
Sebastian Huynh
Project 2: SQL Database
October 31, 2023
CIS 3050-04
Fall 2023

Table of Contents

1. Table of Contents…….. 1
2. Statement of Academic Honesty…….. 2
3. Introduction…….. 3
4. Project Descriptions…….. 4
5. Test Queries…….. 5-27
6. Results & Discussions…….. 28
7. Lessons Learned…….. 28
8. Conclusion…….. 29
9. Cited References…….. 30

Statement of Academic Honesty

My name is: Sebastian Huynh, I declare that, except where fully referenced, no aspect of this project has been copied from any other source. I understand that any act of Academic Dishonesty such as plagiarism or collusion may result in serious offense and punishments. I promise not to lie about my academic work, to cheat, or to steal the words or ideas of others, nor will I help fellow students to violate the Code of Academic Honesty.

Name: Sebastian Huynh
Date: Oct 31, 2023

## Introduction

Previously in project 1, we explored the integrity and structure of relational databases using visual data modeling. By highlighting meaningful connections between entities, companies are able to organize large amounts of input for advanced purposes such as enhanced decision-making and predictive analysis (that go beyond just record keeping). Proceeding to project 2, the focus shifts from database representations and toward the creation of an actual, working database that adheres to desired data model guidelines. Structured query language has become one of the most popular methods for communication taking place to and from databases and data-related applications (Hoffer, 2018). It is one of the standardized ways of creating and manipulating a working relational database. Furthermore, there are three subcategories for writing SQL which range from data definition, data manipulation, and data control, with the first two being essential for development in a testing-only environment. Data definition SQL statements "specify a database's structure" (Panigrahi, 2023) while data manipulation SQL statements alter or retrieve data from an already established database. Project 2 consists primarily of such DDL and DML solutions.

## Project Description

Software Required: Microsoft SQL Server Management Studio, SQL Server Express or Developer, and Windows OS

- Management Studio is the encompassing program and integrated environment that provides user-friendly tools as a bridge to databases hosted by SQL Server
- SQL Server is Microsoft's relational database system software that holds all database information and can be accessed by other programs like management studio.
- Management Studio and SQL Server can only be installed on Windows, but Linux users may also use SQL Server if they configure it.

Project Objective: Demonstrate the ability to develop a relational database using MSSQL for the order fulfillment process of a company.

- With input values given, create six tables, one for each entity using DDL statements. Establish relationships between entities using constraints.
- Run DML statements that will insert the input values into the empty tables then use select statements to bring back filtered results according to project instructions. Also, demonstrate the ability to insert and delete data from specified tables.

**Query 1**: Write a query that displays a list of all customers showing the customer's last name, and phone number. Sort the results by customer last name, then phone number.

SELECT customer_last_name, customer_phone FROM customers ORDER BY customer_last_name ASC, customer_phone ASC;

query_1.sql - (loc...MCK6AGK\crota (57))

```
SELECT customer_last_name, customer_phone
FROM customers
ORDER BY customer_last_name ASC, customer_phone ASC;
```

100 %

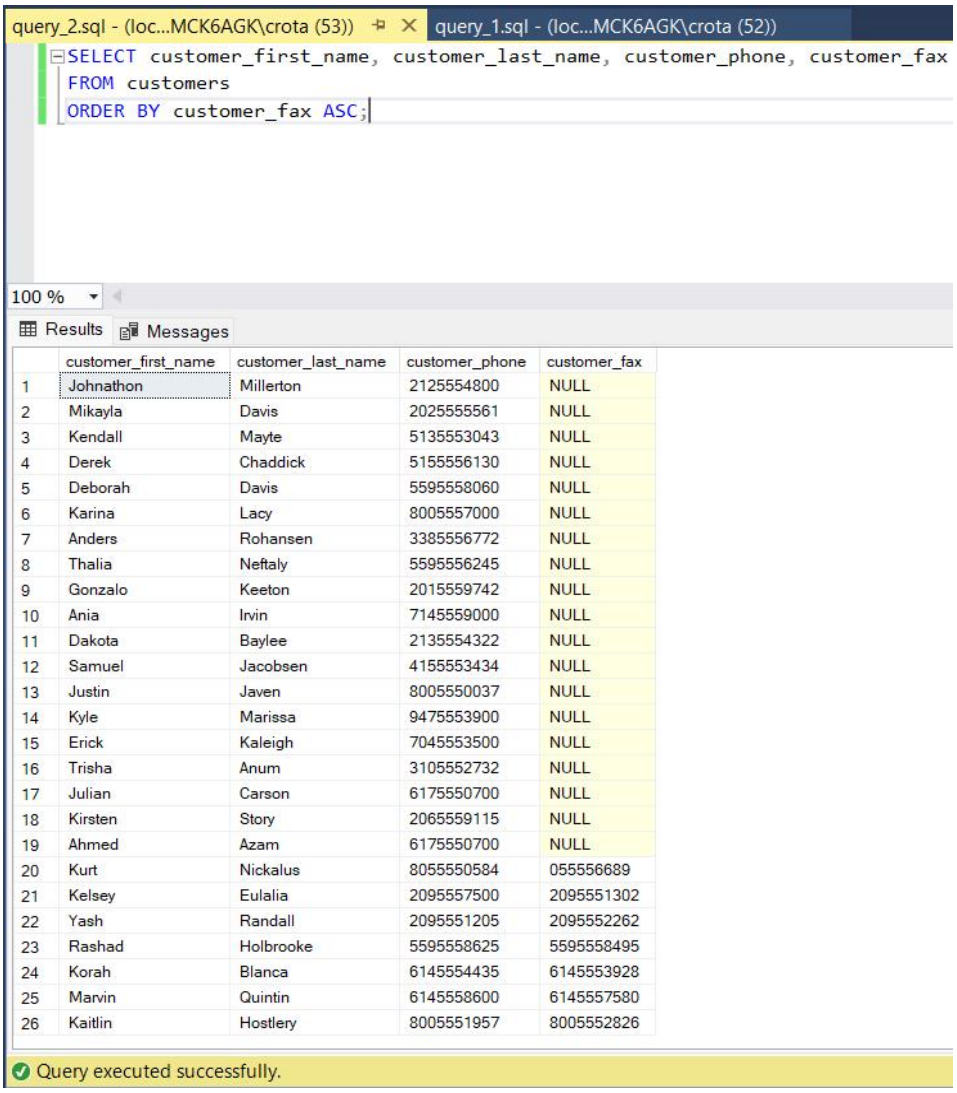
Results | Messages

| | customer_last_name | customer_phone |
|---|---|---|
| 1 | Anum | 3105552732 |
| 2 | Azam | 6175550700 |
| 3 | Baylee | 2135554322 |
| 4 | Blanca | 6145554435 |
| 5 | Carson | 6175550700 |
| 6 | Chaddick | 5155556130 |
| 7 | Davis | 2025555561 |
| 8 | Davis | 5595558060 |
| 9 | Eulalia | 2095557500 |
| 10 | Holbrooke | 5595558625 |
| 11 | Hostlery | 8005551957 |
| 12 | Irvin | 7145559000 |
| 13 | Jacobsen | 4155553434 |
| 14 | Javen | 8005550037 |
| 15 | Kaleigh | 7045553500 |
| 16 | Keeton | 2015559742 |
| 17 | Lacy | 8005557000 |
| 18 | Marissa | 9475553900 |
| 19 | Mayte | 5135553043 |
| 20 | Millerton | 2125554800 |
| 21 | Neftaly | 5595556245 |
| 22 | Nickalus | 8055550584 |
| 23 | Quintin | 6145558600 |
| 24 | Randall | 2095551205 |
| 25 | Rohansen | 3385556772 |
| 26 | Story | 2065559115 |

Explanation: Showing only two column fields, customer's last name and their corresponding phone number from the customer's table. Resulting rows are displayed from A-Z according to their last names and then smallest to largest according to their phone numbers. This is sorted by default ascending order.

**Query 2**: Write a query that displays a list of all customers showing the customer's first name, last name, phone number and fax. Sort the results by customer fax number in ascending order.

SELECT customer_first_name, customer_last_name, customer_phone, customer_fax FROM customers ORDER BY customer_fax ASC;



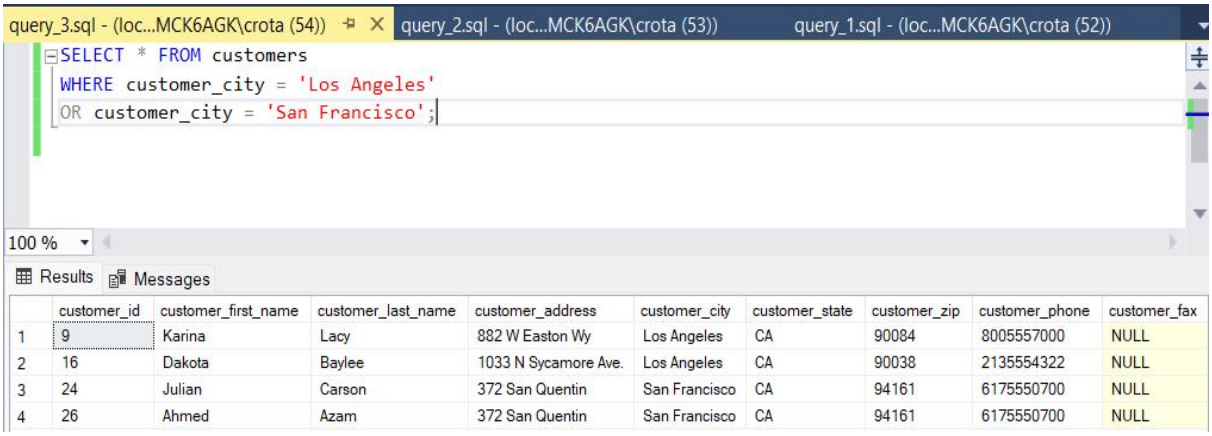

| | customer_first_name | customer_last_name | customer_phone | customer_fax |
|---|---|---|---|---|
| 1 | Johnathon | Millerton | 2125554800 | NULL |
| 2 | Mikayla | Davis | 2025555561 | NULL |
| 3 | Kendall | Mayte | 5135553043 | NULL |
| 4 | Derek | Chaddick | 5155556130 | NULL |
| 5 | Deborah | Davis | 5595558060 | NULL |
| 6 | Karina | Lacy | 8005557000 | NULL |
| 7 | Anders | Rohansen | 3385556772 | NULL |
| 8 | Thalia | Neftaly | 5595556245 | NULL |
| 9 | Gonzalo | Keeton | 2015559742 | NULL |
| 10 | Ania | Irvin | 7145559000 | NULL |
| 11 | Dakota | Baylee | 2135554322 | NULL |
| 12 | Samuel | Jacobsen | 4155553434 | NULL |
| 13 | Justin | Javen | 8005550037 | NULL |
| 14 | Kyle | Marissa | 9475553900 | NULL |
| 15 | Erick | Kaleigh | 7045553500 | NULL |
| 16 | Trisha | Anum | 3105552732 | NULL |
| 17 | Julian | Carson | 6175550700 | NULL |
| 18 | Kirsten | Story | 2065559115 | NULL |
| 19 | Ahmed | Azam | 6175550700 | NULL |
| 20 | Kurt | Nickalus | 8055550584 | 055556689 |
| 21 | Kelsey | Eulalia | 2095557500 | 2095551302 |
| 22 | Yash | Randall | 2095551205 | 2095552262 |
| 23 | Rashad | Holbrooke | 5595558625 | 5595558495 |
| 24 | Korah | Blanca | 6145554435 | 6145553928 |
| 25 | Marvin | Quintin | 6145558600 | 6145557580 |
| 26 | Kaitlin | Hostlery | 8005551957 | 8005552826 |

Query executed successfully.

