**AIRIOHUODION CLEMENT**

**APRIL 24, 2024**

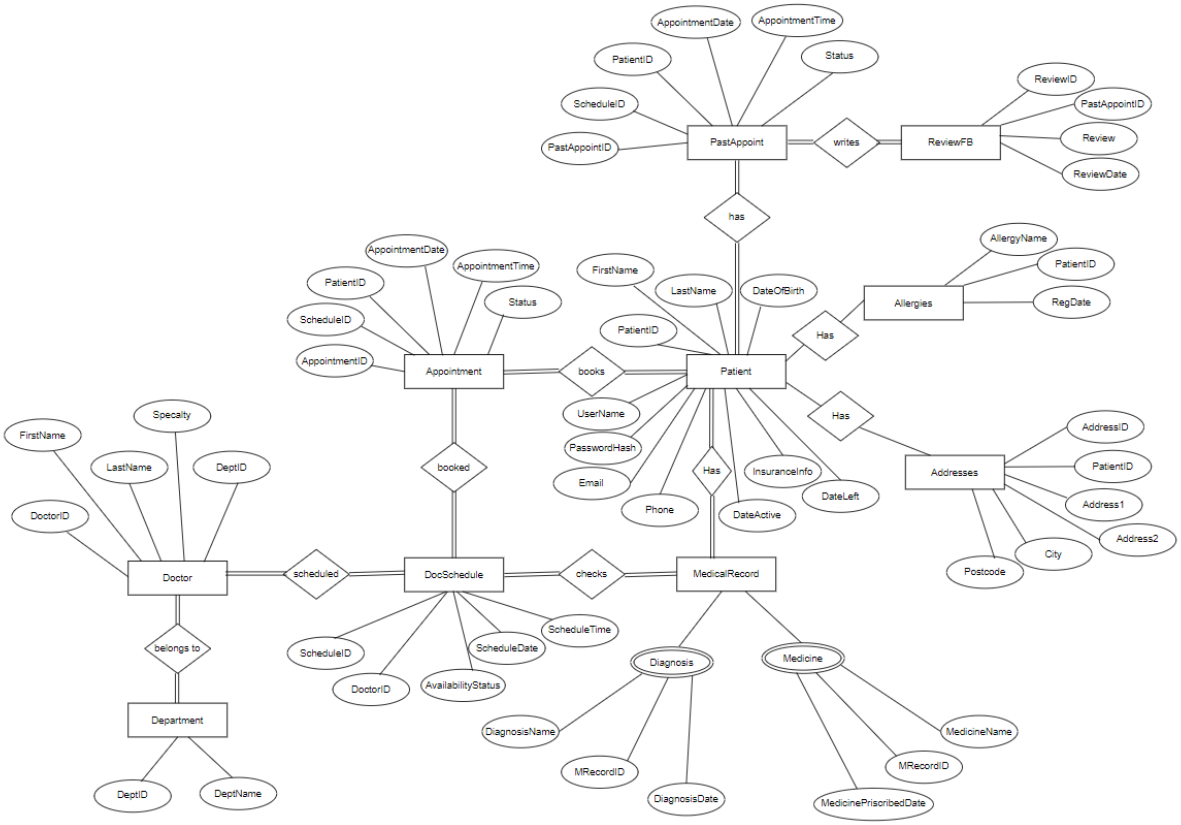
## INTRODUCTION

As a database developer consultant for a hospital, I'm to create a complete database system that satisfies the needs of the customer. To effectively store and handle data on patients, medicaldoctor, medical records, appointments and departments, the hospital is creating a new database system. Particular client requirements were obtained at the first meeting in order to make sure that the database system meets the demands of both patients and healthcare practitioners. By making patient registration, appointment scheduling, medical record administration, and feedback gathering easier, this database system will expedite healthcare operations and enhance patient care.

In this task the first step I've decided to take is to prepare an ER proposed database design into 3NF, with the Entities, attributes and the relationships between them, taking into consideration the principles of database design which were thought in lecture. Below is the sketch of my proposed database tables and attributes.

**PART ONE:**

## Designing and normalizing my proposed database into 3NF




ER DESIGN

**Figure 1.1.1:**

Based on the request by the Hospital my design has been structured to support the hospital management system. To further explain the purpose of my normalization let me start by explaining the purpose of the entities suggested.

1. **Patient:** It keeps records of patient information, such as contact details, insurance information, and active/inactive status.
2. **Doctor:** Contains information about physicians, such as their names, specializations, and departmental affiliations.

3. **Department:** Provides details on the various hospital department and each doctor employed to that department.
4. **Appointment**: Keeps track of patient appointment with physicians.
5. **PastAppointment:** Holds historical information about previous appointments, such as the patients details and status.
6. **DocSchedule:** This program keeps track of doctor's availability for appointments as well as their schedules.
7. **Addresses:** Keeps track of patient addresses
8. **MedicalRecord:** Contains medical records associated with patients, including diagnosis and prescribed medicines.
9. **Diagnosis:** Documents medical record related to diagnoses of patients from tests.
10. **Medicine:** Keeps track of Prescribed medications in relation to medical records.
11. **ReviewFB:** Holds examine patient comments related to previous meetings with doctors they have met.
12. **Allergies:** It provides medical Doctors with crucial information on patients' allergy, ensuring their safety and the appropriate treatment course.

CREATING DELIGHTHOSPITAL DATABASE

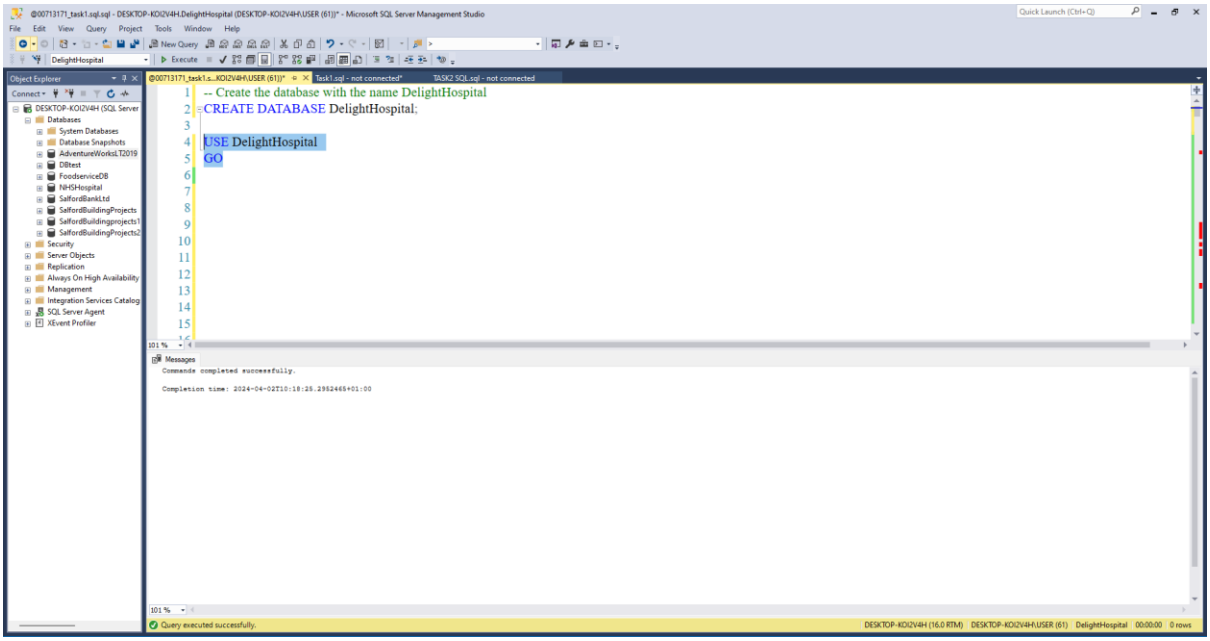



**Figure 1.1.2:** In this first step I have written a TSQL code to create a database with name **DelightHospital** and with the 'USE' query to change from the current database to the just created database.

