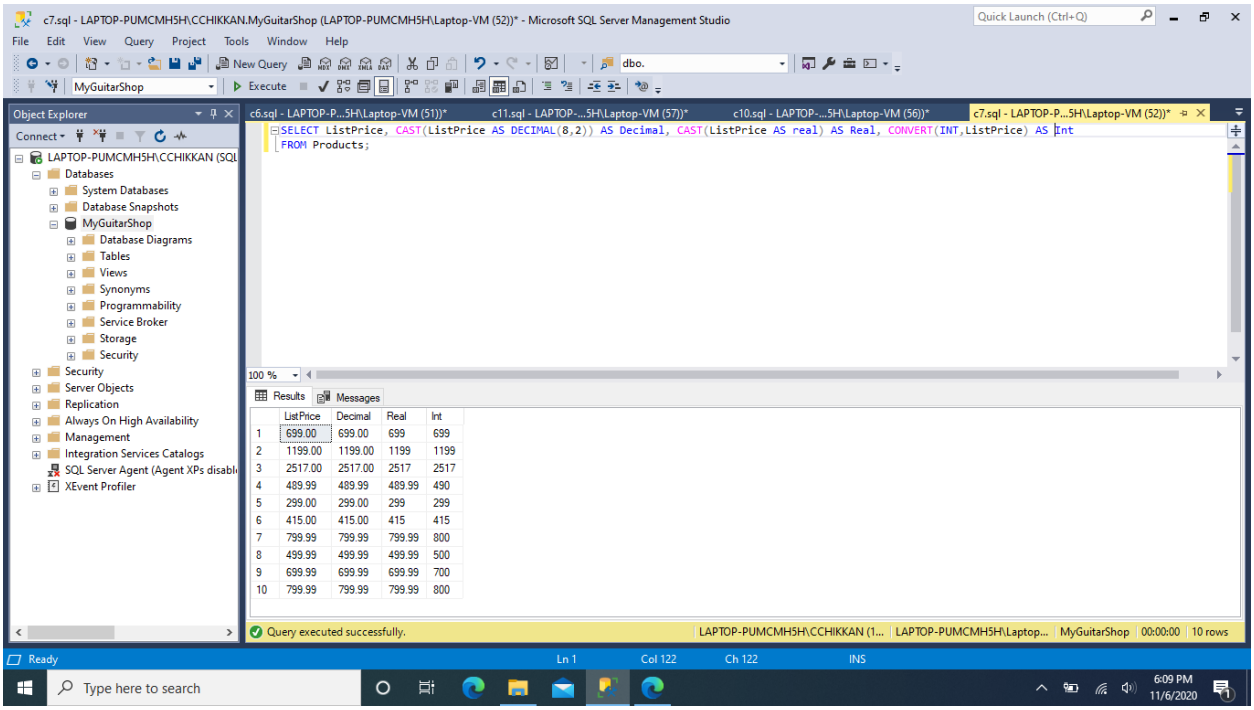
SELECT a.EmailAddress, b.OrderID, b.OrderDate
FROM Customers a JOIN Orders b ON a.CustomerID=b.CustomerID
WHERE b.OrderDate IN (SELECT MAX(OrderDate)
FROM Orders as c
WHERE a.CustomerID = c.CustomerID);



7. [4] Write a SELECT statement that returns these columns from the Products table:
   a) The ListPrice column
   b) A column that uses the CAST function to return the ListPrice column with 2 digits to the right of the decimal point
   c) A column that uses the CAST function to return the ListPrice column as a real number
   d) A column that uses the CONVERT function to return the ListPrice column as an integer.

Solution:
He below is a SELECT statement that returns the columns from the Products table as mentioned in the above question by using CAST and CONVERT functions:

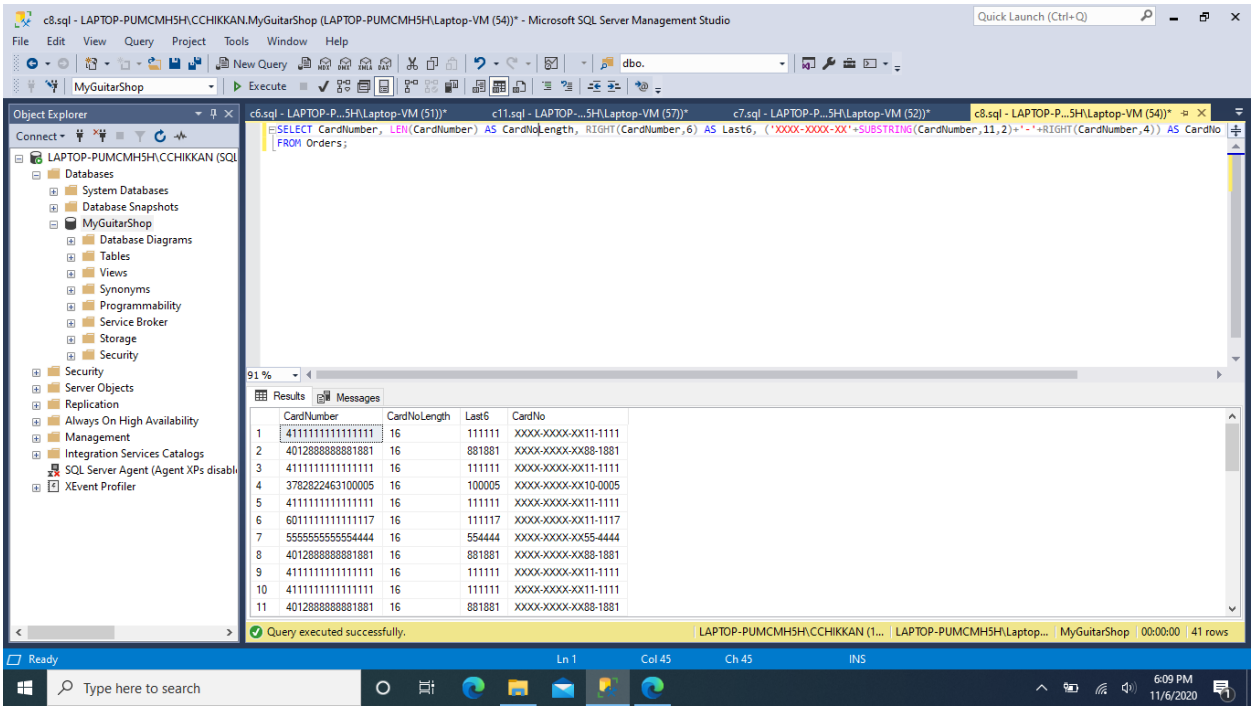
SELECT ListPrice, CAST(ListPrice AS DECIMAL(8,2)) AS Decimal, CAST(ListPrice AS real) AS Real, CONVERT(INT,ListPrice) AS Int
FROM Products;



8. [4] Write a SELECT statement that returns these columns from the Orders table:
   a) The CardNumber column
   b) The length of the CardNumber column
   c) The last six digits of the CardNumber column
   d) A column that displays the last four digits of the CardNumber column in this format: XXXX-XXXX-XX12-3456. In other words, use X's for the first 10 digits of the card number and actual numbers for the last six digits of the number and include dash symbols as specified in the format.

Solution:
The below is a select statement that returns the following four columns from the Orders table as mentioned in the above question.