Explanation: Showing only four column fields, a customer's first and last name and then their corresponding phone number and fax number. These fields are derived from the customers table and the resulting rows are sorted from least to greatest (ascending) fax numbers with the least starting from no value or null.

**Query 3**: Write a query that displays all the customers from San Francisco or Los Angeles in the “Customers” table.

SELECT * FROM customers WHERE customer_city = 'Los Angeles' OR customer_city = 'San Francisco';

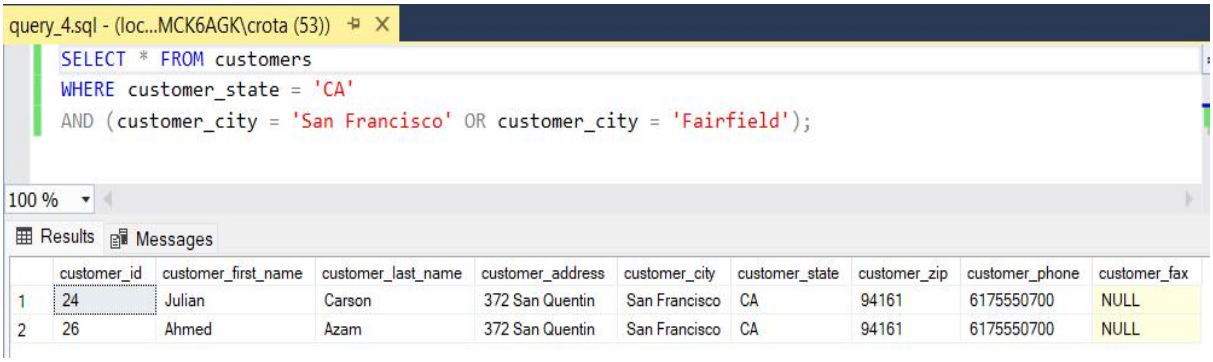



| | customer_id | customer_first_name | customer_last_name | customer_address | customer_city | customer_state | customer_zip | customer_phone | customer_fax |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 9 | Karina | Lacy | 882 W Easton Wy | Los Angeles | CA | 90084 | 8005557000 | NULL |
| 2 | 16 | Dakota | Baylee | 1033 N Sycamore Ave. | Los Angeles | CA | 90038 | 2135554322 | NULL |
| 3 | 24 | Julian | Carson | 372 San Quentin | San Francisco | CA | 94161 | 6175550700 | NULL |
| 4 | 26 | Ahmed | Azam | 372 San Quentin | San Francisco | CA | 94161 | 6175550700 | NULL |

Explanation: This statement shows all available column fields from the customers table for each row having the condition that the customer is either from the city of Los Angeles or San Francisco being met. A customer cannot be part from more than one city according to the field specifications so a customer instance will be returned if just one of these two conditions are met (Los Angeles being the first and San Francisco being the second).

**Query 4**: Write a query that displays all the customers from the state of California and live in San Francisco or Fairfield.

SELECT * FROM customers WHERE customer_state = 'CA'
AND (customer_city = 'San Francisco' OR customer_city = 'Fairfield');

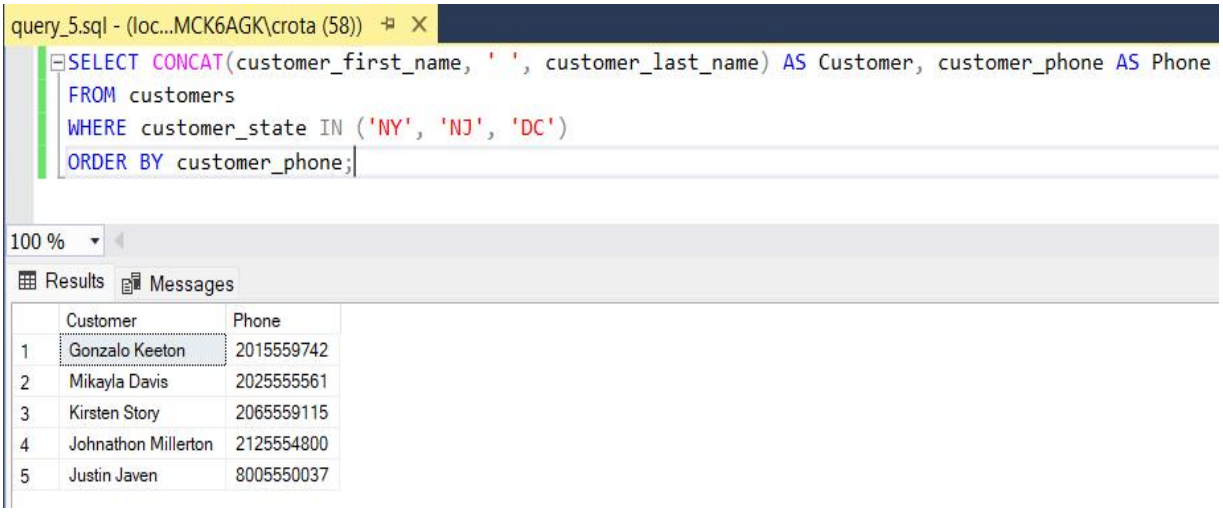



| | customer_id | customer_first_name | customer_last_name | customer_address | customer_city | customer_state | customer_zip | customer_phone | customer_fax |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 24 | Julian | Carson | 372 San Quentin | San Francisco | CA | 94161 | 6175550700 | NULL |
| 2 | 26 | Ahmed | Azam | 372 San Quentin | San Francisco | CA | 94161 | 6175550700 | NULL |

Explanation: This statement shows all available column fields from the customers table, returning every row which meets the condition that the customer is living in either the city of San Francisco or the city of Fairfield, located only in the state of California. This means that any customers who live in a different Fairfield for example outside of California is not returned in the results.

**Query 5**: Write a query that displays each customer name as a single field in the format “firstname lastname” with a heading of Customer, along with their phone number with a heading of Phone. Use the IN operator to only display customers in New York, New Jersey, or Washington D.C. Sort the results by phone number.

SELECT CONCAT(customer_first_name, ' ', customer_last_name) AS Customer, customer_phone AS Phone FROM customers WHERE customer_state IN ('NY', 'NJ', 'DC') ORDER BY customer_phone;

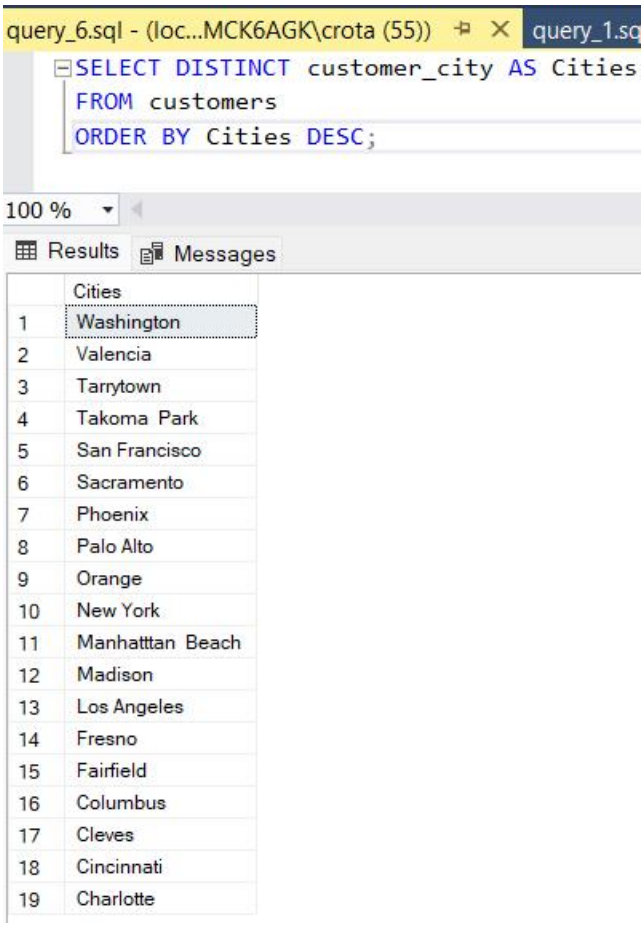



| | Customer | Phone |
|---|---|---|
| 1 | Gonzalo Keeton | 2015559742 |
| 2 | Mikayla Davis | 2025555561 |
| 3 | Kirsten Story | 2065559115 |
| 4 | Johnathon Millerton | 2125554800 |
| 5 | Justin Javen | 8005550037 |

Explanation: The first field returned is a string combination of customers’ first names followed by a space and then their last name for each returned row. The alias name for this first field is ‘Customer’. For the second field, the column containing customer phone numbers is returned under the alias ‘Phone’. Everything is derived from the customers table. Everything is derived from the customers table and only customers who live in either New York, New Jersey, or Washington DC are shown in ascending (least to greatest) order of their corresponding phone numbers.