## Normalization Process for Patient table:

Given that atomic values are listed in each column, the patient table looks to be in 1NF. I have made sure there are no partial dependencies in order to guarantee 2NF. Every attribute in the patient table is dependent on the patient ID primary key. I had to get rid of transitive dependencies for 3NF. The patient table represents each patient and contains record of their contact and personal data.

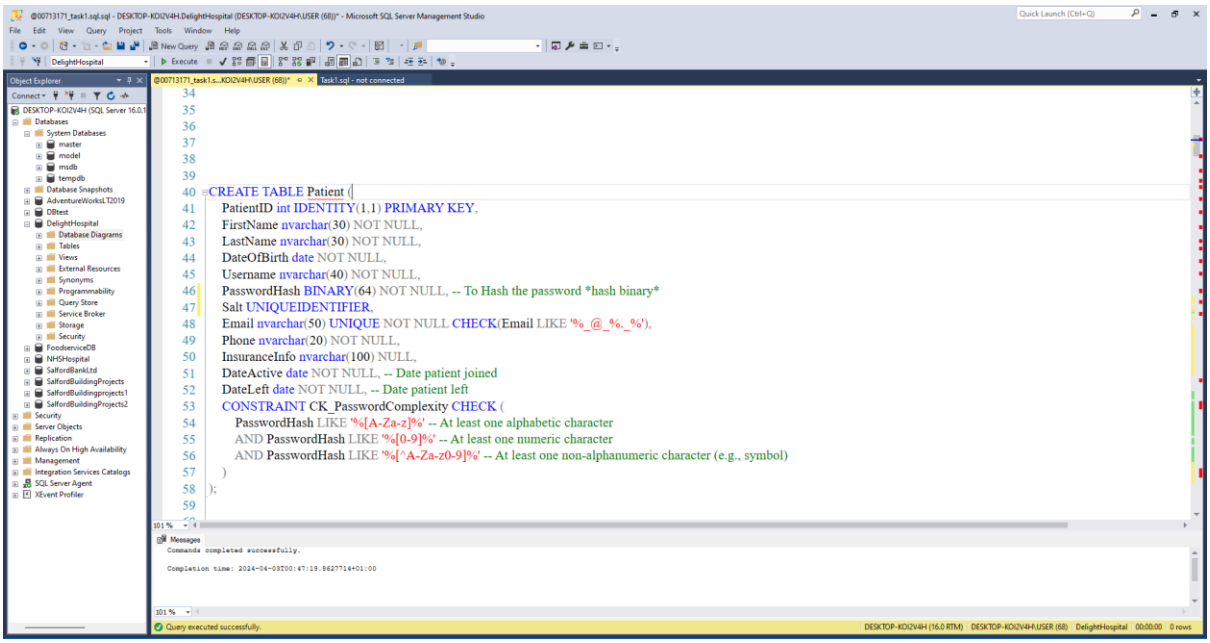



**Figure 1.1.3**: Justification

**PatientID** is the primary key for each patient record, auto-increasing. The **IDENTITY(1,1)** column uniquely identifies each patient. Other columns include FirstName, LastName, Username, Email, Phone, and Insuranceinfo, with string data stored in **nvarchar** and date values in DateOfBirth, DateActive, and DateLeft columns.

**PasswordHash** is stored as '**Binary(64)**' efficient storage of hashed password data. Unique salts are created using '**UNIQUEIDENTIFIER'**.

**Not null** constraints ensure required fields are filled in, while CHECK constraint ensures password complexity for security.

Unique constraints for emails ensure they contain the @ symbol and at least one character after the dot.

## Normalization Process for Allergies table:

The Allergies table is now in the 3NF, free of any partial or transitive dependencies, and designed to preserve data integrity thanks to this normalizing procedure.

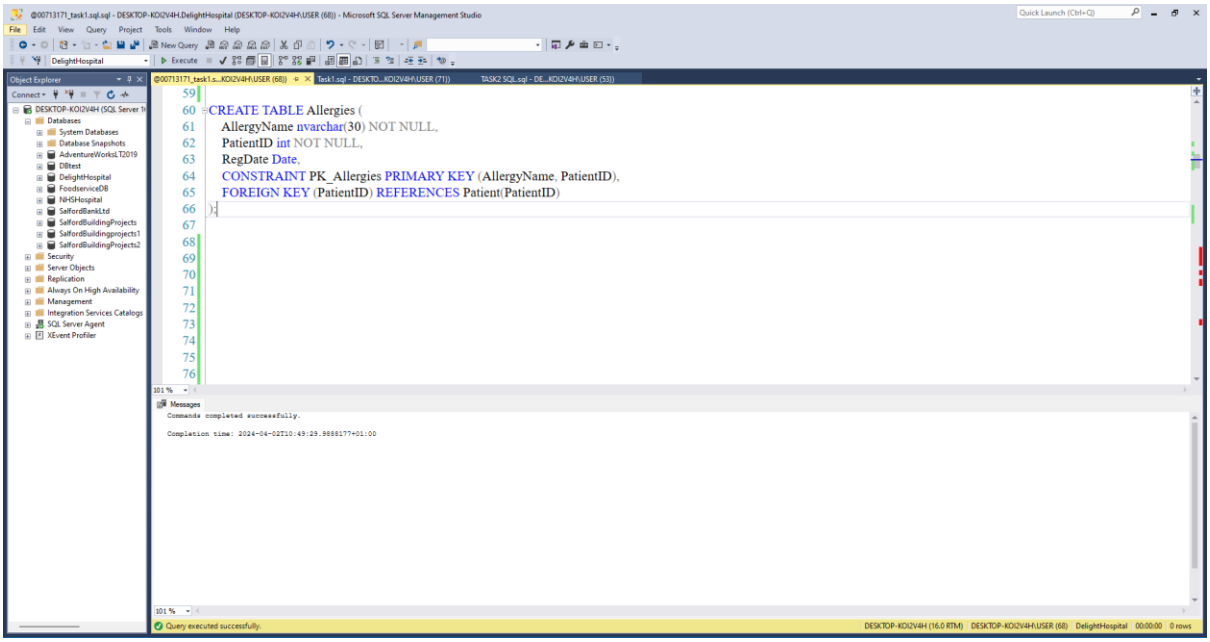



**Figure 1.1.4:** Justification

The table uses a **nvarchar** datatype for allergy names, allowing for a range of lengths. A **foreign key** is created for patients linked to allergies, and a column for Registration Date keeps dates for each allergy. The composite key (**AllergyName, PatientID**) ensures uniqueness in joint relations to MedicalRecord. The foreign key

(**PatientID**) ensures referential integrity between the Allergy and Patient tables, referencing the Patient column in the Patient table.

## Normalization Process for Department table:

There appears to be 1NF, 2NF, 3NF in the Department table. There are no transitive or partial dependencies, and it contains atomic values. The Department table facilitates departmental management and categorization by providing information about various hospital departments.