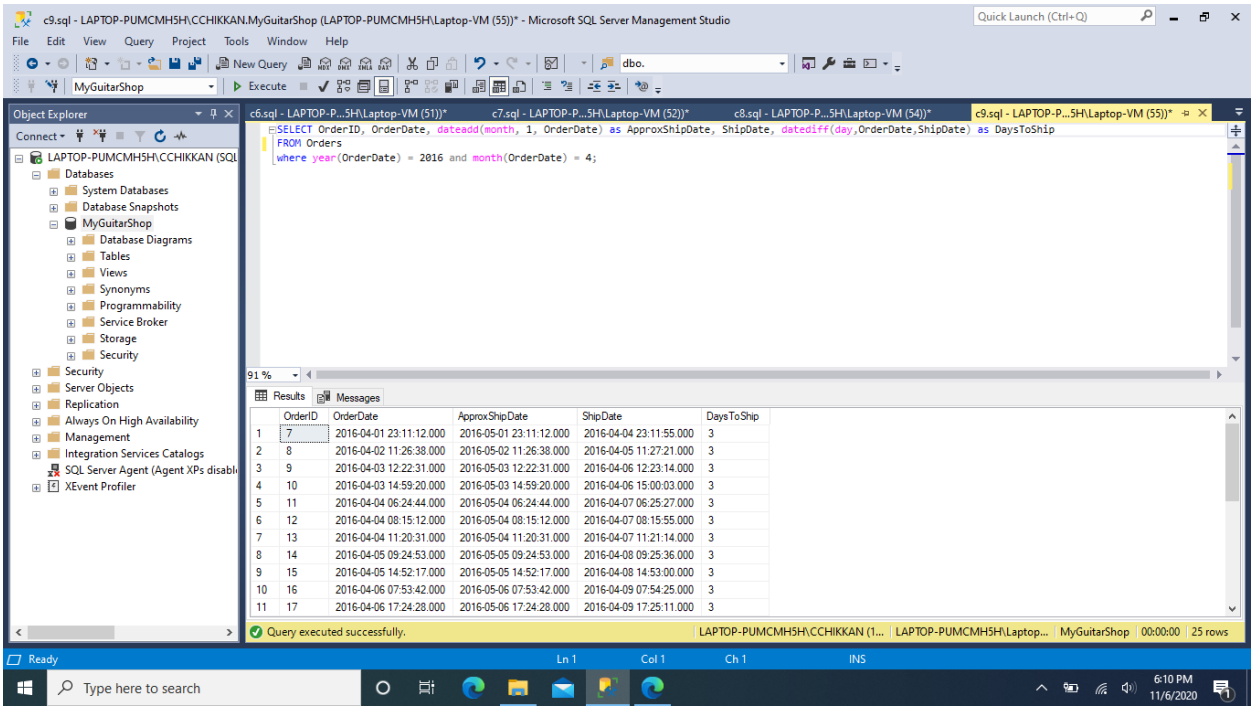
SELECT CardNumber, LEN(CardNumber) AS CardNoLength, RIGHT(CardNumber,6) AS Last6, ('XXXX-XXXX-XX'+SUBSTRING(CardNumber,11,2)+'-'+RIGHT(CardNumber,4)) AS CardNo
FROM Orders;



9. [4] Write a SELECT statement that returns these columns from the Orders table:
   a) The OrderID column
   b) The OrderDate column
   c) A column named ApproxShipDate that's calculated by adding 1 month to the OrderDate column
   d) The ShipDate column
   e) A column named DaysToShip that shows the number of days between the order date and the ship date
   When you have this working, add a WHERE clause that retrieves just the orders for April 2016.

Solution:
The below query is a SELECT statement that returns the columns from the Orders table as mentioned in the above question

SELECT OrderID, OrderDate, dateadd(month, 1, OrderDate) as ApproxShipDate, ShipDate, datediff(day,OrderDate,ShipDate) as DaysToShip
FROM Orders
where year(OrderDate) = 2016 and month(OrderDate) = 4;

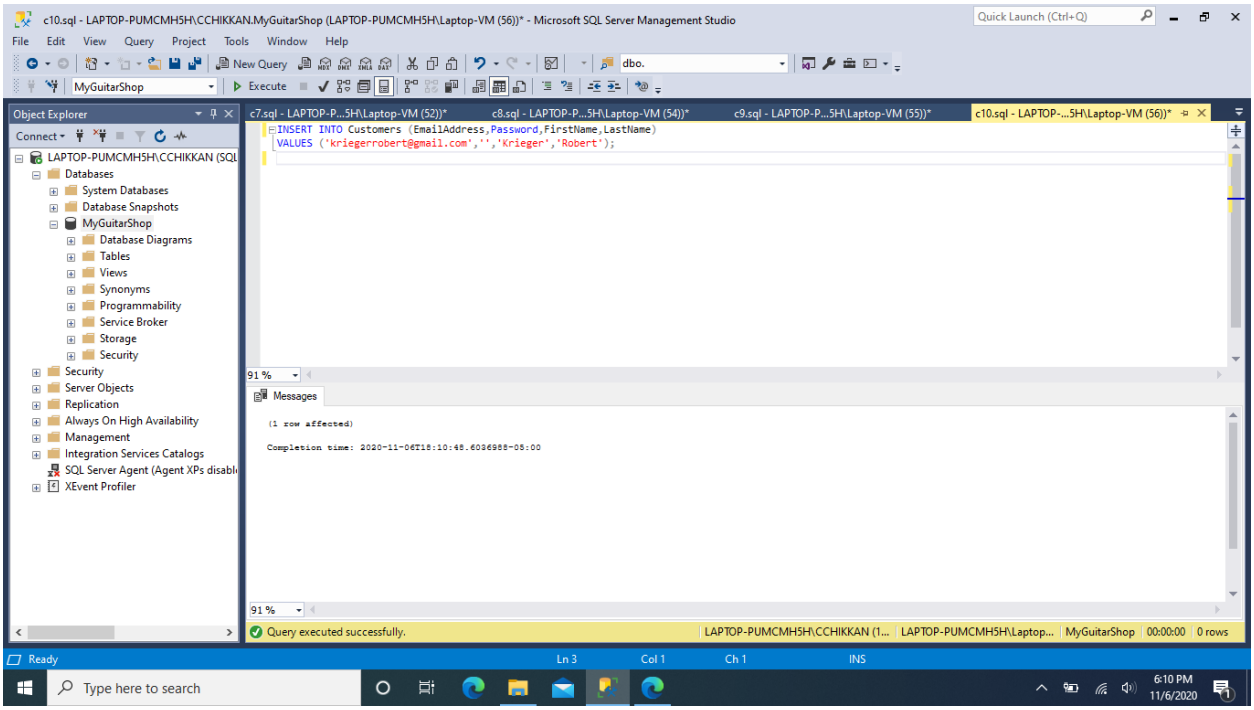


10. [4] Write an INSERT statement that adds this row to the Customers table:
    EmailAddress: kriegerrobert@gmail.com
    Password: (empty string)
    FirstName: Krieger
    LastName: Robert
    Use a column list for this statement.

Solution:

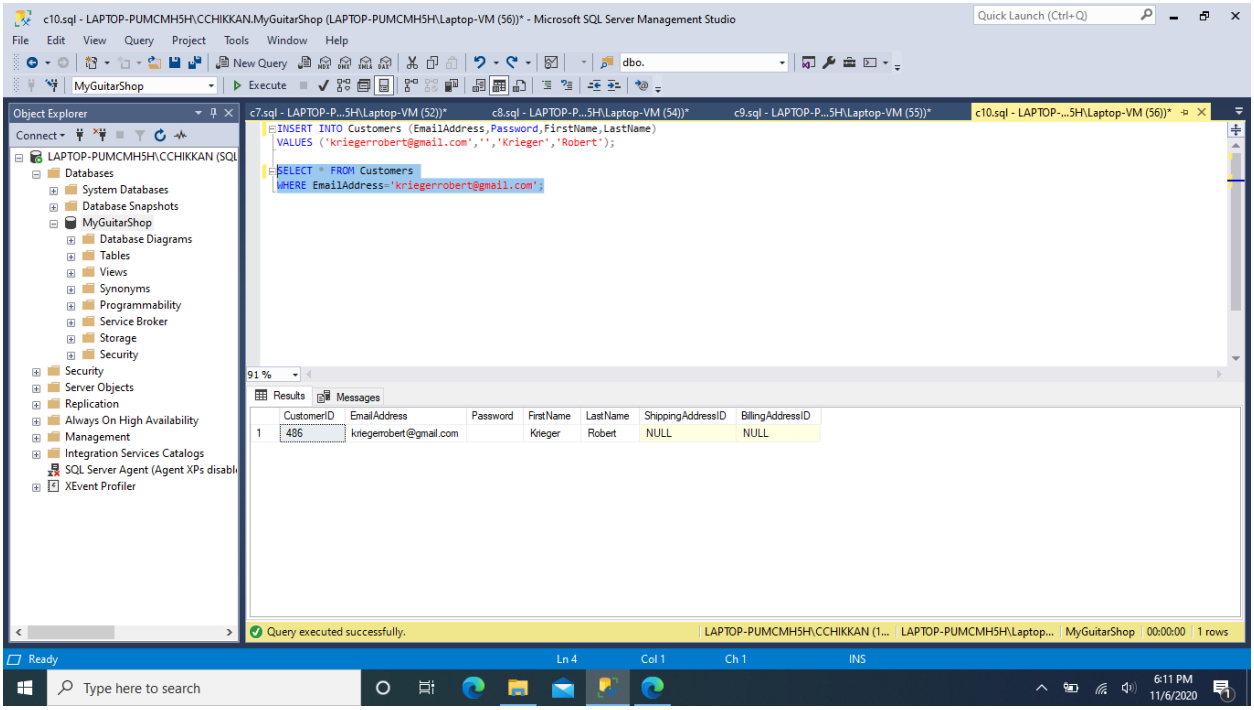
He below query is an INSERT INTO statement that adds a row into the table with the values mentioned above in the question.