**Query 6**: Write a query that will list all the cities that have customers with a heading of Cities. Only list each city once (no duplicates) and sort in descending alphabetical order.

SELECT DISTINCT customer_city AS Cities FROM customers ORDER BY Cities DESC;

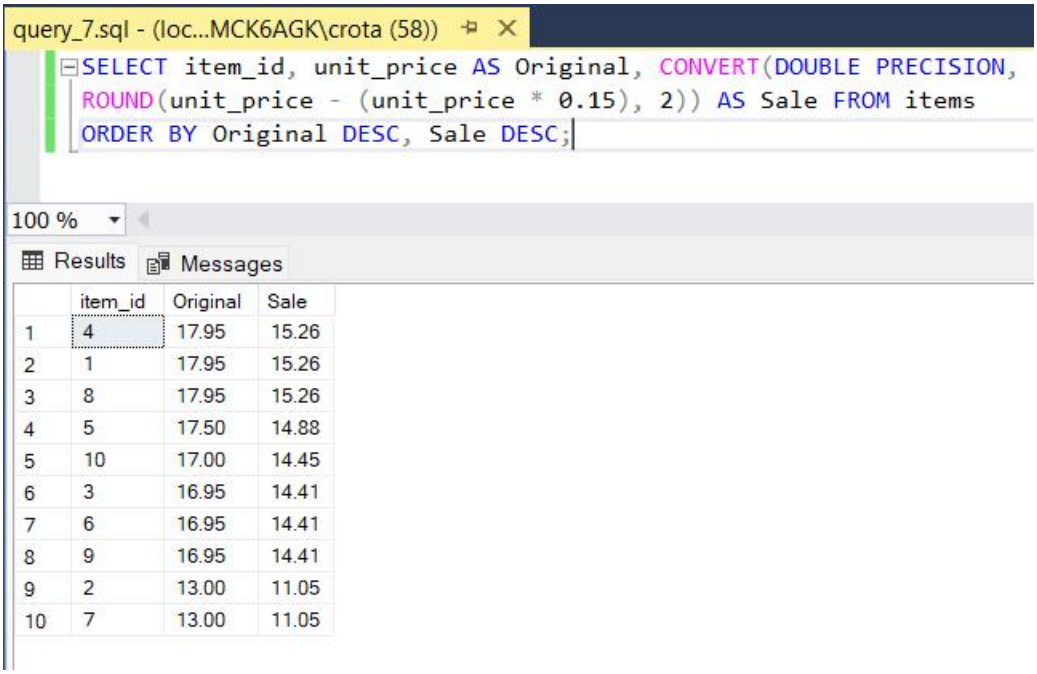



| | Cities |
|---|---|
| 1 | Washington |
| 2 | Valencia |
| 3 | Tarrytown |
| 4 | Takoma Park |
| 5 | San Francisco |
| 6 | Sacramento |
| 7 | Phoenix |
| 8 | Palo Alto |
| 9 | Orange |
| 10 | New York |
| 11 | Manhatttan Beach |
| 12 | Madison |
| 13 | Los Angeles |
| 14 | Fresno |
| 15 | Fairfield |
| 16 | Columbus |
| 17 | Cleves |
| 18 | Cincinnati |
| 19 | Charlotte |

Explanation: This statement returns only one field with the alias name ‘Cities’ because we only want to see the list of possible cities a customer in our database could be from. By using the distinct descriptor any city that appears will only show up once in the results even if they appear in the tables multiple times. It is derived from the customers table.

**Query 7**: Write a query that displays the title of each item along with the price (with a heading of Original) and a calculated field reflecting the price with a 15% discount (with a heading of Sale). Display the sale price with two decimal places using the ROUND function. Sort by price from highest to lowest.

SELECT item_id, unit_price AS Original, CONVERT(DOUBLE PRECISION, ROUND(unit_price - (unit_price * 0.15), 2)) AS Sale FROM items ORDER BY Original DESC, Sale DESC;

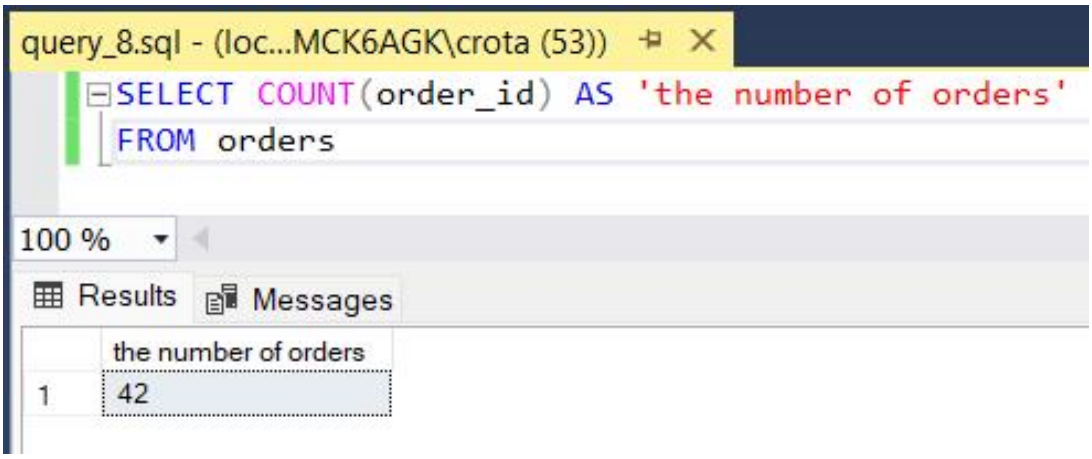



| | item_id | Original | Sale |
|---|---|---|---|
| 1 | 4 | 17.95 | 15.26 |
| 2 | 1 | 17.95 | 15.26 |
| 3 | 8 | 17.95 | 15.26 |
| 4 | 5 | 17.50 | 14.88 |
| 5 | 10 | 17.00 | 14.45 |
| 6 | 3 | 16.95 | 14.41 |
| 7 | 6 | 16.95 | 14.41 |
| 8 | 9 | 16.95 | 14.41 |
| 9 | 2 | 13.00 | 11.05 |
| 10 | 7 | 13.00 | 11.05 |

Explanation: In the results, the first two columns are originally from the table of items and are used to calculate the third column which is the new price after a 15% discount had been applied to each item. To get the discounted price, we take the original price and subtract it from 15% of the original price. This filtered query allows us to see how much of a difference the original and sales prices are. The round function allows the decimal number to format to cents and double precision is specified to truncate any leading zeros after the 2nd decimal place.

**Query 8**: Write a query that displays the number of orders.

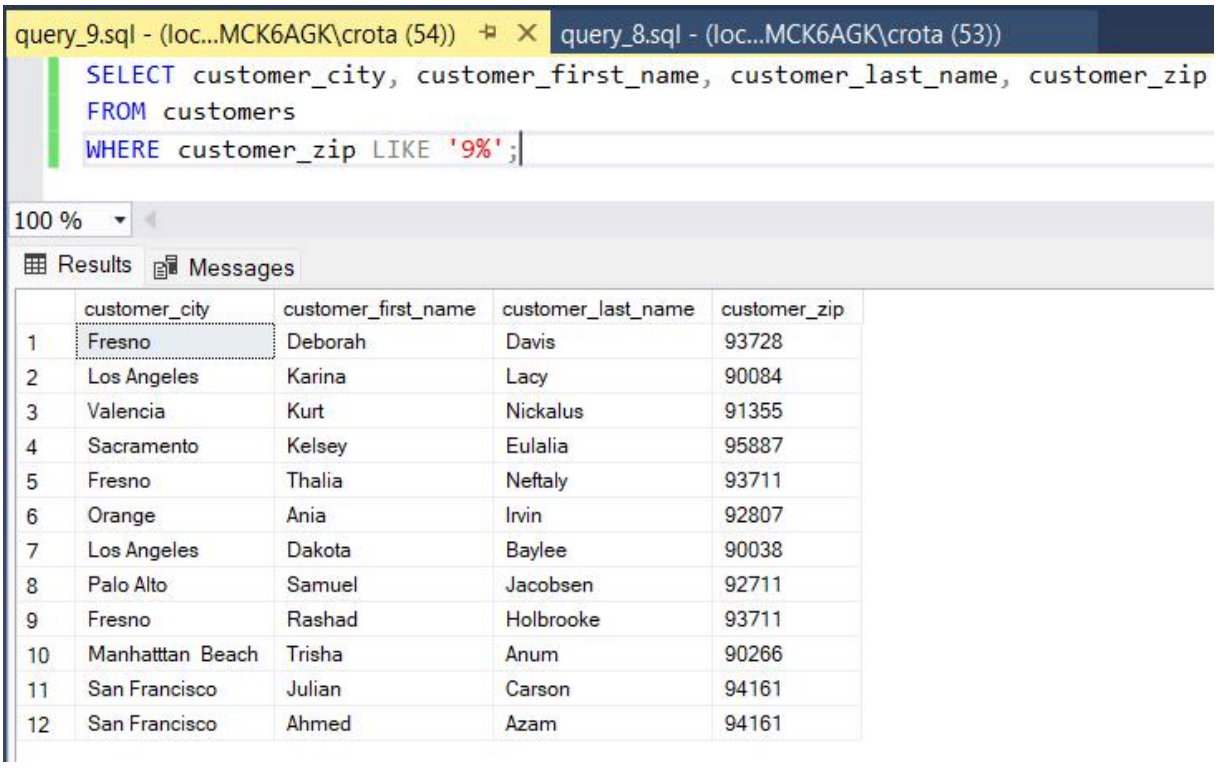
```
SELECT COUNT(order_id) AS 'the number of orders'
FROM orders
```




Explanation: From the orders table, we are interested the column field of order_id to select unique orders. This is important as we don't want to count the quantity of each order but only how many different orders there are in total. Using the count function we can aggregate this amount.

**Query 9**: Write a query that displays the customer city, first name, last name, and zip code from the customer's table. Use the LIKE operator to only display customers that reside in any zip code beginning with 9.