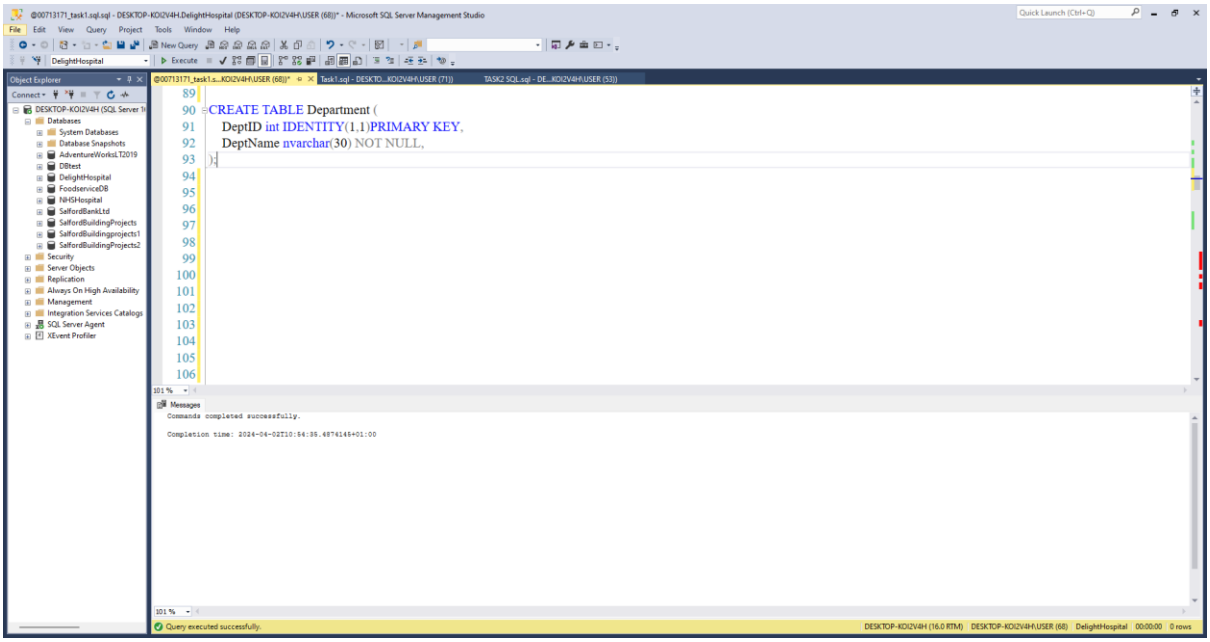



**Figure 1.1.5:** Justification

DeptID is the **primary key** for each department record, auto incremented in **integers**, it uniquely identifies each department using IDENTITY(1,1). DeptName, a string data type, is typically stored in **nvarchar**, accepting variable lengths and international characters. The DeptName table does not have **null** constraints.

## Normalization Process for Doctor table:

The Doctor table appears to be in 1NF, 2NF, and 3NF, just like the Patient table. Physicians' data, such as names, specializations, and department affiliations, are kept in the Doctor table. The hospital can effectively manage doctor related data thanks to this table.

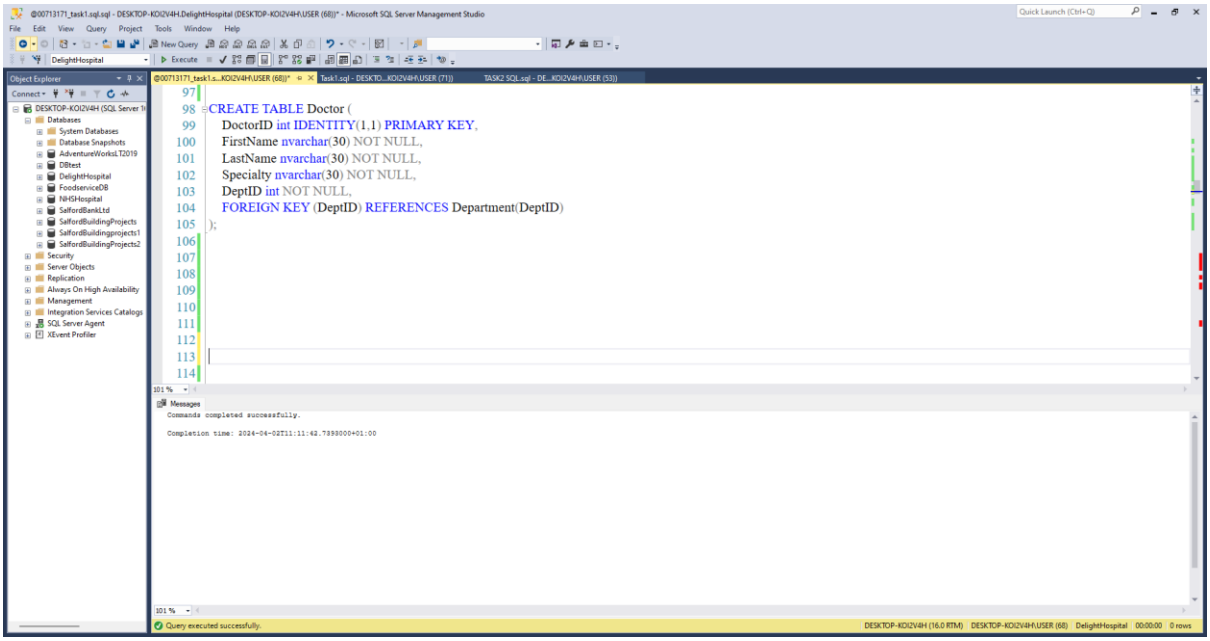



**Figure 1.1.6:** Justification

DoctorID is the primary key for each doctor record, auto incremented (int). it uniquely identifies each doctor using IDENTITY(1,1). FirstName, LastName, and Specialty are string data stored in nvarchar, with variable lengths and international characters. DeptID is the foreign key for the Department table, indicating the Department each Doctor is affiliated with. NOT NULL constraints ensure data integrity by providing necessary fields like FirstName, LastName, Specialty and Dept.

**Normalization Process for DocSchedule table:**

There are 3NF, 2NF, and 1NF in the DocSchedule table, no transitive relationships and every attribute is dependent on the primary key, ScheduleID. The DocSchedule table helps the hospital efficiently arrange physician schedules by managing doctors' calendars, including their availability for appointment