INSERT INTO Customers (EmailAddress,Password,FirstName,LastName)
VALUES ('kriegerrobert@gmail.com','','Krieger','Robert');



The below SELECT statement verifies weather a new row was inserted into the table or not.

SELECT * FROM Customers
WHERE EmailAddress='kriegerrobert@gmail.com';



11. [4] Write an UPDATE statement that modifies the Customers table. Change the password column to “secret@1234” for the customer with an email address: kriegerrobert@gmail.com.

Solution:
In this question, we have to perform update operation of password column. Initially we have some other value in the password column for “kriegerrobert@gmail.com”. We can see this in the result screenshot which was obtained below.

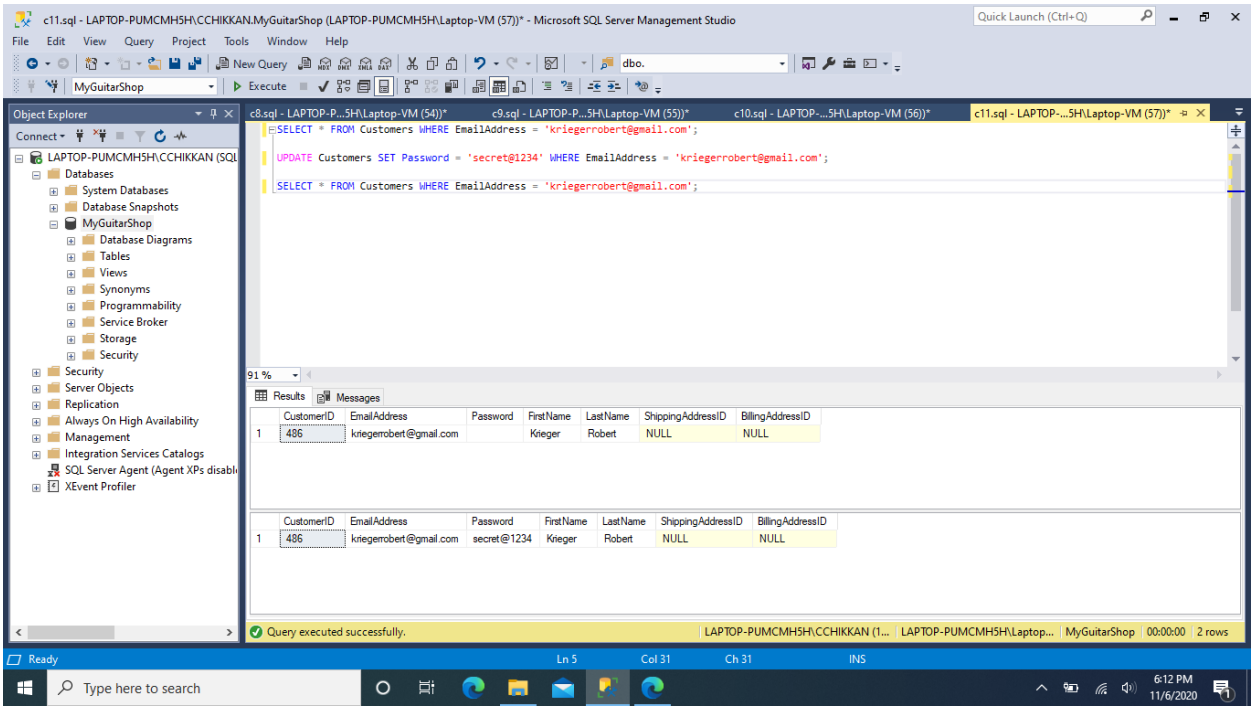
SELECT * FROM Customers WHERE EmailAddress = 'kriegerrobert@gmail.com';

Update operation is performed by using the following query. It updates the password column to “secret@123” for the new customer.

UPDATE Customers SET Password = 'secret@1234' WHERE EmailAddress = 'kriegerrobert@gmail.com';

The below SELECT statement verifies the result.

SELECT * FROM Customers WHERE EmailAddress = 'kriegerrobert@gmail.com';

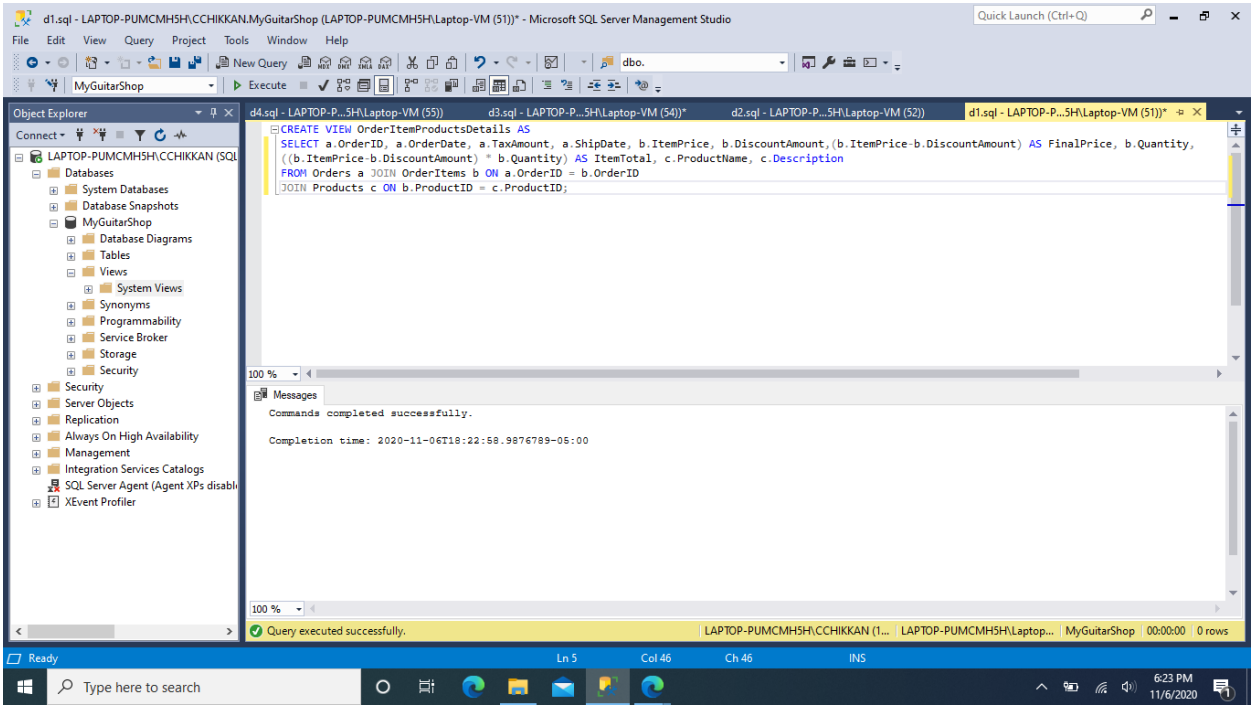


D. Advanced SQL skills (views/stores procedures/ functions / scripts) [24 pts.]

1. [4] Create a view named OrderItemProductsDetails that returns columns from the Orders, OrderItems, and Products tables.
   a) This view should return these columns from the Orders table: OrderID, OrderDate, TaxAmount, and ShipDate.
   b) This view should return these columns from the OrderItems table: ItemPrice, DiscountAmount, FinalPrice (the discount amount subtracted from the item price), Quantity, and ItemTotal (the calculated total for the item).
   c) This view should return the ProductName and Description column from the Products table.