SELECT customer_city, customer_first_name, customer_last_name, customer_zip FROM customers WHERE customer_zip LIKE '9%';

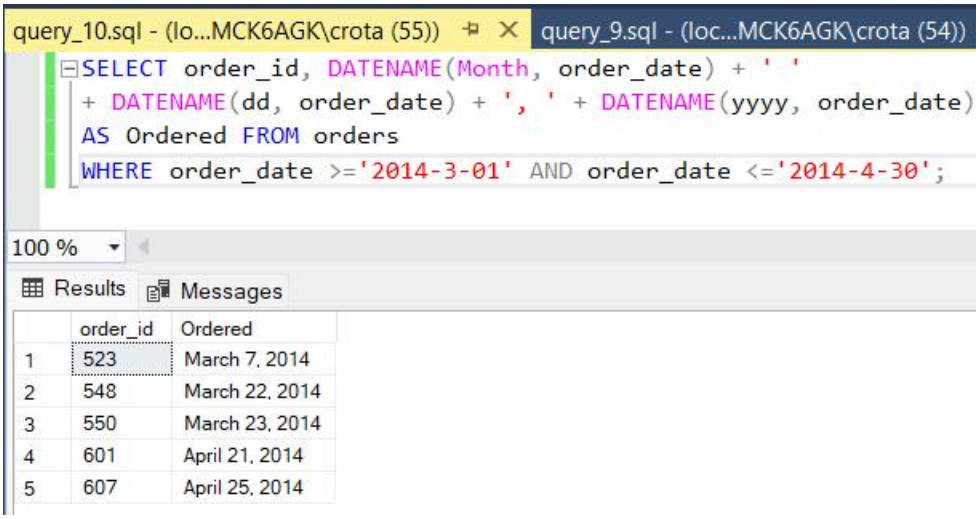



| | customer_city | customer_first_name | customer_last_name | customer_zip |
|---|---|---|---|---|
| 1 | Fresno | Deborah | Davis | 93728 |
| 2 | Los Angeles | Karina | Lacy | 90084 |
| 3 | Valencia | Kurt | Nickalus | 91355 |
| 4 | Sacramento | Kelsey | Eulalia | 95887 |
| 5 | Fresno | Thalia | Neftaly | 93711 |
| 6 | Orange | Ania | Irvin | 92807 |
| 7 | Los Angeles | Dakota | Baylee | 90038 |
| 8 | Palo Alto | Samuel | Jacobsen | 92711 |
| 9 | Fresno | Rashad | Holbrooke | 93711 |
| 10 | Manhatttan Beach | Trisha | Anum | 90266 |
| 11 | San Francisco | Julian | Carson | 94161 |
| 12 | San Francisco | Ahmed | Azam | 94161 |

Explanation: Selecting four customer-related column fields from the customers table. In this query, the only records/rows that return are for customers who have zip codes beginning with the number 9. To accommodate for the different numbers after the initial 9 we use the percent sign when putting the condition for customer_zip.

**Query 10**: Write a query that displays the order id and order date for any orders placed from March 1, 2014 through April 30, 2014. Do this WITHOUT using the BETWEEN clauses. Format the date field as Month dd, yyyy and use a heading of "Ordered".

SELECT order_id, DATENAME(Month, order_date) + ' ' + DATENAME(dd, order_date) + ', ' + DATENAME(yyyy, order_date) AS Ordered FROM orders WHERE order_date >='2014-3-01' AND order_date <='2014-4-30';

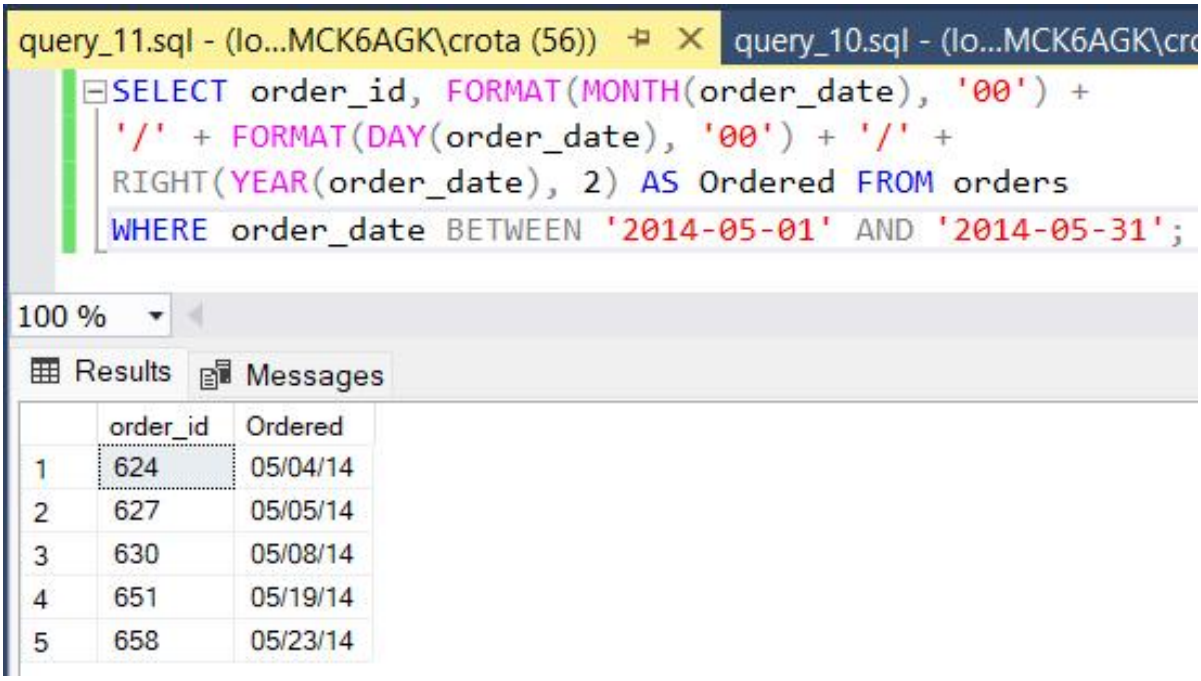



| | order_id | Ordered |
|---|---|---|
| 1 | 523 | March 7, 2014 |
| 2 | 548 | March 22, 2014 |
| 3 | 550 | March 23, 2014 |
| 4 | 601 | April 21, 2014 |
| 5 | 607 | April 25, 2014 |

Explanation: By not using the between clauses, the alternative is by using comparison operators to create a range from start date and end date. For this query we select the order id field column from orders and then concatenate string for the second returned field using the addition operator. The date name function returns the specific name or number of a date given the time period (month, day, year, etc).

**Query 11**: Write a query that displays the order id and order date for any orders placed during the month of May 2014. Do this using the BETWEEN clauses. Format the date field as mm/dd/yy and use a heading of "Ordered".

SELECT order_id, FORMAT(MONTH(order_date), '00') + '/' + FORMAT(DAY(order_date), '00') + '/' + RIGHT(YEAR(order_date), 2) AS Ordered FROM orders WHERE order_date BETWEEN '2014-05-01' AND '2014-05-31';

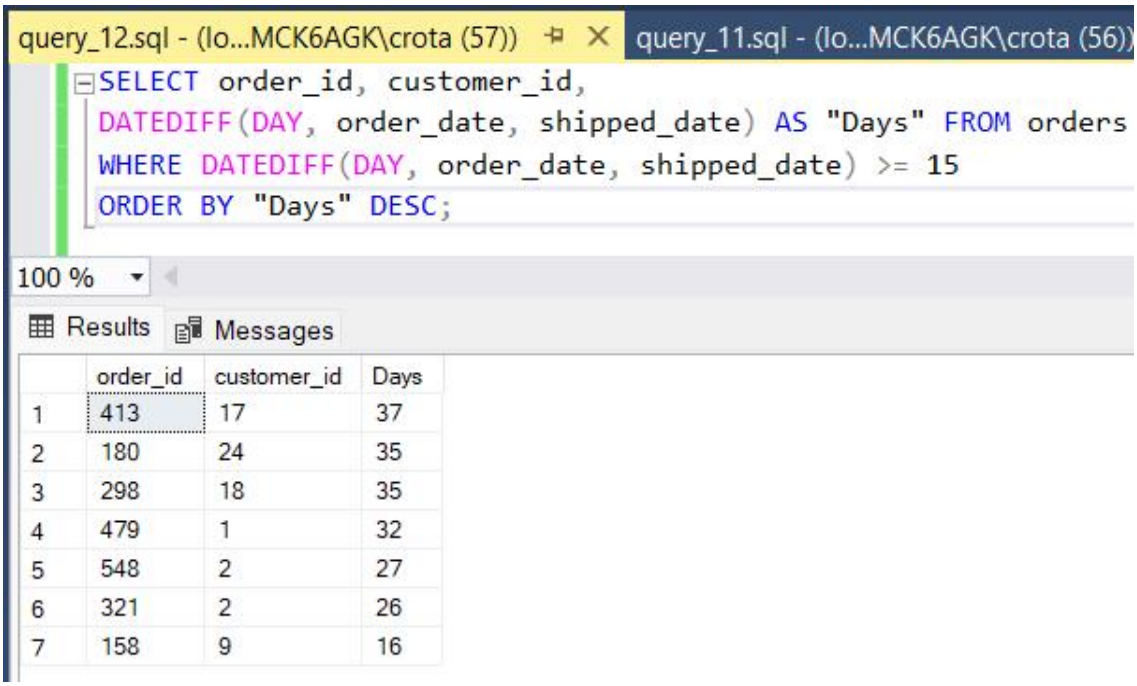



| | order_id | Ordered |
|---|---|---|
| 1 | 624 | 05/04/14 |
| 2 | 627 | 05/05/14 |
| 3 | 630 | 05/08/14 |
| 4 | 651 | 05/19/14 |
| 5 | 658 | 05/23/14 |

Explanation: Two columns are returned using data from the orders table, with the first being the identifying column for each order and the second being a string formatted date column of the order date corresponding to each order. The format function returns 2-digit places for the month and day, and using the right clause allows us to show only the last two digits of the year. Using the between clause constrains dates from the first day of May to the last day of May from 2014 and is the condition for an order to be returned.

**Query 12**: Write a query which displays the order id, customer id, and the number of days between the order date and the ship date (use the DATEDIFF function). Name this column “Days” and sort by highest to lowest number of days. Only display orders where this result is 15 days or more.