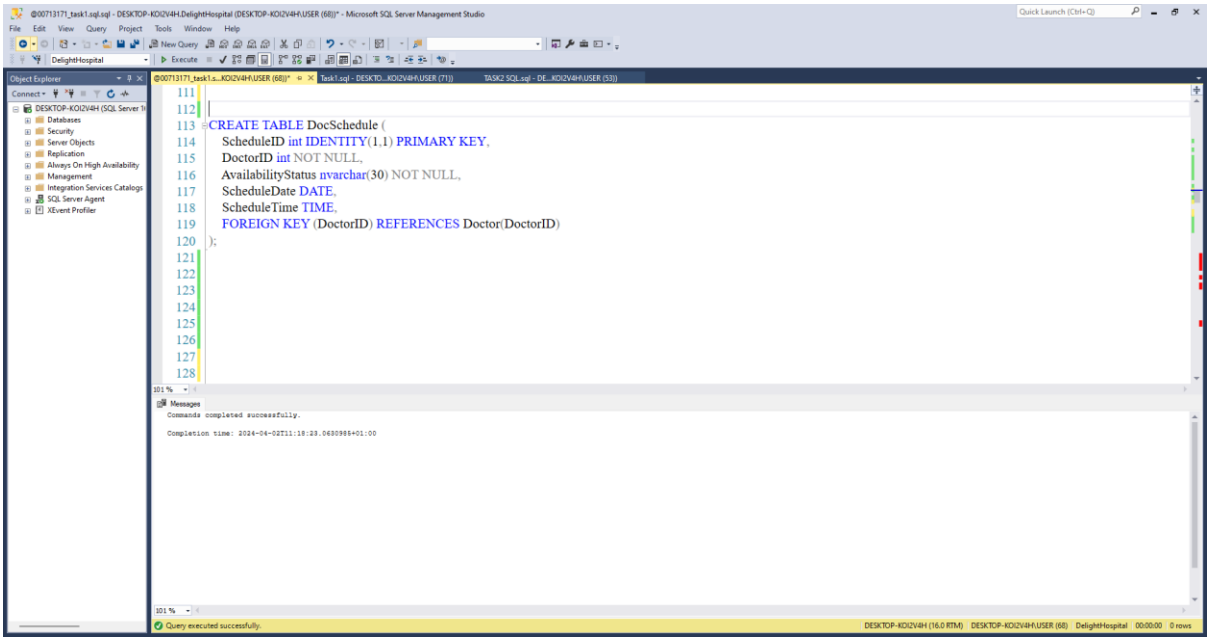



**Figure 1.1.7:** Justification

Since **ScheduleID** distinguishes every Doctor Schedule record individually, it is selected as the primary key, it is auto incremented (int). Every DocSchedule is identified specifically and uniquely in this column using IDENTITY(1,1).

**AvailabilityStatus** have string data and is typically stored in datatypes like nvarchar, which accept variable lengths and support international characters. While the **DoctorID** carries (int) datatype to take integer only

**ScheduleDate, ScheduleTime**, keeps the date and time record of the appointment.

Referential integrity between the DocSchedule and Doctor tables is ensured by the **FOREIGN KEY** constraint, which references the **DoctorID** column in the Doctor table.

**Not null** constraints in the DoctorID, AvailabilityStatus, tables guarantee that it is filled in at all times.

## Normalization Process for Appointment table:

There are no transitive relationships and every attribute is dependent on the primary key "AppointmentID" so the table is in 1NF, 2NF, and 3NF. Essential information including the appointment date, time, and status are stored in the Appointment table, which keeps track of patient-doctor appointments.

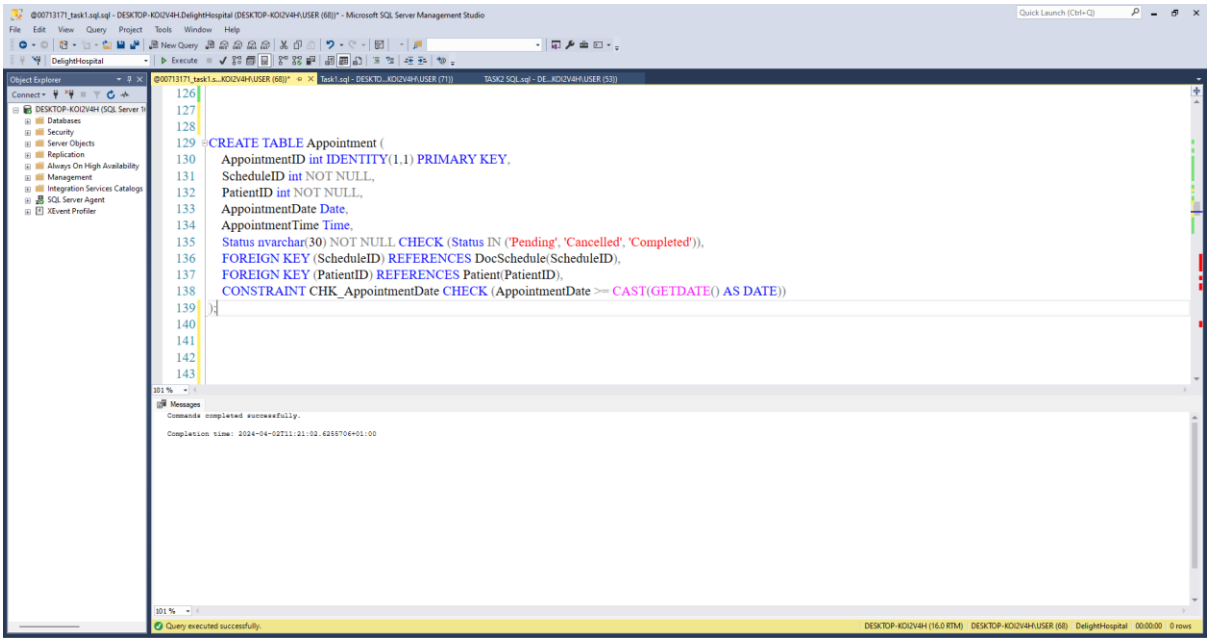



**Figure 1.1.8:** Justification

The **primary** key for Appointment table is AppointmentID, which is auto-incremented by IDENTITY(1,1), and uniquely identified for each record. The **ScheduleID** column is used to create a connection with the **DocSchedule** table

and represents the appointment's scheduled time slot. **PatientID** is used to identify the patient connected to their booked appointment. **AppointmentDate** records dates without time components, while **AppointmentTime** stores time separately from the date. The **status** column is set to nvarchar for descriptive representation. **Not null** constraints ensure that these columns are filled in at all times. **Foreign KEY** constraints ensure referral integrity between the Appointment table and related tables. The **CHECK** constraint guarantees the AppointmentDate is higher than or equal to the current date.

## Normalization Process for PastAppoint table:

There are no transitive relationships and every attribute is dependent on the primary key "PastAppointID" so the table is in 1NF, 2NF, and 3NF. Essential information including the **Past appointment** date, time, and **Completed** status are stored in the PastAppoint table, which keeps track of patient-doctor Past appointments.