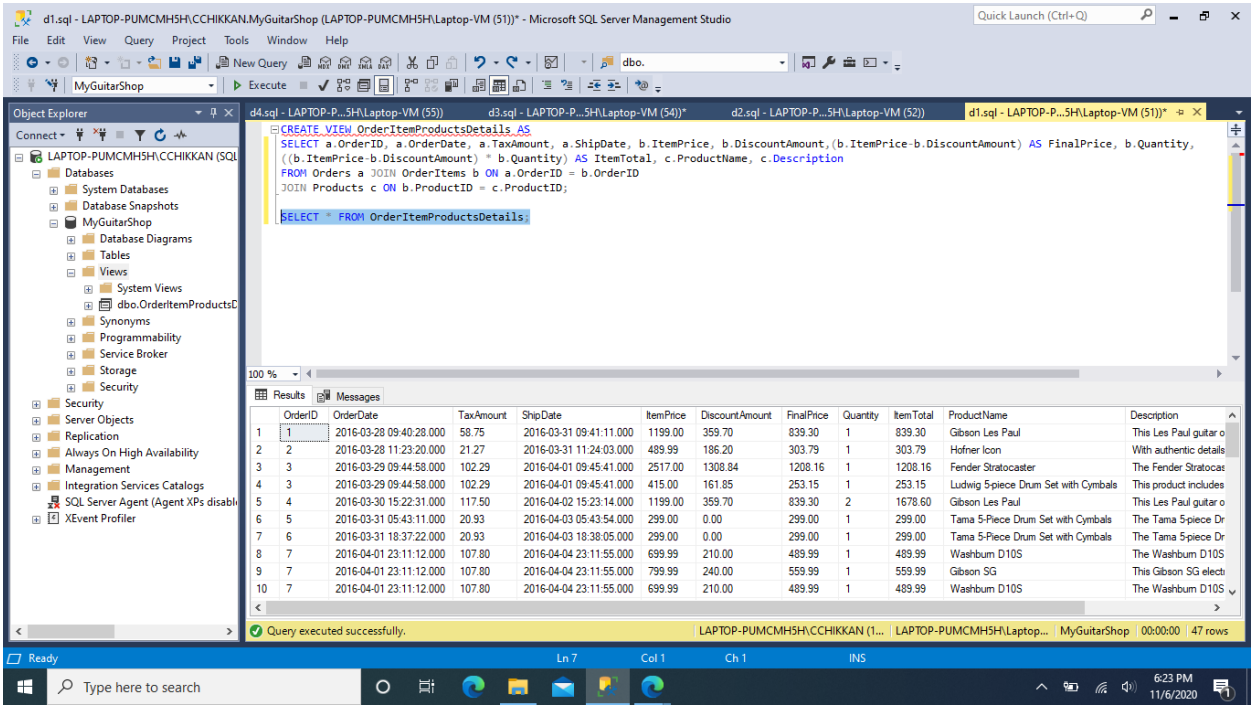
Solution:
The below is a CREATE VIEW statement that creates a view named OrderItemProductsDetails that returns the columns mentioned in the above question from the Orders, OrderItems, and Products tables.

```
CREATE VIEW OrderItemProductsDetails AS
SELECT a.OrderID, a.OrderDate, a.TaxAmount, a.ShipDate, b.ItemPrice,
b.DiscountAmount,(b.ItemPrice-b.DiscountAmount) AS FinalPrice, b.Quantity,
((b.ItemPrice-b.DiscountAmount) * b.Quantity) AS ItemTotal, c.ProductName,
c.Description
FROM Orders a JOIN OrderItems b ON a.OrderID = b.OrderID
JOIN Products c ON b.ProductID = c.ProductID;
```



The below is a SELECT statement written to verify the weather the view contains all the information mentioned in the question. So, executing the select statement below returns all the information contained in the view.

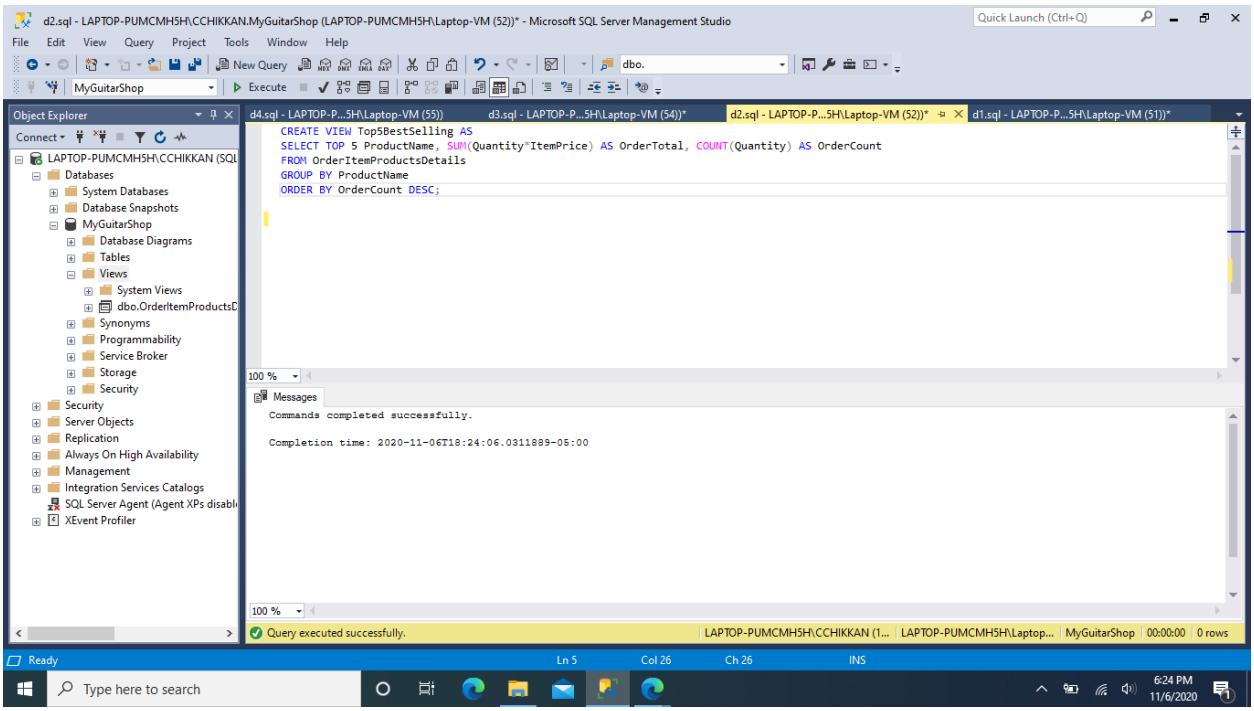
SELECT * FROM OrderItemProductsDetails;



2. [5] Create a view named Top5BestSelling that uses the view you created in Section D Question 1. This view should return some summary information about five best selling products. Each row should include these columns: ProductName, OrderTotal (the total sales for the product) and OrderCount (the number of times the product has been ordered).

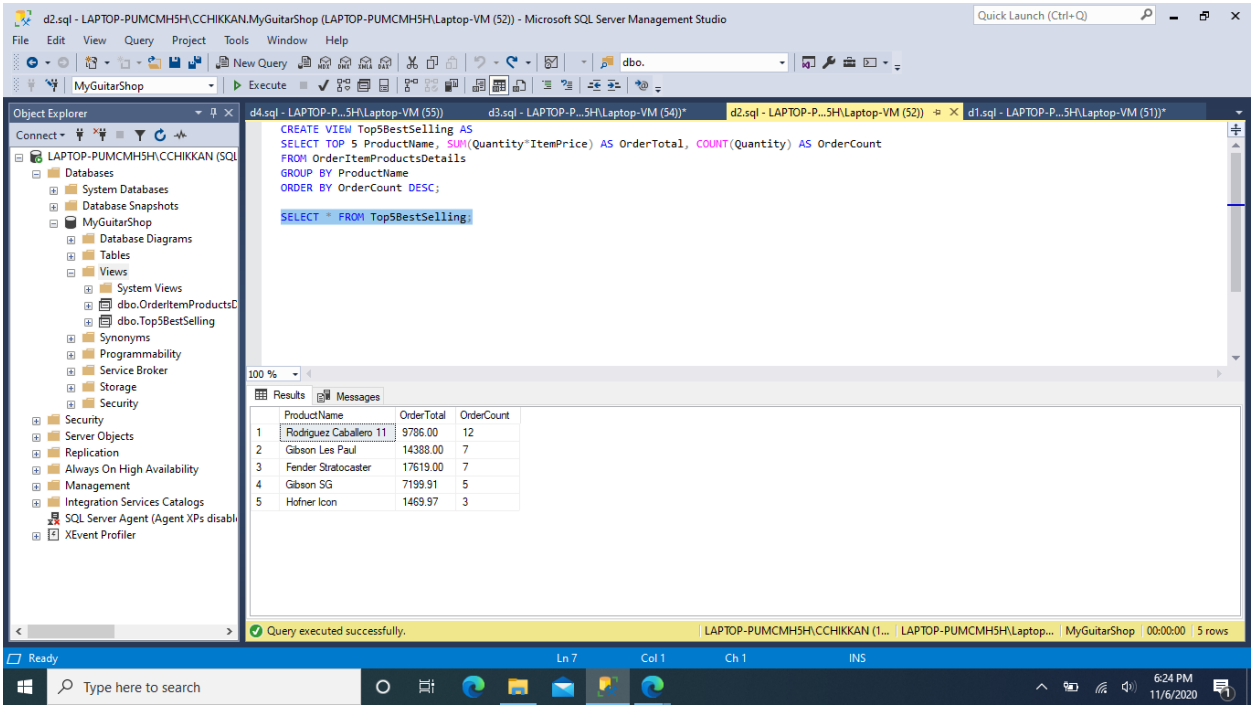
Solution:
The below is a CREATE VIEW statement that creates a view named Top5BestSelling that uses the view (OrderItemProductsDetails) which was created in the previous statement and returns the information about the 5 best-selling products.