SELECT order_id, customer_id,
DATEDIFF(DAY, order_date, shipped_date) AS "Days" FROM orders
WHERE DATEDIFF(DAY, order_date, shipped_date) >= 15
ORDER BY "Days" DESC;

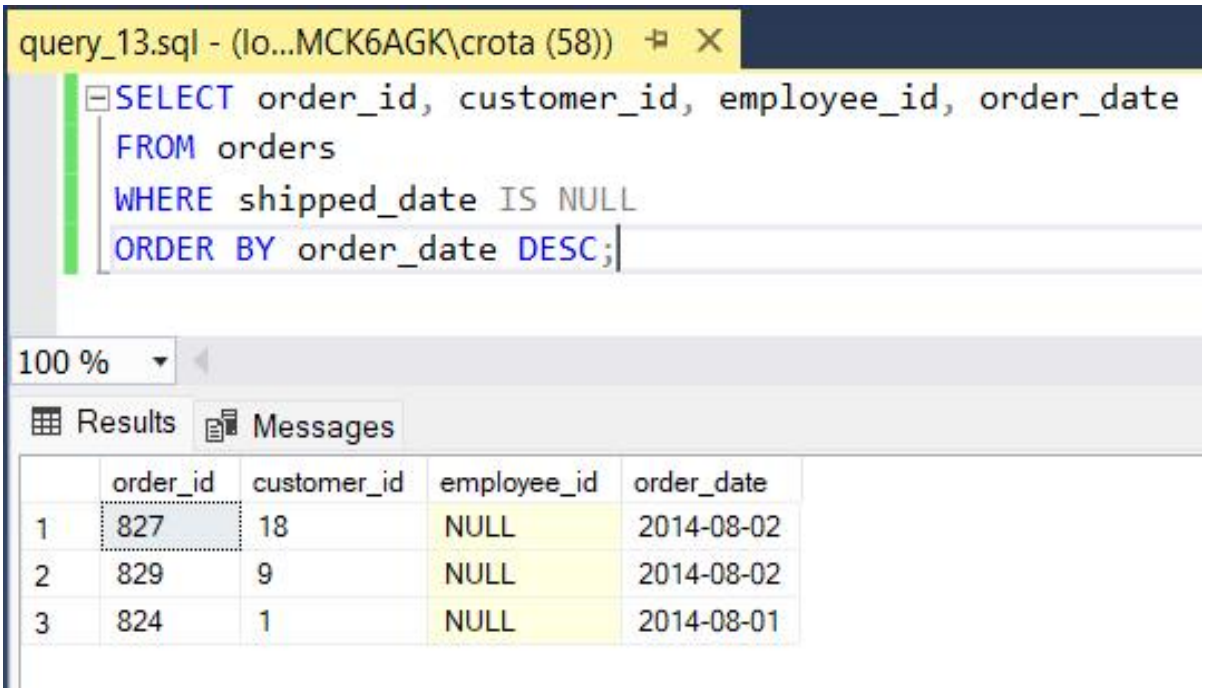



| | order_id | customer_id | Days |
|---|---|---|---|
| 1 | 413 | 17 | 37 |
| 2 | 180 | 24 | 35 |
| 3 | 298 | 18 | 35 |
| 4 | 479 | 1 | 32 |
| 5 | 548 | 2 | 27 |
| 6 | 321 | 2 | 26 |
| 7 | 158 | 9 | 16 |

Explanation: From the order tables we are interested in seeing the order and customer combination identified by the id fields and then a calculated field to see the lead time aka the time it took from the day an order was placed to the day it was then shipped. Using the datediff function allows us to show the number difference between two given dates after specifying the time unit (day, month, year, etc). The resulting rows are then sorted from most days to least days it took.

**Query 13**: Write a query which displays the order id, customer id, employee id, and order date for all orders that have NOT been shipped, sorted by order date with the most recent order at the top.

SELECT order_id, customer_id, employee_id, order_date
FROM orders WHERE shipped_date IS NULL ORDER BY order_date DESC;

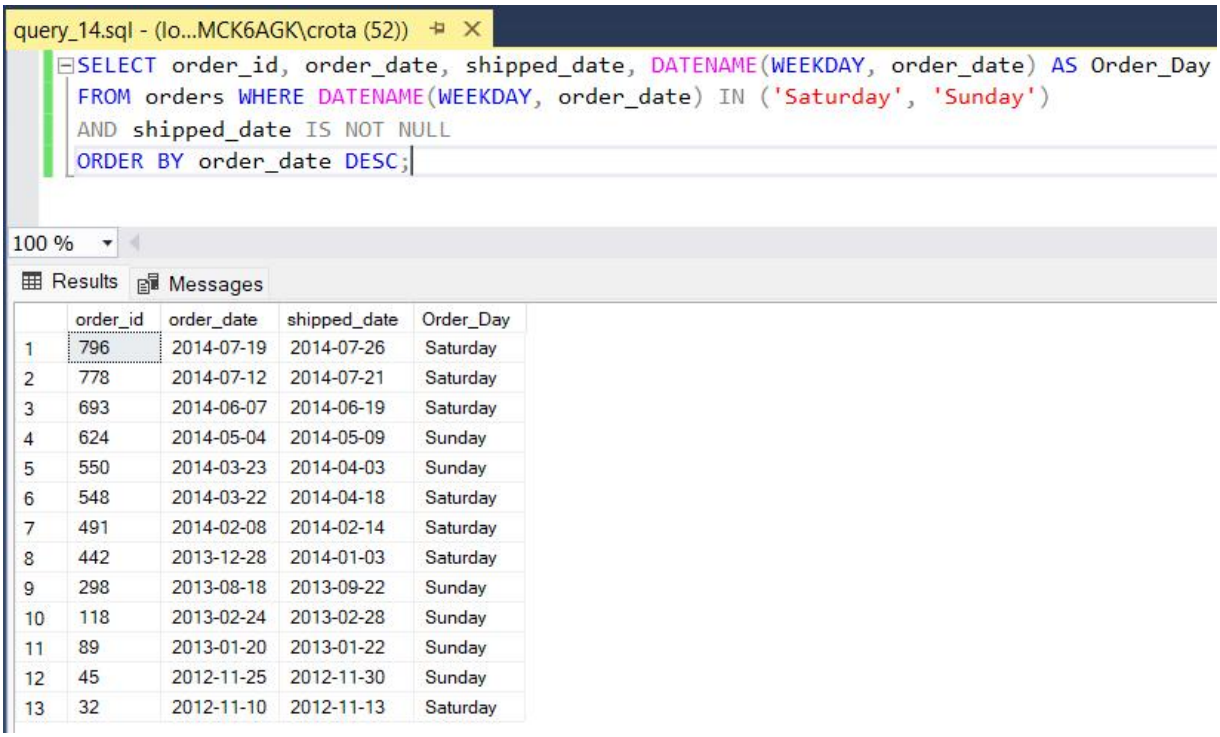



| | order_id | customer_id | employee_id | order_date |
|---|---|---|---|---|
| 1 | 827 | 18 | NULL | 2014-08-02 |
| 2 | 829 | 9 | NULL | 2014-08-02 |
| 3 | 824 | 1 | NULL | 2014-08-01 |

Explanation: This query is to return any orders that still need to be shipped. The column fields we are interested in is the order id to know which order specifically we need to know about, and also the customer the order belongs to and the employee which may be responsible for the order as well as the date the order was initially placed. All of these columns have been derived from the orders table. The condition for any orders/rows that are returned is when the value in shipping date does not exist aka null. Then we order the rows from most recent to later order date.

**Query 14**: The Marketing Department has requested a new report of shipped orders for which the order was placed on either a Saturday or a Sunday. Write a query which displays the order id, order date, shipped date, along with a calculated column labeled “Order_Day” showing the day of the week the order was placed (use the DAYNAME function). Only display orders that have shipped and were placed on a Saturday or Sunday. Sort by order date with most recent orders at the top.

SELECT order_id, order_date, shipped_date, DATENAME(WEEKDAY, order_date) AS Order_Day FROM orders WHERE DATENAME(WEEKDAY, order_date) IN ('Saturday', 'Sunday') AND shipped_date IS NOT NULL ORDER BY order_date DESC;

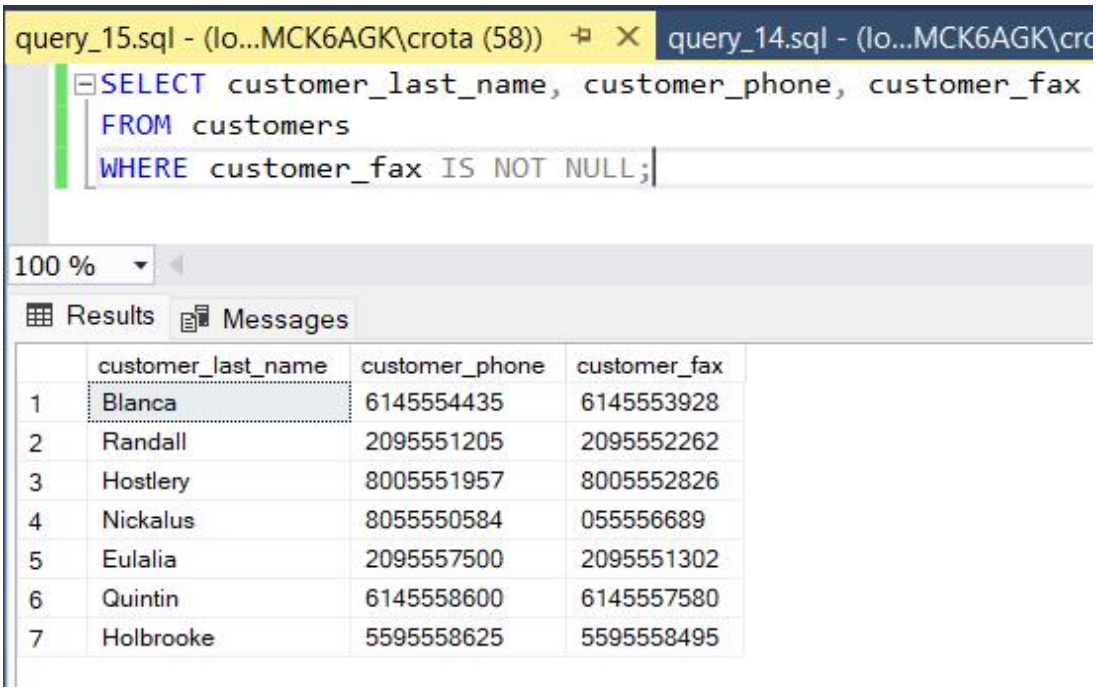



| | order_id | order_date | shipped_date | Order_Day |
|---|---|---|---|---|
| 1 | 796 | 2014-07-19 | 2014-07-26 | Saturday |
| 2 | 778 | 2014-07-12 | 2014-07-21 | Saturday |
| 3 | 693 | 2014-06-07 | 2014-06-19 | Saturday |
| 4 | 624 | 2014-05-04 | 2014-05-09 | Sunday |
| 5 | 550 | 2014-03-23 | 2014-04-03 | Sunday |
| 6 | 548 | 2014-03-22 | 2014-04-18 | Saturday |
| 7 | 491 | 2014-02-08 | 2014-02-14 | Saturday |
| 8 | 442 | 2013-12-28 | 2014-01-03 | Saturday |
| 9 | 298 | 2013-08-18 | 2013-09-22 | Sunday |
| 10 | 118 | 2013-02-24 | 2013-02-28 | Sunday |
| 11 | 89 | 2013-01-20 | 2013-01-22 | Sunday |
| 12 | 45 | 2012-11-25 | 2012-11-30 | Sunday |
| 13 | 32 | 2012-11-10 | 2012-11-13 | Saturday |

Explanation: Selecting the identifying column for orders, the date of orders, the date an order was shipped, and the day of the week the order was placed as the fields of interest, all derived or calculated from the table of orders. To get the name of the day, the date name function is used which takes a time unit and then the desired date as inputs. Only orders that have been shipped will be returned in the query results and sorted by most recent to later order dates.