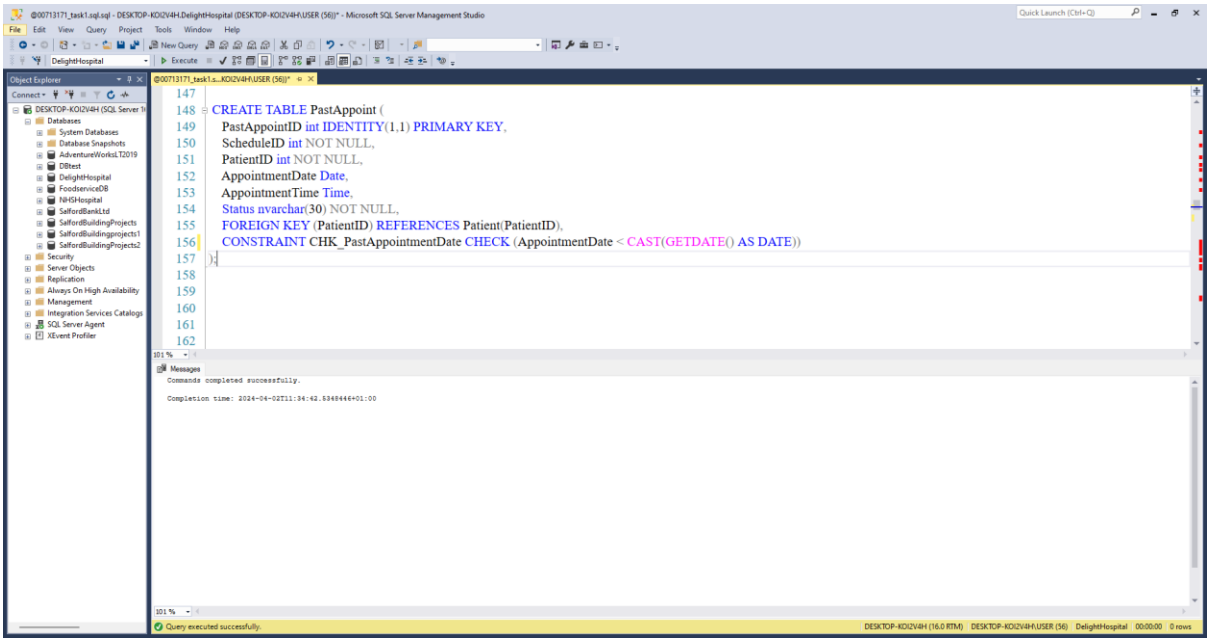



**Figure 1.1.9:** Justification

**PastAppointID** is the **primary** key for each Past Appointment record, auto-increasing to identify each record uniquely. **ScheduleID** is used to store DocSchedule IDs in relation to other records moved from the Appointment table. **Patient ID** is used as a **foreign key** to establish a relationship with the Patient table. **AppointmentDate** records dates without time components, while **AppointmentTime** stores time separately from date. The **status** column is set to **nvarchar** for descriptive representation. **Not null** constraints ensure that these tables are filled in at all times. **Foreign KEY** constraint ensures referral integrity between the **PastAppoint** table and related tables, referencing the PatientID column in the Patient table. The **CHECK** constraint guarantees that the Past AppointmentDate is past, less than the current date.

## Normalization Process for ReviewID table:

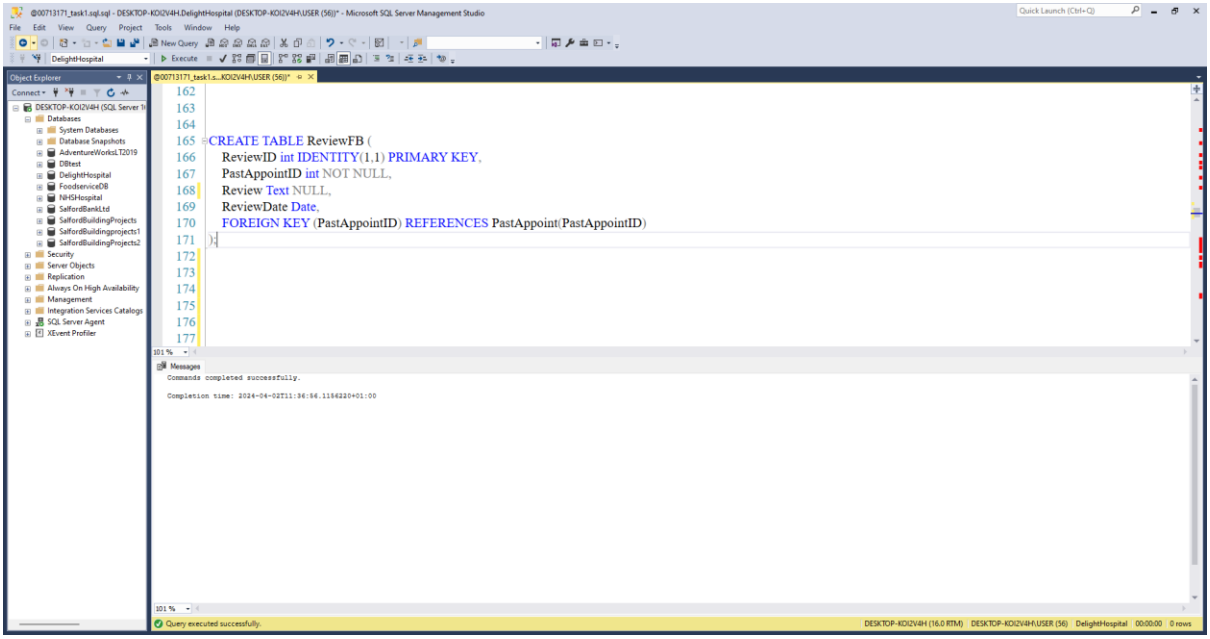



**Figure 1.2.0:** Justification

The table contains columns for review feedback input, previous appointment IDs, review text, review data, and references for the Foreign key(PastAppointID). The ReviewID column uniquely identifies each input and auto-increments using IDENTITY(1,1). The ReviewText column stores the textual substance of the comments, with "**TEXT**" being the data type. NULL values are allowed for non-review feedback situations. The ReviewDate column records the date of the review feedback. including ReviewID, PastAppointID, ReviewText, ReviewDate, and ReviewText, ReviewDate, and References for the Foreign key. Each input is uniquely identified using IDENTITY(1,1). The data type is TEXT, and NULL values are allowed for non-review feedback

This TSQL create a foreign key constraint, pastAppointID(PastAppoint), to ensure the PastAppointID in the ReviewFB table refers to a legitimate PastAppointID in the PastAppoint table.

## Normalization Process for MedicalRecord table:

There appears to be 1NF, 2NF, 3NF in the MedicalRecord table. There are no transitive or partial dependencies, and it contains atomic values. The MedicalRecord table houses medical information related to individual patient that have been uploaded after assessment.

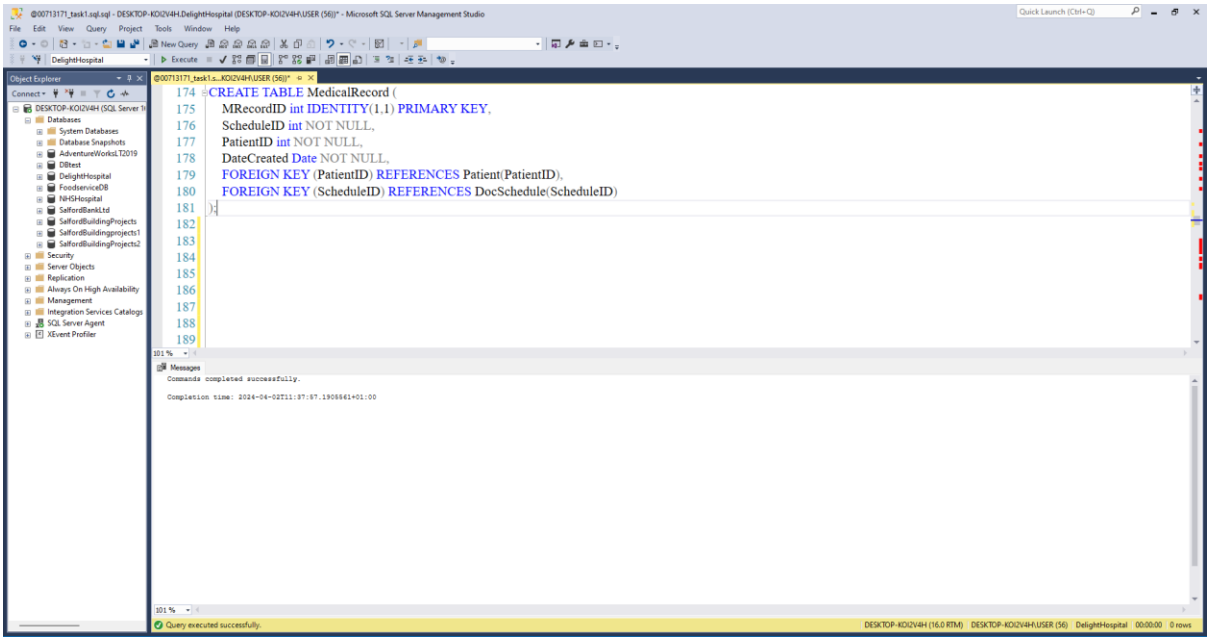



**Figure 1.2.1:** Justification

The MedicalRecord is uniquely identified by its MRecordID column, which is an auto-incremented integer. The schedule linked to the medical record, which is linked to the DocSchedule table. The PatientID field indicates the patient linked to the medical record, and the DateCreated field indicates the health record's creation date.