CREATE VIEW Top5BestSelling AS
SELECT TOP 5 ProductName, SUM(Quantity*ItemPrice) AS OrderTotal,
COUNT(Quantity) AS OrderCount
FROM OrderItemProductsDetails
GROUP BY ProductName
ORDER BY OrderCount DESC;



Below is the SELECT statement written to verify the weather the view contains all the information mentioned in the question. Executing the select statement below returns all the information contained in the view.

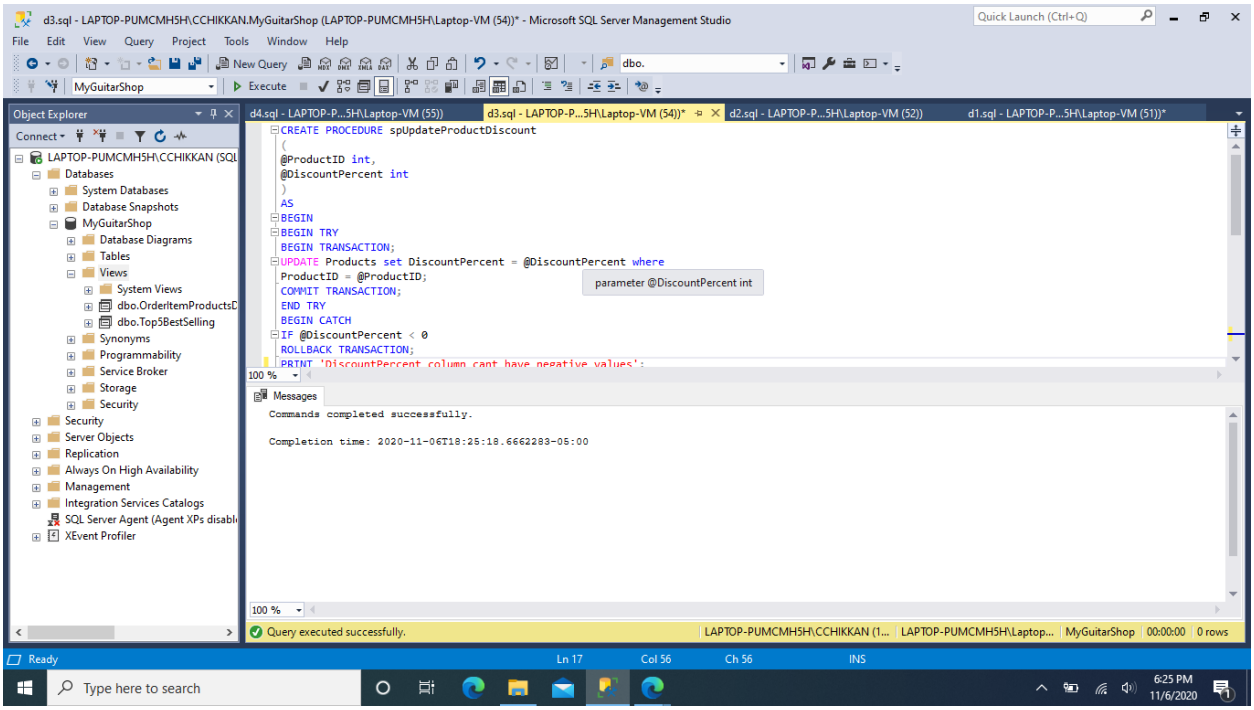
SELECT * FROM Top5BestSelling;



3. [5] Write a script that creates and calls a stored procedure named spUpdateProductDiscount that updates the DiscountPercent column in the Products table. This procedure should have one parameter for the product ID and another for the discount percent. If the value for the DiscountPercent column is a negative number, the stored procedure should raise an error that indicates that the value for this column must be a positive number. Code at least two EXEC statements that test this procedure.

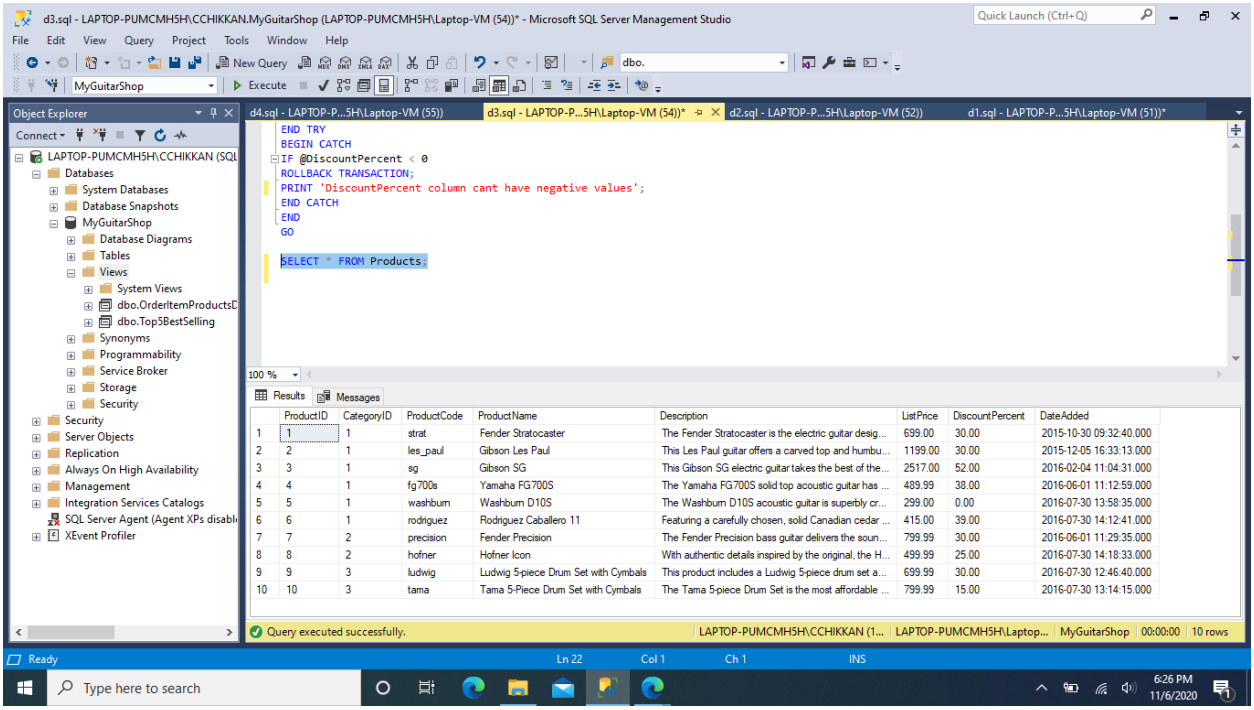
Solution:
The below script creates and calls a stored procedure named spUpdateProductDiscount that updates the DiscountPercent column in the Products table.
The procedure contains a single parameter for the product ID and another for the discount percent. If DiscountPercent column is a negative number, the procedure raises an error.

```
CREATE PROCEDURE spUpdateProductDiscount
(
@ProductID int,
@DiscountPercent int
)
AS
BEGIN
BEGIN TRY
BEGIN TRANSACTION;
UPDATE Products set DiscountPercent = @DiscountPercent where
ProductID = @ProductID;
COMMIT TRANSACTION;
END TRY
BEGIN CATCH
IF @DiscountPercent < 0
ROLLBACK TRANSACTION;
PRINT 'DiscountPercent column cant have negative values';
END CATCH
END
GO
```



The SELECT statement verifies weather the view contains all the information mentioned in the question. Executing the select statement below returns all the information contained in the view.