**Query 15**: Write a query to display the customer's last name, phone number, and fax number but only display those customers that have a fax number.

```
SELECT customer_last_name, customer_phone, customer_fax
FROM customers
WHERE customer_fax IS NOT NULL;
```

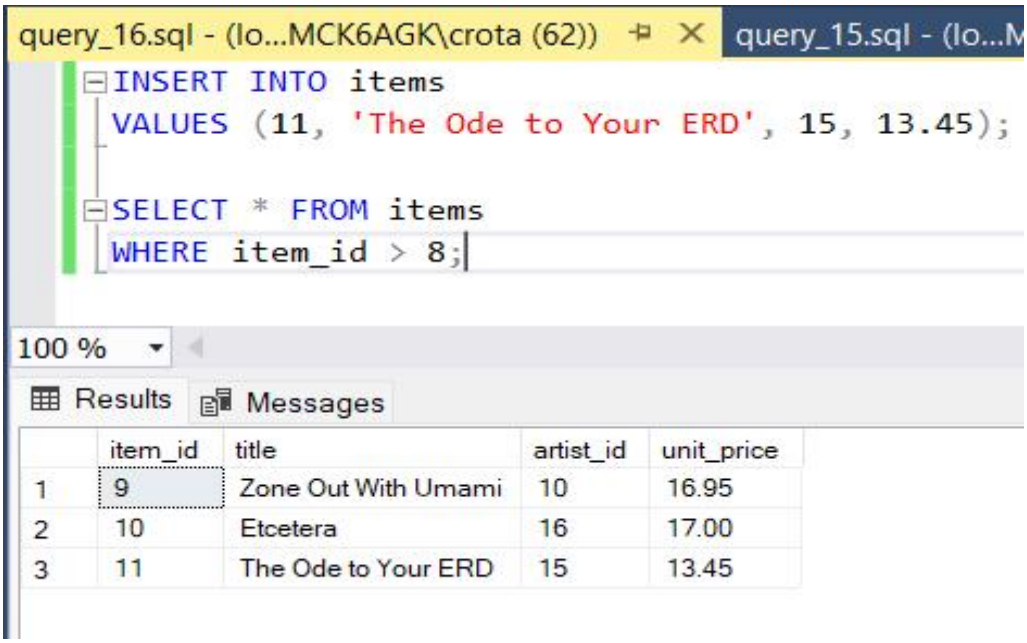



| | customer_last_name | customer_phone | customer_fax |
|---|---|---|---|
| 1 | Blanca | 6145554435 | 6145553928 |
| 2 | Randall | 2095551205 | 2095552262 |
| 3 | Hostlery | 8005551957 | 8005552826 |
| 4 | Nickalus | 8055550584 | 055556689 |
| 5 | Eulalia | 2095557500 | 2095551302 |
| 6 | Quintin | 6145558600 | 6145557580 |
| 7 | Holbrooke | 5595558625 | 5595558495 |

Explanation: The query displays the three column fields with the customer's last name, their phone number, and their fax number which is derived from the customers table. Because we want to see all customers who actually have a fax number, customers who do not have a fax number should not be shown/returned. By putting the condition where the customer fax field must not be empty (null) then we achieve this effect.

**Query 16**: Create a statement to insert a new record into the items table with the following values: item_id: 11, title: The Ode to Your ERD, Artist_id: 15, unit_price: 13.45

INSERT INTO items VALUES (11, 'The Ode to Your ERD', 15, 13.45); SELECT * FROM items WHERE item_id > 8;

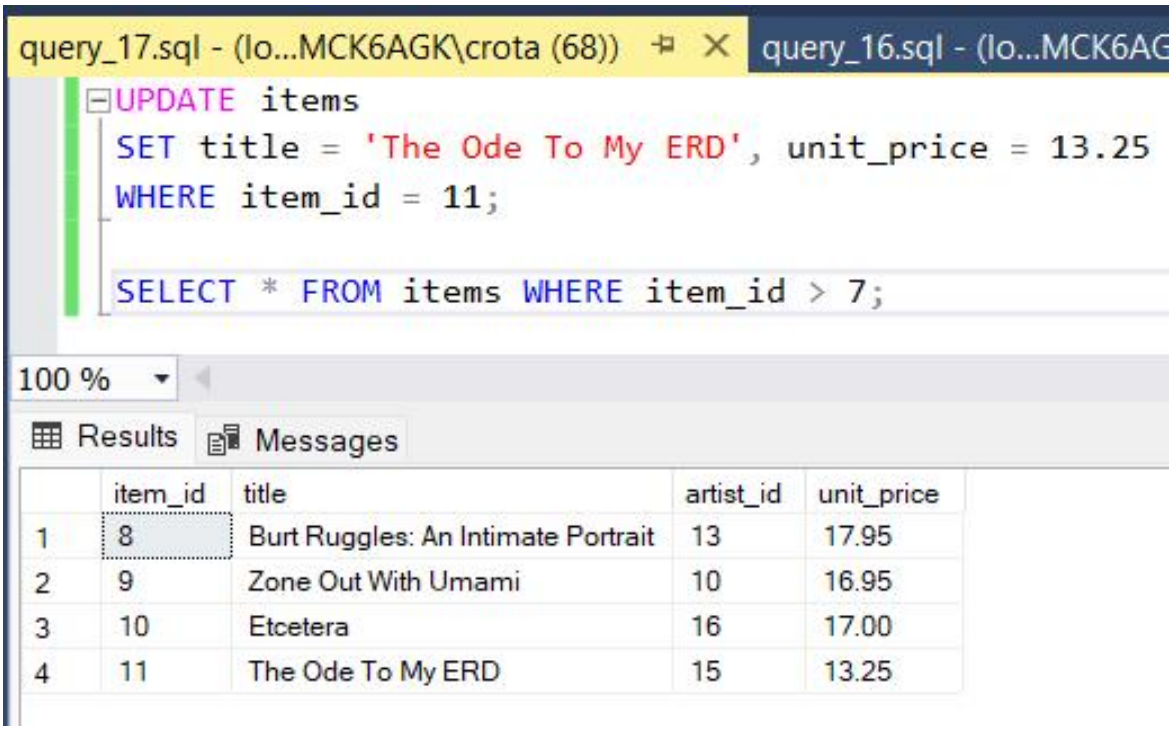



| | item_id | title | artist_id | unit_price |
|---|---|---|---|---|
| 1 | 9 | Zone Out With Umami | 10 | 16.95 |
| 2 | 10 | Etcetera | 16 | 17.00 |
| 3 | 11 | The Ode to Your ERD | 15 | 13.45 |

Explanation: Using the insert function we specify the table that the values should be inserted into which in this case is the items table. These new values are placed by order of the columns of the items table as seen in the parenthesis as the last row in the items table. The select statement is then used to verify that a new row had been created after item 10.

**Query 17**: Create a statement to update the record inserted in the previous step to change the unit price of this item to 13.25 with item_id: 11, title: The Ode To My ERD, artist: 15, and unit_price: 13.25.

UPDATE items SET title = 'The Ode To My ERD', unit_price = 13.25
WHERE item_id = 11; SELECT * FROM items WHERE item_id > 7;

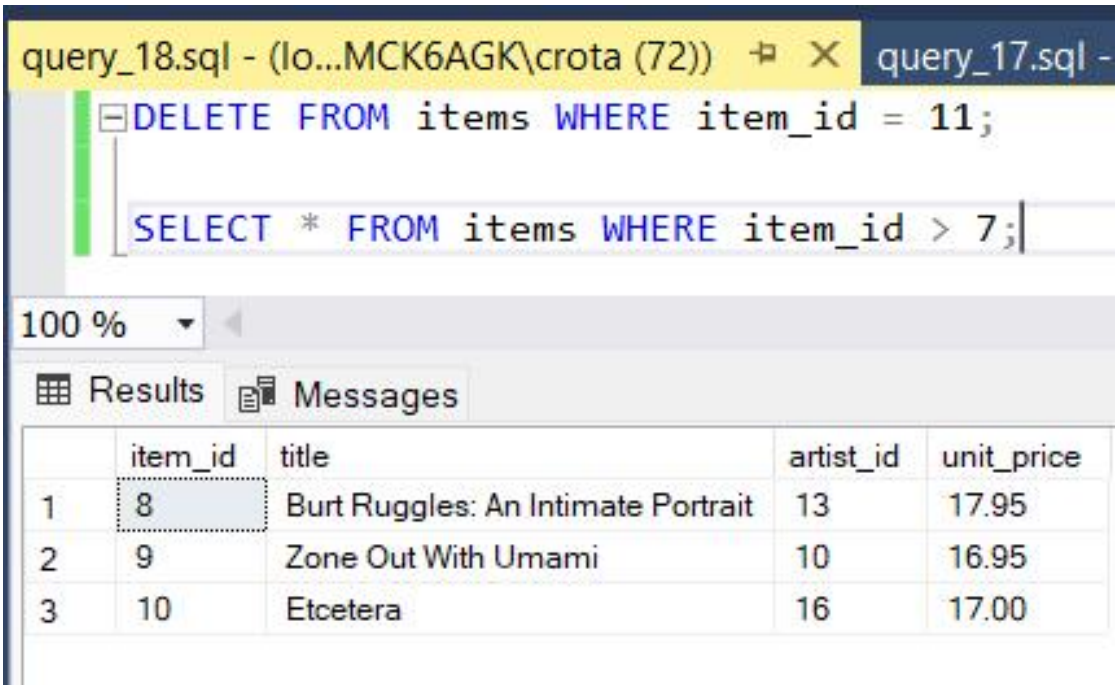



| | item_id | title | artist_id | unit_price |
|---|---|---|---|---|
| 1 | 8 | Burt Ruggles: An Intimate Portrait | 13 | 17.95 |
| 2 | 9 | Zone Out With Umami | 10 | 16.95 |
| 3 | 10 | Etcetera | 16 | 17.00 |
| 4 | 11 | The Ode To My ERD | 15 | 13.25 |

Explanation: In query 16 the inserted row had “The Ode to Your ERD” as the title and “13.25” as the unit price. In this query the update clause is used to update the table items with the condition that the update will take place at item 11 and the new values of “The Ode to My ERD” is set to the title and the unit price of 13.25. The select clause is used to verify that the row had been successfully updated.