## Normalization Process for Diagnosis table:

There appears to be 1NF, 2NF, 3NF in the Diagnosis table. There are no transitive or partial dependencies, and it contains atomic values. The Diagnosis table houses Diagnosed illnesses information related to individual patient that have been uploaded after assessment and attached to the medical record.

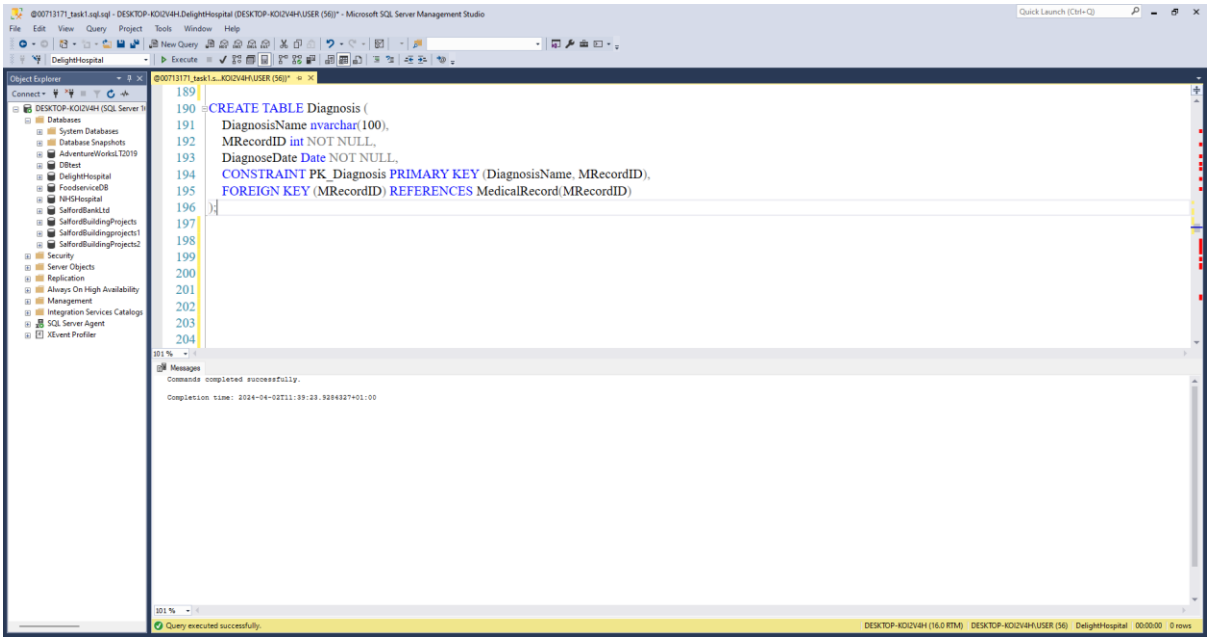



**Figure 1.2.2:** Justification

The DiagnosisName column stores the diagnosis name in nvarchar(100) format. **DiagnoseDate**: The data type is Date, with the primary key being **DiagnosisName** and **MRecordID**, ensuring uniqueness for every combination of diagnosis name and medical recordID. The **FOREIGN KEY**(MRecordID) ensures referential integrity between the Diagnosis table and the MedicalRecord table, referencing the MRecordID column.

## Normalization Process for Medicine table:

There appears to be 1NF, 2NF, 3NF in the Medicine table. There are no transitive or partial dependencies, and it contains atomic values. The Medicine table houses Medicine medications administered to patient but attached to the medical records.

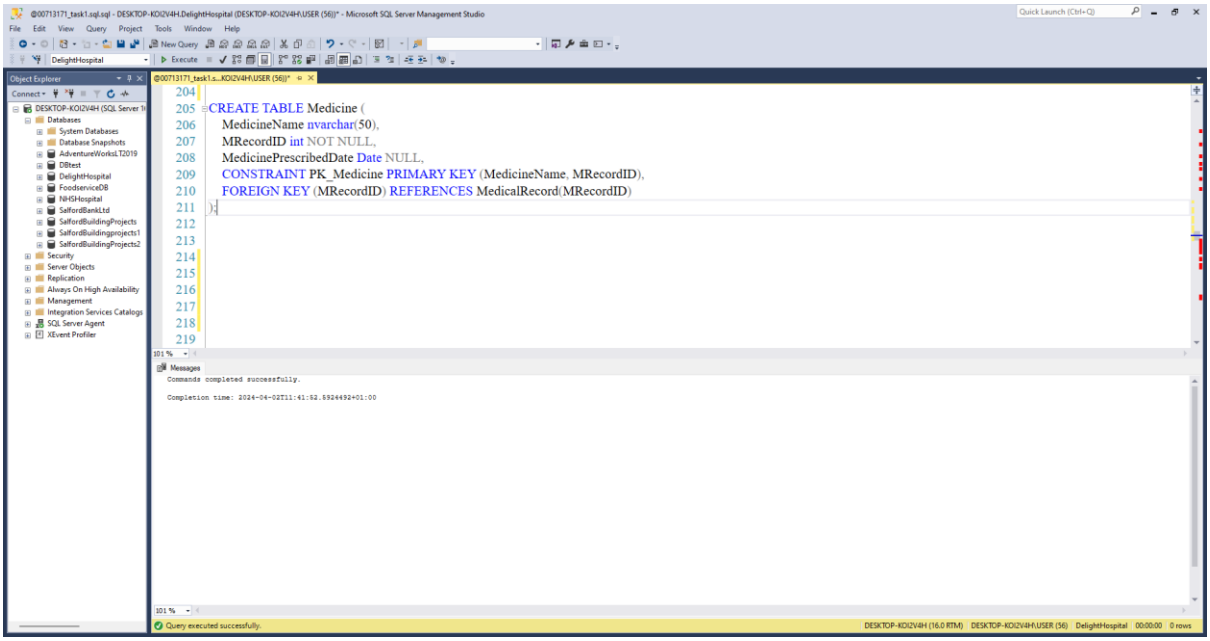



**Figure 1.2.3:** Justification

The table contains column for **Medication names**, **MRecordID**, and **Prescribed date**. The MedicineName column provides a variable length string using '**nvarchar**' for medication names, while the MRecordID column connects to the MedicalRecord table. The '**MedicinePrescribedDate**' column keeps dates that correspond to the medication's prescription date. The '**Composite**' primary key (MedicineName, MRecordID) ensures uniqueness, while the FOREIGN KEY (MRecordID) maintains referential integrity between the Medicine and MedicalRecord tables, finally **NOT NULL** is enforced on MRecordID and prescribed because they are very important data.

## Database Diagram

All tables contained in this database have been adequately joined with the required constraints such as foreign keys which has been used to reference or link together related tables so as to query accurate result. Below is the database diagram.