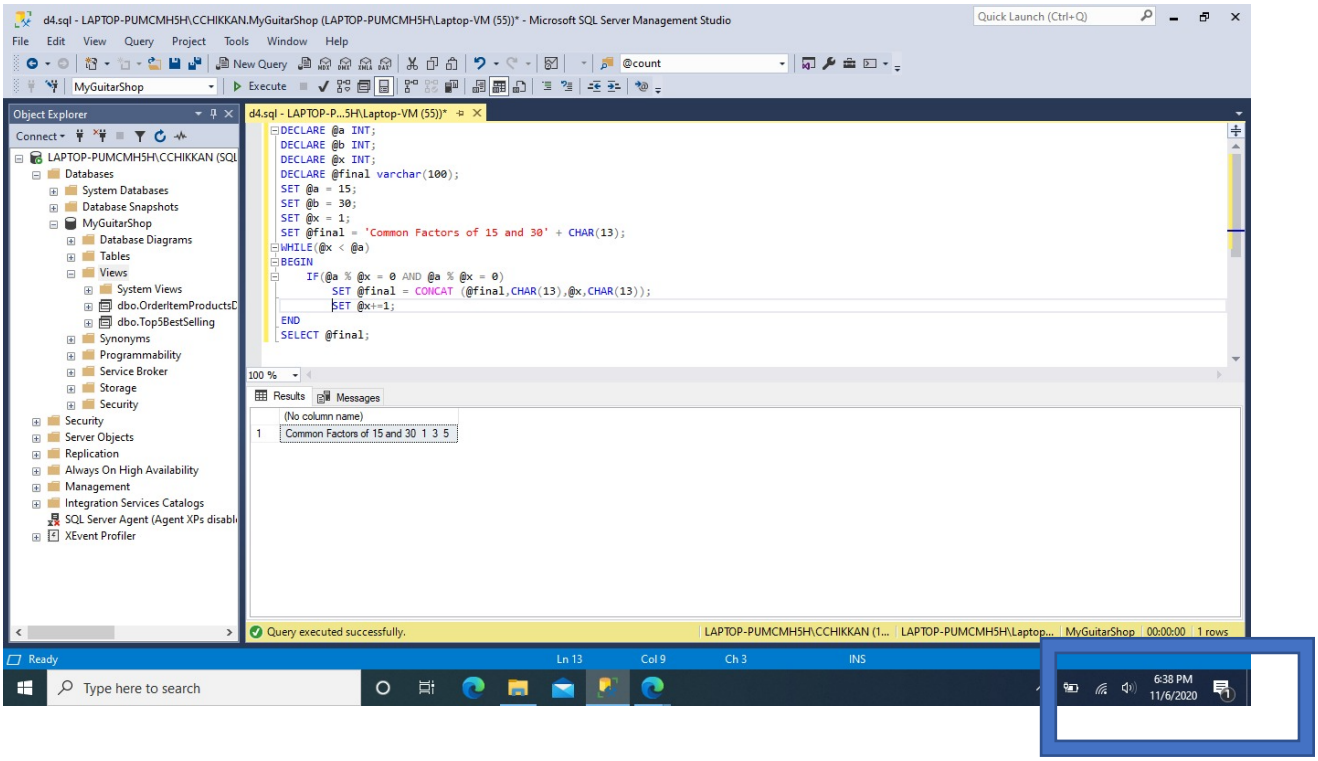
SELECT * FROM Products;



4. [5] Write a script that calculates the common factors between 15 and 30. To find a common factor, you can use the modulo operator (%) to check whether a number can be evenly divided into both numbers. Then, this script should print lines that display the common factors like this: Common factors of 15 and 30
   1
   3
   5
   15

Solution:
The below script calculates the common factors between 15 and 30

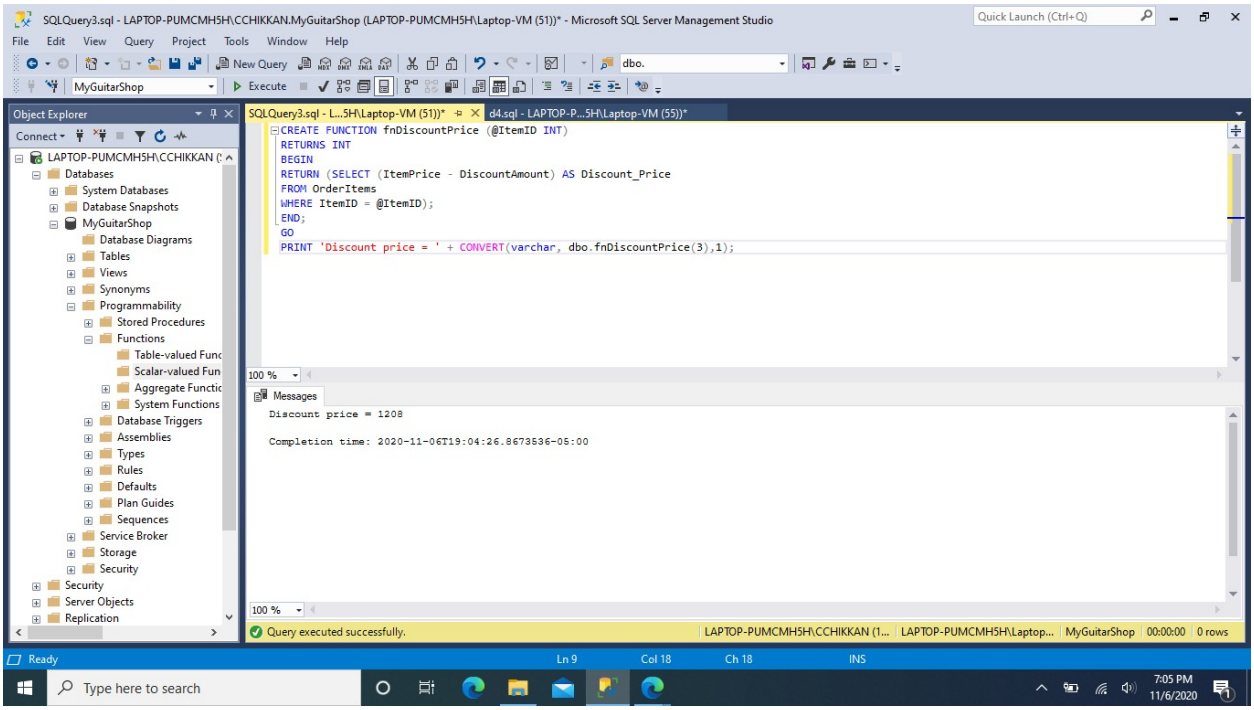
```
DECLARE @a INT;
DECLARE @b INT;
DECLARE @x INT;
DECLARE @final varchar(100);
SET @a = 15;
SET @b = 30;
SET @x = 1;
SET @final = 'Common Factors of 15 and 30' + CHAR(13);
WHILE(@x < @a)
BEGIN
        IF(@a % @x = 0 AND @a % @x = 0)
                SET @final = CONCAT (@final,CHAR(13),@x,CHAR(13));
                SET @x+=1;
END
SELECT @final;
```



5. [5] (1) Write a script that creates and calls a function named fnDiscountPrice that calculates the discount price of an item in the OrderItems table (discount amount subtracted from item price). To do that, this function should accept one parameter for the item ID, and it should return the value of the discount price for that item.
   (2) Write a script that creates and calls a function named fnItemTotal that calculates the total amount of an item in the OrderItems table (discount price multiplied by quantity). To do that, this function should accept one parameter for the item ID, it should use the DiscountPrice function that you created in (1), and it should return the value of the total for that item.