**Query 18**: Create a statement to delete the entire record that was inserted and then updated in the previous steps. Show your DELETE statement along with the results of the following SELECT query to verify that the insert worked correctly.

DELETE FROM items WHERE item_id = 11;
SELECT * FROM items WHERE item_id > 7;

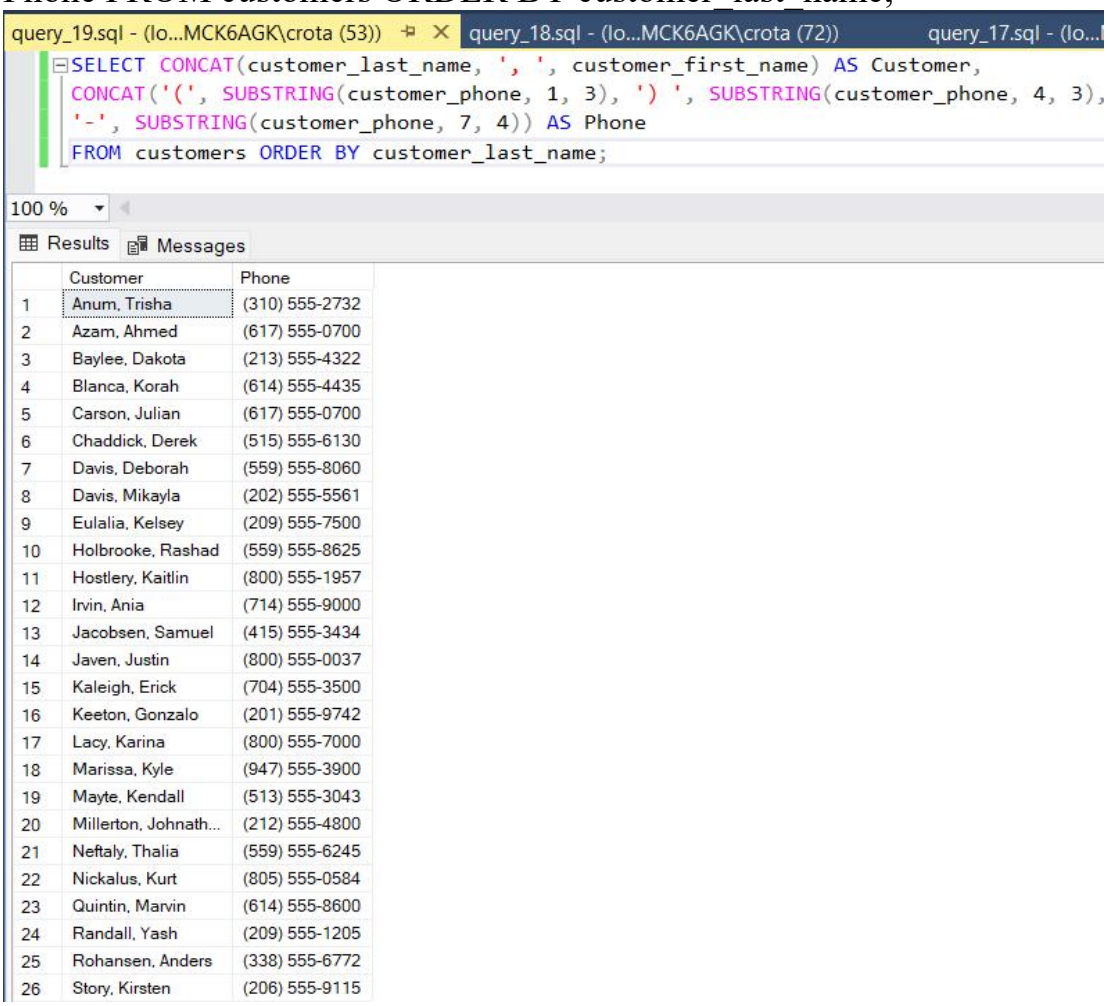



| | item_id | title | artist_id | unit_price |
|---|---|---|---|---|
| 1 | 8 | Burt Ruggles: An Intimate Portrait | 13 | 17.95 |
| 2 | 9 | Zone Out With Umami | 10 | 16.95 |
| 3 | 10 | Etcetera | 16 | 17.00 |

Explanation: Using the delete cause, the new row inserted in query 16 is now deleted when specified the table items from which the row is in and the condition that the row has the value 11 for its item id to signify we are removing item 11. Then the select statement is used to show all rows after item 7 to see that the row with item 11 has been removed.

**Query 19**: Using the SUBSTRING and CONCAT functions, write a query to display each customer name as a single field in the format “Jones, Tom” with a heading of Customer along with the customer_phone field in a nicely formatted calculated column named Phone. For example, a record containing the customer_phone value 9095595443 would be output with parentheses, spaces, and hyphens, like this: (909) 559-5443. Sort by last name.

SELECT CONCAT(customer_last_name, ', ', customer_first_name) AS Customer, CONCAT('(', SUBSTRING(customer_phone, 1, 3), ') ', SUBSTRING(customer_phone, 4, 3), '-', SUBSTRING(customer_phone, 7, 4)) AS Phone FROM customers ORDER BY customer_last_name;

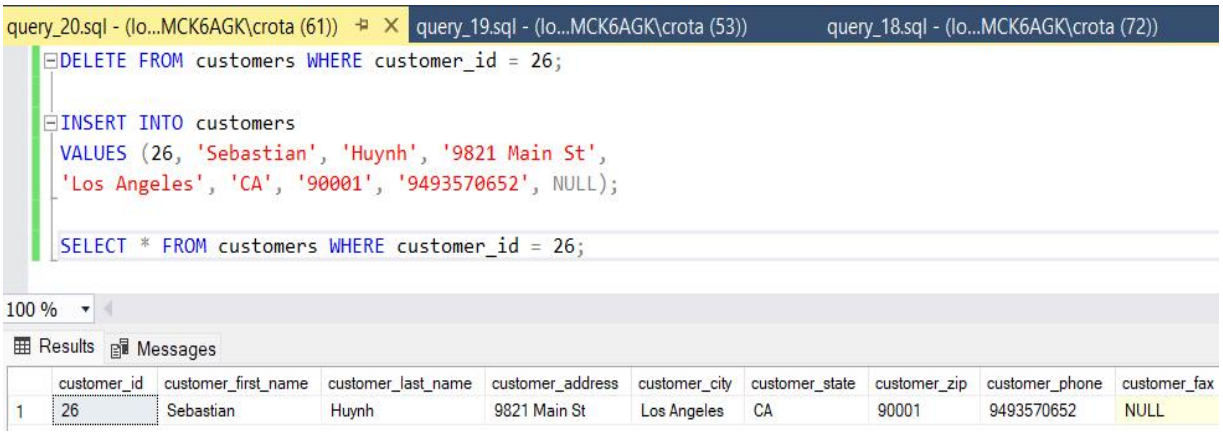



| | Customer | Phone |
|---|---|---|
| 1 | Anum, Trisha | (310) 555-2732 |
| 2 | Azam, Ahmed | (617) 555-0700 |
| 3 | Baylee, Dakota | (213) 555-4322 |
| 4 | Blanca, Korah | (614) 555-4435 |
| 5 | Carson, Julian | (617) 555-0700 |
| 6 | Chaddick, Derek | (515) 555-6130 |
| 7 | Davis, Deborah | (559) 555-8060 |
| 8 | Davis, Mikayla | (202) 555-5561 |
| 9 | Eulalia, Kelsey | (209) 555-7500 |
| 10 | Holbrooke, Rashad | (559) 555-8625 |
| 11 | Hostlery, Kaitlin | (800) 555-1957 |
| 12 | Irvin, Ania | (714) 555-9000 |
| 13 | Jacobsen, Samuel | (415) 555-3434 |
| 14 | Javen, Justin | (800) 555-0037 |
| 15 | Kaleigh, Erick | (704) 555-3500 |
| 16 | Keeton, Gonzalo | (201) 555-9742 |
| 17 | Lacy, Karina | (800) 555-7000 |
| 18 | Marissa, Kyle | (947) 555-3900 |
| 19 | Mayte, Kendall | (513) 555-3043 |
| 20 | Millerton, Johnath... | (212) 555-4800 |
| 21 | Neftaly, Thalia | (559) 555-6245 |
| 22 | Nickalus, Kurt | (805) 555-0584 |
| 23 | Quintin, Marvin | (614) 555-8600 |
| 24 | Randall, Yash | (209) 555-1205 |
| 25 | Rohansen, Anders | (338) 555-6772 |
| 26 | Story, Kirsten | (206) 555-9115 |

Explanation: Derived from customers table. Combining two columns with the format of “last name, first name” for each customer and the alias given to this field is “Customer”. The other column is a formatted string converting raw phone numbers into the format seen with parenthesis and semicolon. This is done with the concatenating different portions of the phone number using substring and adding the appropriate characters together. The results are then sorted by last name.

**Query 20**: Create a statement to insert a new record with your values: your customer id, first name, last name, address, city, state, zip code and fax number. Use your real name and (9821 Main St, Los Angeles, CA 90001) as your address: Show your INSERT statement along with the results of the following SELECT query to verify that the insert worked correctly.