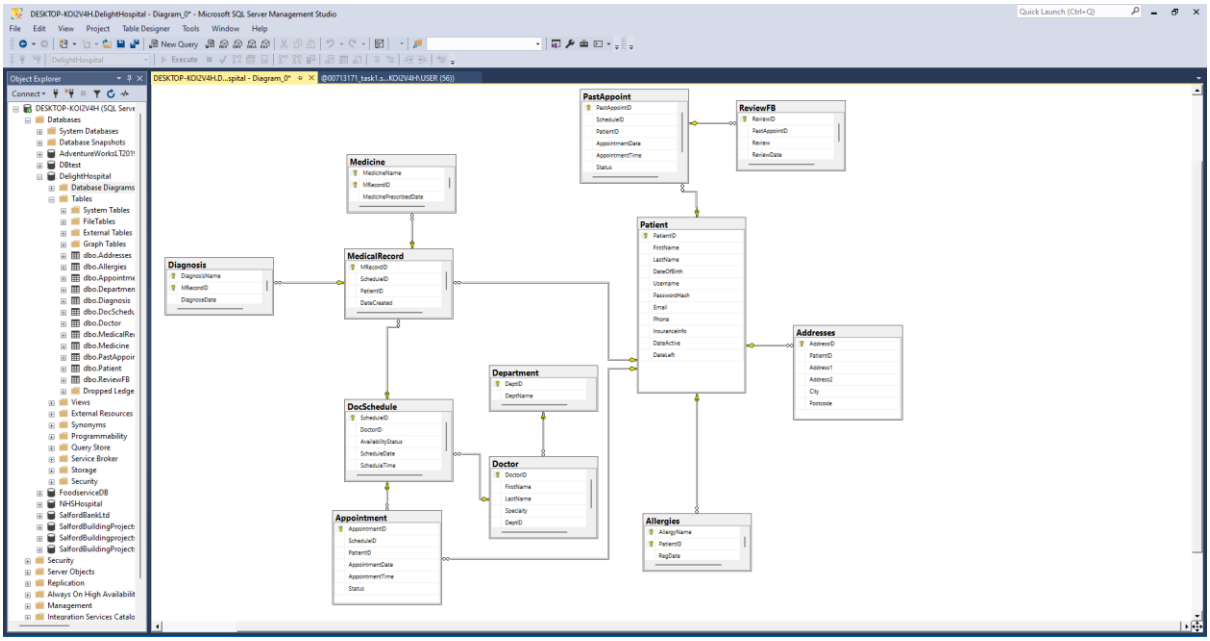
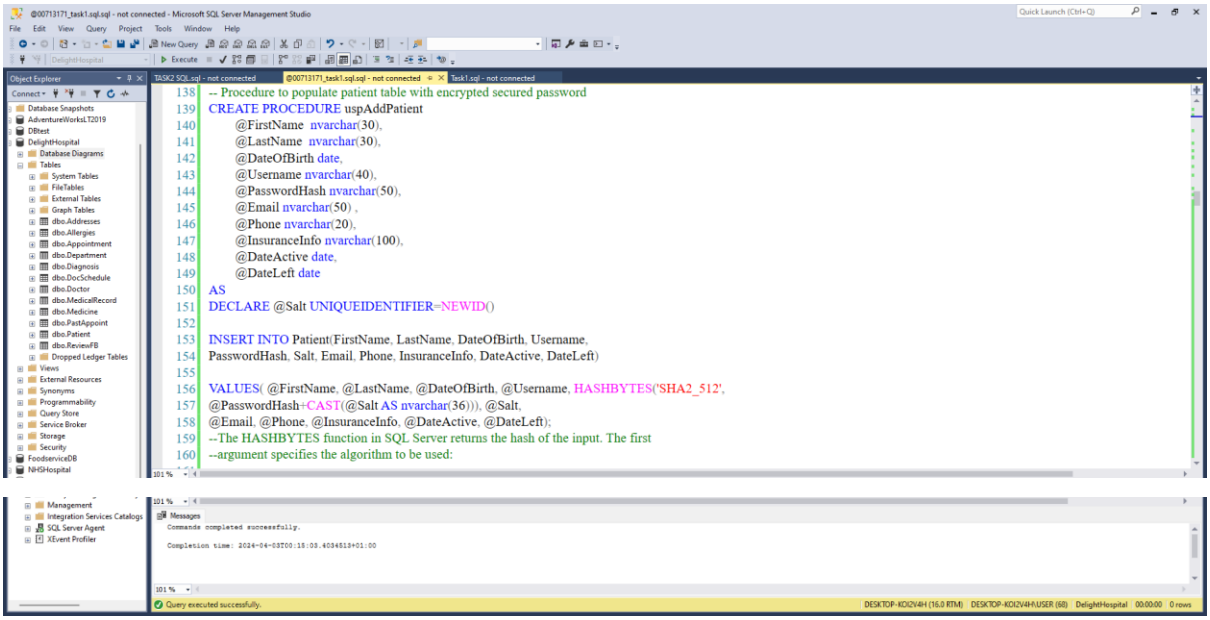



**Figure 1.2.4:** Showing the visual representation of each table and their relation.

**BELOW ARE THE STEPS TAKEN TO INSERT RECORDS INTO EACH TABLE CREATED ABOVE**

**Patient inserted record:**



```
-- Procedure to populate patient table with encrypted secured password
CREATE PROCEDURE uspAddPatient
    @FirstName  nvarchar(30),
    @LastName  nvarchar(30),
    @DateOfBirth date,
    @Username nvarchar(40),
    @PasswordHash nvarchar(50),
    @Email nvarchar(50) ,
    @Phone nvarchar(20),
    @InsuranceInfo nvarchar(100),
    @DateActive date,
    @DateLeft date
AS
DECLARE @Salt UNIQUEIDENTIFIER=NEWID()

INSERT INTO Patient(FirstName, LastName, DateOfBirth, Username,
PasswordHash, Salt, Email, Phone, InsuranceInfo, DateActive, DateLeft)

VALUES( @FirstName, @LastName, @DateOfBirth, @Username, HASHBYTES('SHA2_512',
@PasswordHash+CAST(@Salt AS nvarchar(36)), @Salt,
@Email, @Phone, @InsuranceInfo, @DateActive, @DateLeft);
--The HASHBYTES function in SQL Server returns the hash of the input. The first
--argument specifies the algorithm to be used:
```

**Figure 1.2.5:** On my patient table there is need to secure patients' passwords and for this reason I did not use the regular INSERT statement to populate the table. I have used the **binary datatype** to store **password hash**, to make it harder for password to be cracked I have also introduced the **salted hash**. **Figure 1.2.7** shows the same list of columns in my patient table. The hash of the input is returned by SQL Server's HASHBYTES function. The algorithm to be applied is specified in the first argument.