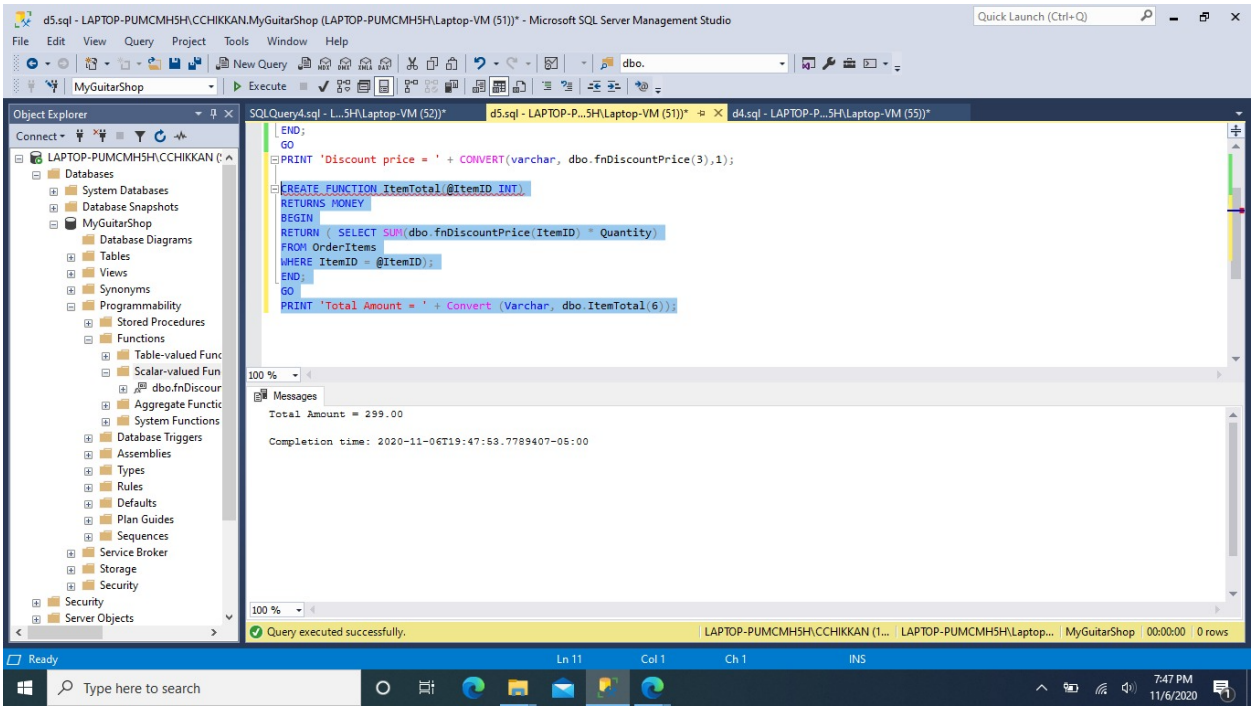
Solution:
The script creates and calls a function named fnDiscountPrice that calculates the discount price of an item in the OrderItems table. The function accepts a parameter for itemID and it returns the value of the discount price for that item.

```
CREATE FUNCTION fnDiscountPrice (@ItemID INT)
RETURNS INT
BEGIN
RETURN (SELECT (ItemPrice - DiscountAmount) AS Discount_Price
FROM OrderItems
WHERE ItemID = @ItemID);
END;
GO
PRINT 'Discount price = ' + CONVERT(varchar, dbo.fnDiscountPrice(3),1);
```



The script creates and calls a function named fnItemTotal that calculates the total amount of an item in the OrderItems table. The function here accepts a parameter for itemID and it uses the DiscountPrice function that was created above and it returns the value of the discount price for that item.

```
CREATE FUNCTION ItemTotal(@ItemID INT)
RETURNS MONEY
BEGIN
RETURN ( SELECT SUM(dbo.fnDiscountPrice(ItemID) * Quantity)
FROM OrderItems
WHERE ItemID = @ItemID);
END;
GO
PRINT 'Total Amount = ' + Convert (Varchar, dbo.ItemTotal(6));
```

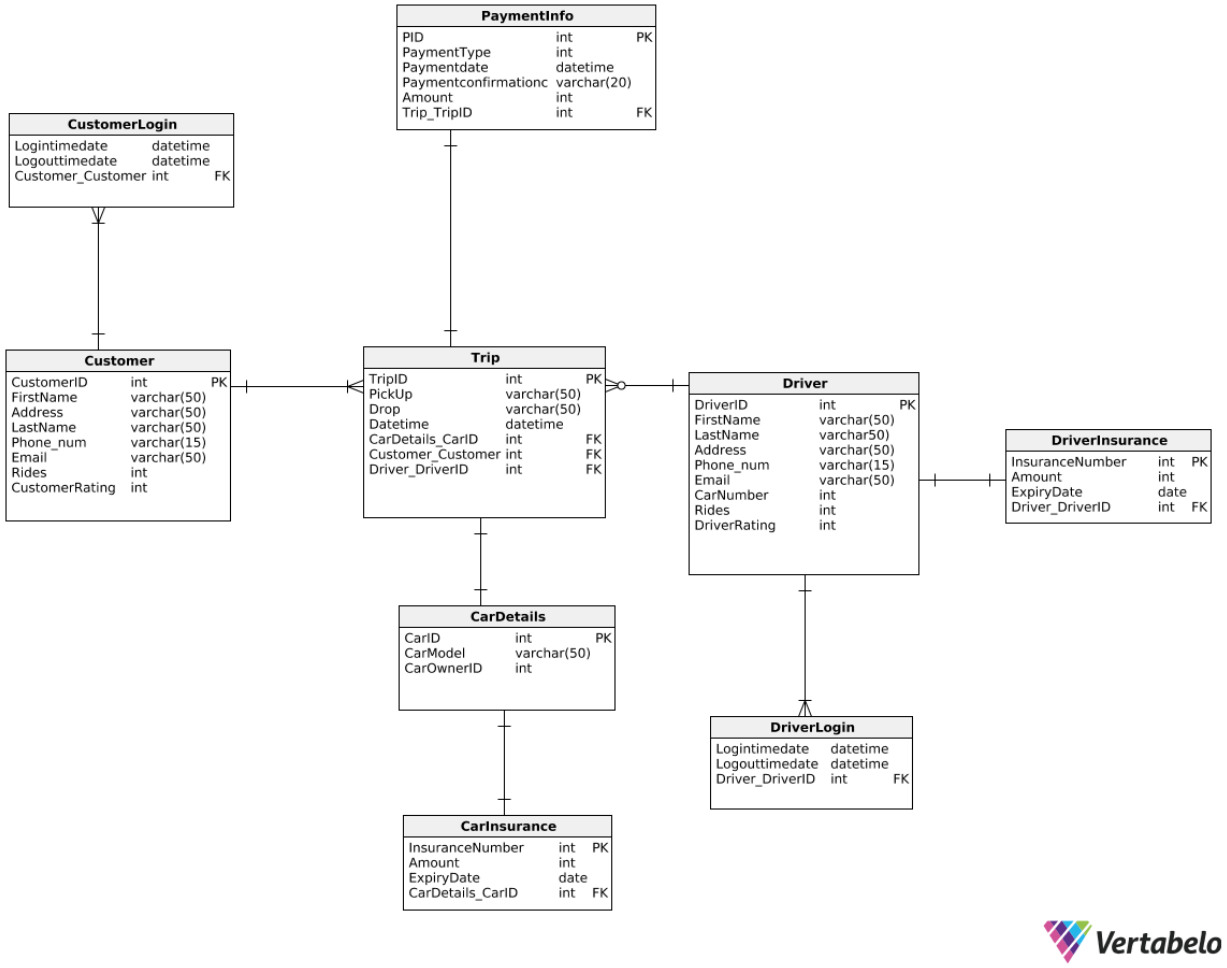


II. Database Design

Create a sample database design for Online Cab System and draw one database model for it. It needs to keep track of Customer information, Driver information, login details of both customer and driver, Trip information, Payment details used by customers to pay for a trip, Car details, Rating information given for driver by customer and for customer given by driver and Insurance information(for both car and drivers if applicable). Your design could add more things to the existing requirements, but all the given requirements should be met.

1. [3] Design the database that makes sense for the problem and select fields that make the most sense. A complete screenshot of your final design model is required.

Below is the screenshot of my final design model with all the tables and its attributes and the relationships among the tables.



2. [10] Determine the tables, columns, primary keys, nullabilities and show relationships between tables (one -one/one-many/many-many).

In this database design, I have included tables like Customer, Driver, Trip, CustomerLogin, DriverLogin, PaymentInfo, CarDetails, CarInsurance and DriverInsurance.

Customer:

- This table contains the customer details. The columns in the table are:
- CustomerID, FirstName, LastName, Address, Phone_num, Email, Rides and CustomerRating.
- CustomerID is the primary key in the table.
- CustomerRating column can be a nullable value when the driver chose not to give any ratings for a particular customer.

Driver:

- This table contains the driver details. The columns are:
- DriverID, FirstName, LastName, Address, Phone_num, Email, CarNumber, Rides and DriverRating

- DriverID is the primary key in the table.
- DriverRating column can be nullable when a customer chose not to give rating to the driver.