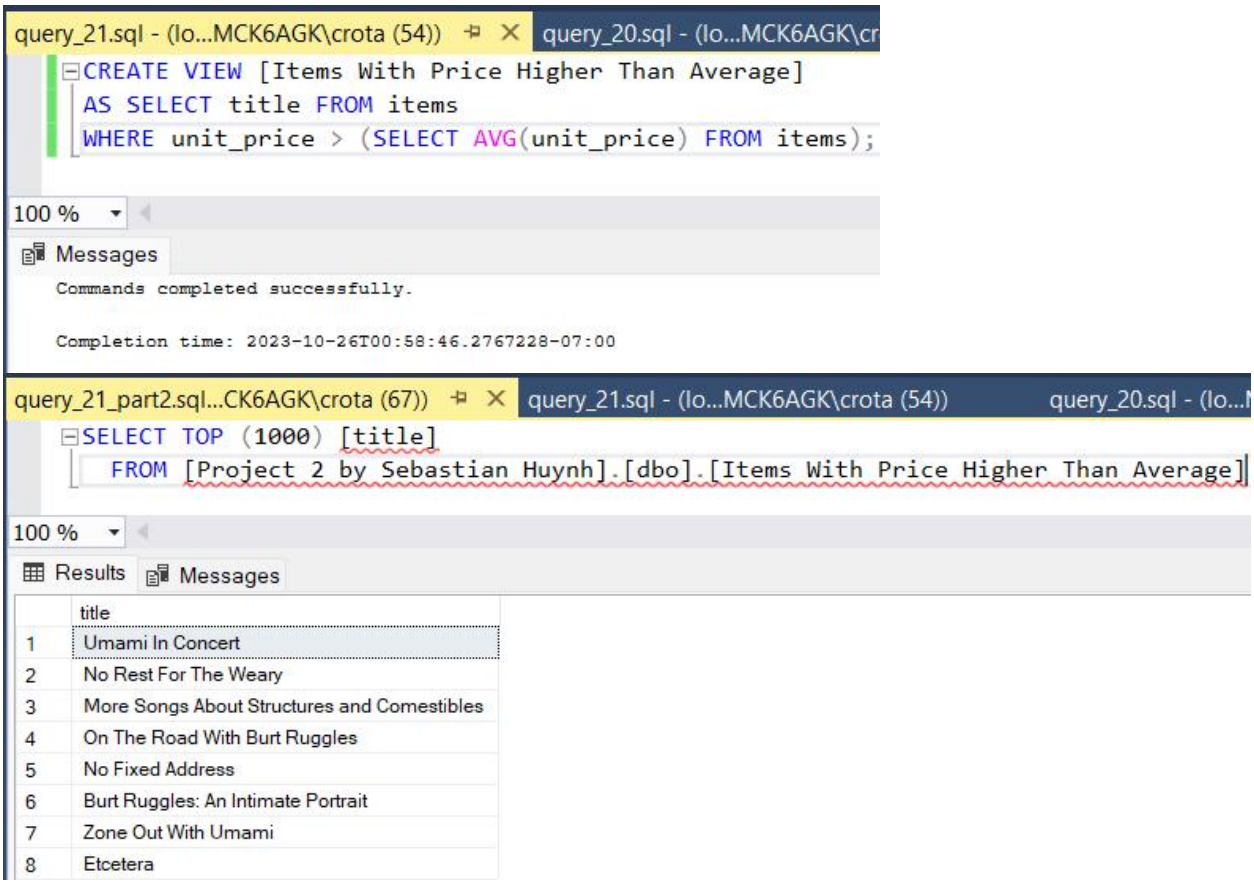
DELETE FROM customers WHERE customer_id = 26;
INSERT INTO customers VALUES (26, 'Sebastian', 'Huynh', '9821 Main St', 'Los Angeles', 'CA', '90001', '9493570652', NULL);
SELECT * FROM customers WHERE customer_id = 26;




Explanation: Adding my own information as if I was a customer. Customer 26 already exists so in order to add my information without throwing an error, the delete cause for the table of customers is used. Then the insert clause is used with my values placed in parenthesis in the order of the columns of the customer table. The select clause is used to display that the row has been updated.

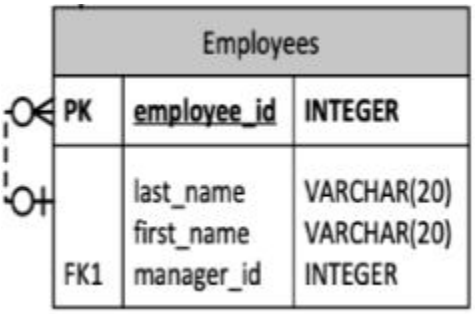
**Query 21**: Creates a view that selects every title in the "item" table with a price higher than the average price.

CREATE VIEW [Items With Price Higher Than Average] AS SELECT title FROM items WHERE unit_price > (SELECT AVG(unit_price) FROM items);
SELECT TOP (1000) [title] FROM [Project 2 by Sebastian Huynh].[dbo].[Items With Price Higher Than Average]




Explanation: Created a view (specialized table object) that contains one column field from the items table which are the item titles. “Items with a price higher than average” is the name for this view. The avg function is used on the unit prices to be the condition for the query and unit prices must be larger than it to be returned. Finally, the last select clause was autogenerated by management studio when right clicking ‘select top 1000 rows’ for the view in the SSMS object explorer. This is a quick way to see the first 1000 rows of a table or view to double check work.

**Query 22**: Explain the cardinality from the employees-to-employees table.




- Explanation: The cardinality of the employees table is a non-identifying unary relationship. It is non-identifying because the primary key is not a composite key made up of the foreign key and primary key. It is unary because managers are also employees with a different responsibility. According to the relationship depicted above, a manager may have zero or many employees under him/her but an employee may only belong to zero or one manager. A manager's primary key will be his/her employee id but it will simultaneously appear as the foreign key (manager id) for the employee(s) under him/her.

**Query 23**: Explain the following statement.

```
SELECT pv.ProductID, v.BusinessEntityID, v.Name
FROM Purchasing.ProductVendor AS pv
INNER JOIN Purchasing.Vendor AS v
   ON (pv.BusinessEntityID = v.BusinessEntityID)
WHERE StandardPrice > $10
   AND Name LIKE N'F%';
```

- Explanation: This query statement inner/equijoins two tables. We first want this combined table to have the column fields of product id from the ProductVendor table and then the business entity id and name from the Vendor table. Both tables are part of the Purchasing schema of the database and are given alias names with ProductVendor given the name pv and Vendor given the name v. The tables are joined on the common column of business entity id and will only return matching rows that fit the description of having a standard price greater than ten dollars and any length name that starts with letter F.

**Query 24**: Explain the following statement.

```
SELECT pv.ProductID, v.BusinessEntityID, v.Name
FROM Purchasing.ProductVendor AS pv, Purchasing.Vendor AS v
WHERE pv.BusinessEntityID=v.BusinessEntityID
        AND StandardPrice > $10
        AND Name LIKE N'F%';
```

- Explanation: This query statement inner/equijoins two tables by use of the where clause. This combined table has the column fields of product id from the ProductVendor table and then the business entity id and name from the Vendor table. Both tables are part of the Purchasing schema of the database and are given alias names with ProductVendor given the name pv and Vendor given the name v. The tables are joined on the common column of business entity id and will only return matching rows that fit the description of having a standard price greater than ten dollars and any length name that starts with the letter F.

**Query 25**: Compare query 23 and query 24.

- Explanation: Both queries output the same result but the order of execution is different for them. Join clauses usually execute before any of the where clauses so the join would combine the two tables first in query 23 then filter them with where. But in query 24 the join occurs when the where clause filters out all the rows derived using the from clause to match conditions.

## Results & Discussion

By practicing along with the project guidelines, SQL is a powerful language that makes relational databases useful to businesses. This is seen with how helpful queries can be, in the sense that they can show us what a business wants to see quickly rather than sifting through numerous rows and columns manually. This "narrowing down" effect for finding specific data lessens the chance for human error to be made and also highlights any useful connections/relationships in data that may not be easily seen with everything presented together. A drawback to writing SQL is how rigid the rules can be, meaning things that may make sense to you will not make sense to the program or database if syntax is missing or placed incorrectly or disobeys a SQL rule. This happened to me on various occasions during the completion of the project. For example, when trying to reference an expression using its alias name in the where clause throws an error, so I had to repeat the entire expression again for the query to run. Another example is when I attempted to use the like clause and accidently put "is like" which makes sense from the English way of structuring a sentence, but SQL doesn't accept it.

## Lessons Learned

1. Just because the SQL code runs does not mean that the results returned are what I intended it to show.
    - In query #1, initially I did not put ASC/ascending for the second column in the "order by" clause and it did not sort the second column even though it sorted by the first column. So it is important to always check if data corresponds to intention.
2. Formatting is important, especially if a company establishes how data should be presented or formatting makes data realistic.
    - When calculating the discounted price in query #7, having many decimal places in the answer is not useful because cents only go up to two decimal places. Also extra zeroes at the end are not helpful and can confuse those looking at the data which is why the convert to double precision formula was used.
3. There are many possible ways to get the same result.
    - Like observed in query 25
    - Concatenating strings can be done using "concat" or by simply using the "+" sign.

## Conclusion

Since the 1970s, SQL has proven to be one of the best ways to manage relational databases where collaboration between users, programs, and databases is made possible. However, there is also NoSql which stands for “not only sql” and it has been gaining a lot more popularity in recent years. With the rise of many unconventional types of data and internet-based applications, “NoSql databases thrive in environments with highly dynamic data and scale much better than SQL databases” (Powell, 2021).

Cited References

1. Hoffer, Jeff, et al. Modern Database Management. Available from: VitalSource Bookshelf, (13th Edition). Pearson Education (US), 2018.
2. Panigrahi, Kiran Kumar. “Difference between DDL and DML in DBMS.” Online Tutorials, Courses, and eBooks Library, 12 Sept. 2023, www.tutorialspoint.com/difference-between-ddl-and-dml-in-dbms.
3. Powell, Ron. “SQL VS NoSQL Databases.” CircleCI, CircleCI, 10 Sept. 2021, circleci.com/blog/sql-vs-nosql-databases/.