Trip:

- This table contains the information regarding the trip a driver and customer completed. The columns include:
- TripID, PickUp, Drop, Datetime, Customer_CustomerID, Driver_DriverID, CarDetails_CarID.
- TripID is the primary key and Customer_CustomerID, Driver_DriverID, arDetails_CarID are the foreign keys in the table.
- No nullable columns

PaymentInfo:

- This table contains the payment details of a completed trip. The columns are:
- PaymentID, PaymentType, PaymentDate, Paymentconfirmationcode, Amount, Trip_TripID.
- PaymentID is the primary key and Trip_TripID is the foreign key.
- No nullable column.

CustomerLogin:

- This table maintains the customer login details. The columns are:
- Logintimedate, logouttimedate, Customer_CustomerID.
- Customer_CustomerID is the foreign key.
- No nullable column

DriverLogin:

- This table maintains the customer login details. The columns are:
- Logintimedate, logouttimedate, Driver_DriverID.
- Driver_DriverID is the foreign key.
- No nullable column

CarDetails:

- This table contains the car details where the columns include:
- CarID, CarModel, CarOwnerID.
- CarID is the primary key

CarInsurance:

- This table contains the car insurance details with columns:
- InsuranceNumber, Amount, Expirydate, CarDetails_CarID.
- InsuranceNumber is the primark key and CarDetails_CarID is the foreign key.

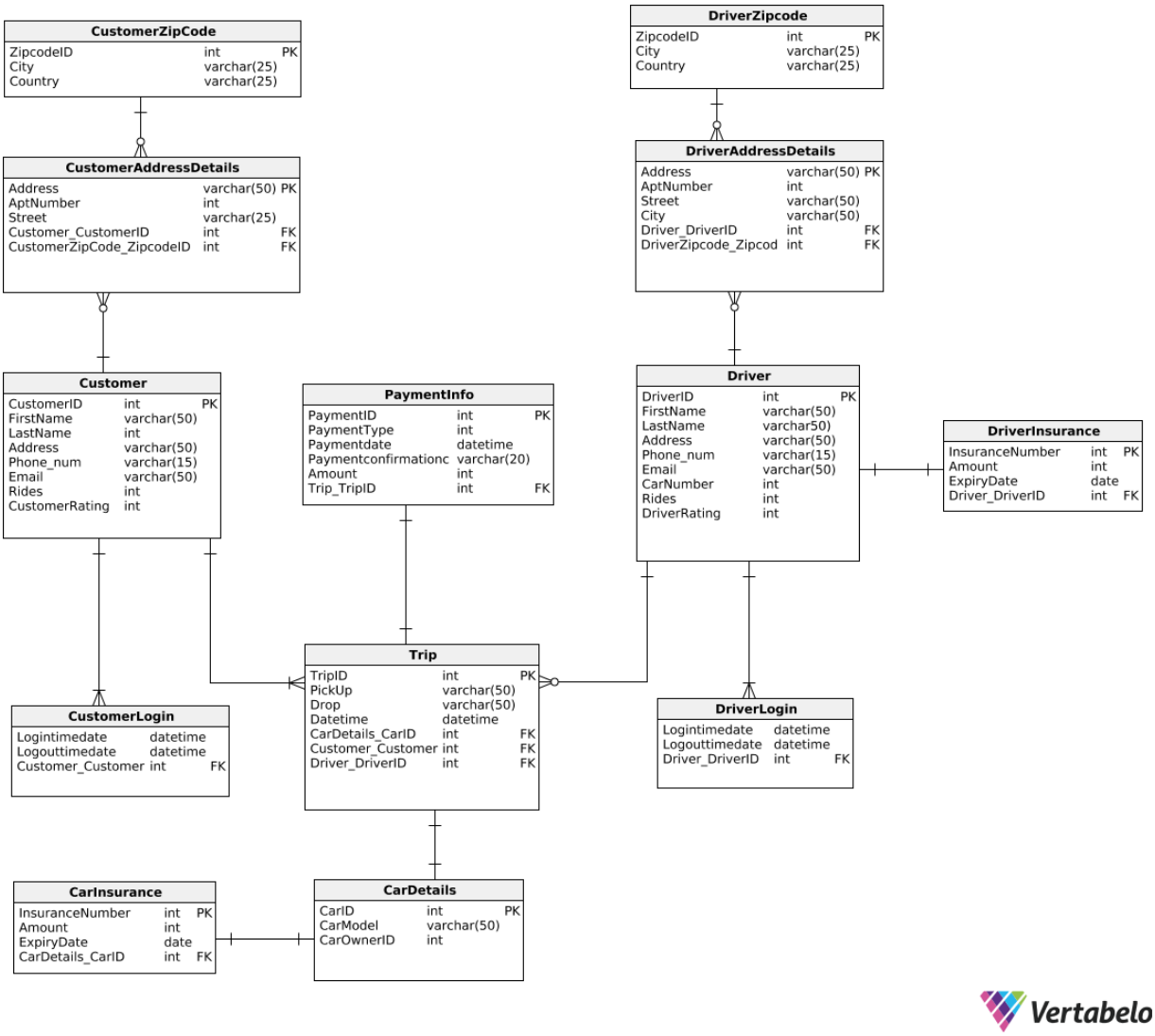
DriverInsurance:

- This table contains the car insurance details with columns:
- InsuranceNumber, Amount, Expirydate, Driver_DriverID.
- InsuranceNumber is the primark key and Driver_DriverID is the foreign key.

Relationships:

- There is one-to-many relationship between Customer and Trip tables.
- There is one-to-many relationship between Driver and Trip tables.
- There is one-to-many relationship between Customer and CustomerLogin tables.
- There is one-to-many relationship between Driver and DriverLogin tables.
- There is one-to-one relationship between PaymentInfo and Trip tables.
- There is one-to-one relationship between CarDetails and CarInsurance
- There is one-to-one relationship between DriverDetails and DriverInsurance

3. [5] Normalize your design into 3rd Normal Form. Please use MS Visio, Vertabelo or any similar database design tool.



4. [2] Explain your design, including relationship between tables.

In the above design, I have added 4 more tables to normalize the database they are:

CustomerAddressDetails:

- This table contains the customer address details with 5 columns
- Address is the primary key and Customer_CustomerID, CustomerZipcode_ZipcodeID are foreign key.

DriverAddressDetails:

- This table contains the driver address details with 5 columns
- Address is the primary key and Driver_DriverID, CustomerZipcode_ZipcodeID are foreign key.

CustomerZipCode:

- ZipcodeID is the primary key.

DriverZipCode:

- ZipcodeID is the primary key.

Relationships:

- There is one-to-many relationship between Customer and CustomerAddressDetails
- There is one-to-many relationship between Customer and CustomerAddressDetails
- There is one-to-many relationship between CustomerAddressDetails and CustomerZipcode
- There is one-to-many relationship between DriverAddressDetails and DriverZipcode

For any table to be in SecondNormalForm, it should be in 1-NF and all non-key attributes must depend on the entire primary key but not on the partial partial primary key. In Customer and Driver table, Address column can be created as a separate table which contains further more information. The CustomerAddressDetails contain AddressID as primary key which is foreign key in Customer table and same in Driver and DriverAddressDetails table.

For any table to be in ThirdNormalForm, it should be in 1-NF, 2-NF and every column should depend only on the primary key and should not depend on the non-primary key.So here in my design ZipCode and CityName columns in a AddressDetails Table are dependent on each other. For example if ZipCode changes CityName value changes and also Some set of cities have same ZipCode. So we have to eliminate this. This can be eliminated by moving the ZipCode and CityName columns into a new table named ZipCode. These values in AddressDetails tables are replaced by primary code of ZipCode table i.e. by ZipCodeID which acts as foreign key to AddressDetails table.

Remarks:

By completing this project, I got user experience on various skills like Creating a new database, adding tables to it, assigning various datatypes each column by defining the size and updating and deleting the previously entered data.

Additional to creating and modifying the existing tables I even learnt how to output the user selected data from the database by using SELECT statements.
I also got to know the various operations performed on the data such as arithmetic, logical and analytical operations and I got a good hold on using JOINS in SQL to in two or more tables to return different columns from more than one table simultaneously.
Creating views are important in DBMS to manage the result set of large data and I learnt how to create a view which can be used as temporary tables and also writing scripts that perform various operations simultaneously by accepting a single input parameter and output the data after executing the script.

In the last part of the project, I learnt how to design a database and how to eliminate redundancy and various kind of dependencies like partial dependency and transitive dependency and how

to convert my database design into various normal forms. By creating a database, I even analyzed the relationships between the tables and the importance of primary and foreign key.

Finally, completing this project gave me knowledge on creating a database on my own and also performing various data manipulations on that.