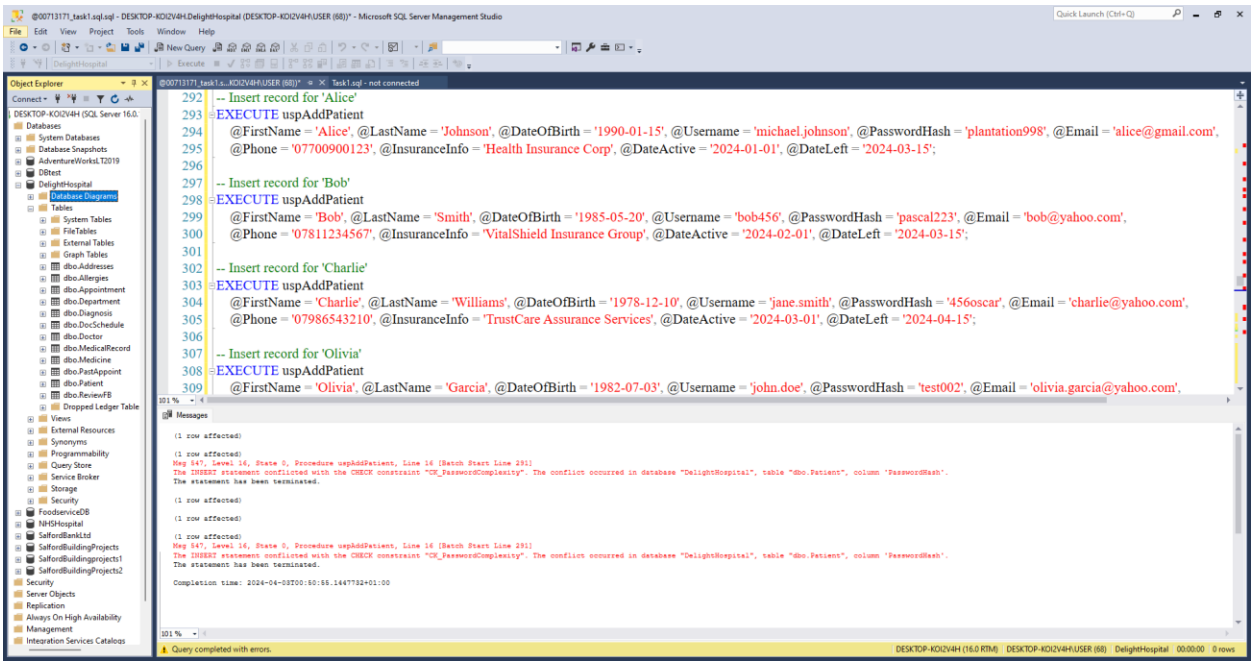



**Figure 1.2.6:** The EXECUTE statements executes the stored procedure in **Figure 1.2.5**, despite two of the five successfully inserted records not running initially

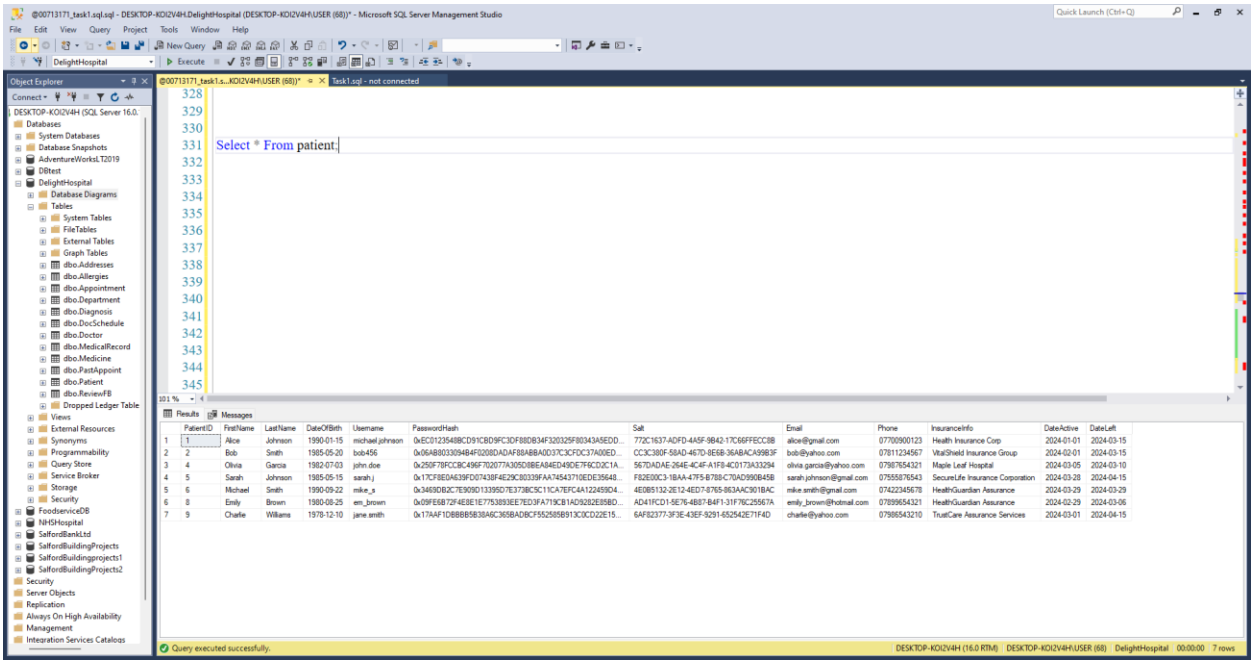



**Figure 1.2.7:** The screen short above shows the inserted records into the patient

table with the Password hashed successfully and the unique Salt column simultaneously generated accordingly.

**Address inserted record:**

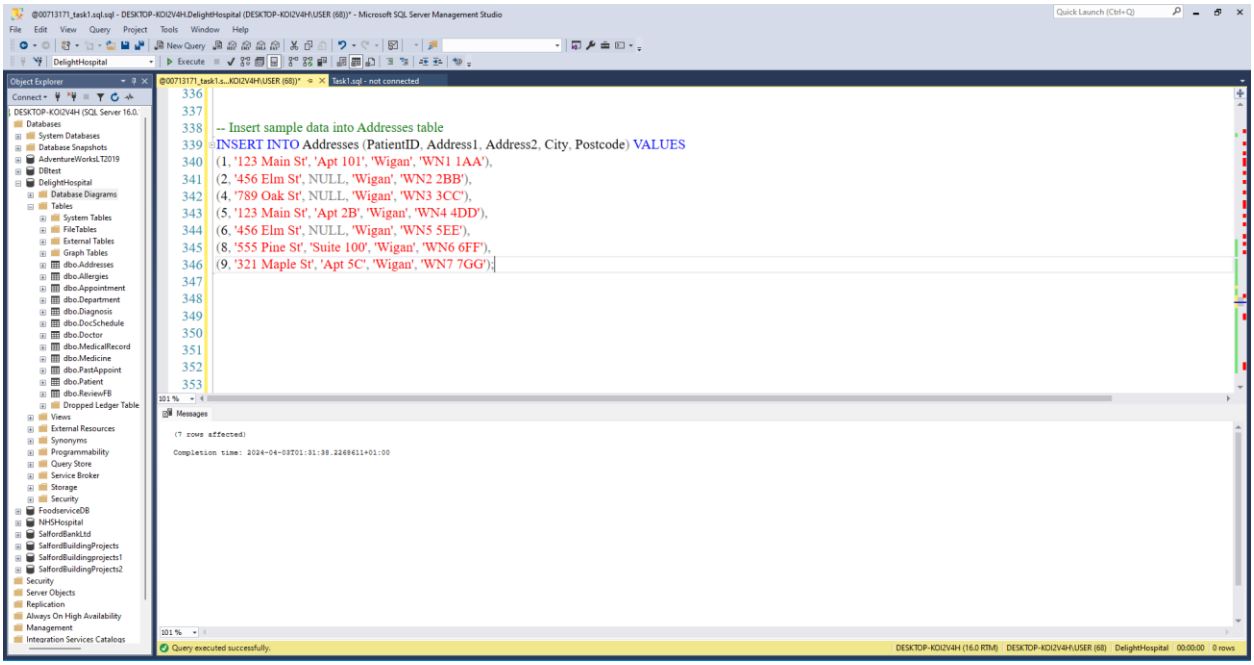



**Figure 1.2.8:** This screen short shows my written Insert TSQL query for Address table and was returned successful with seven (7) record.

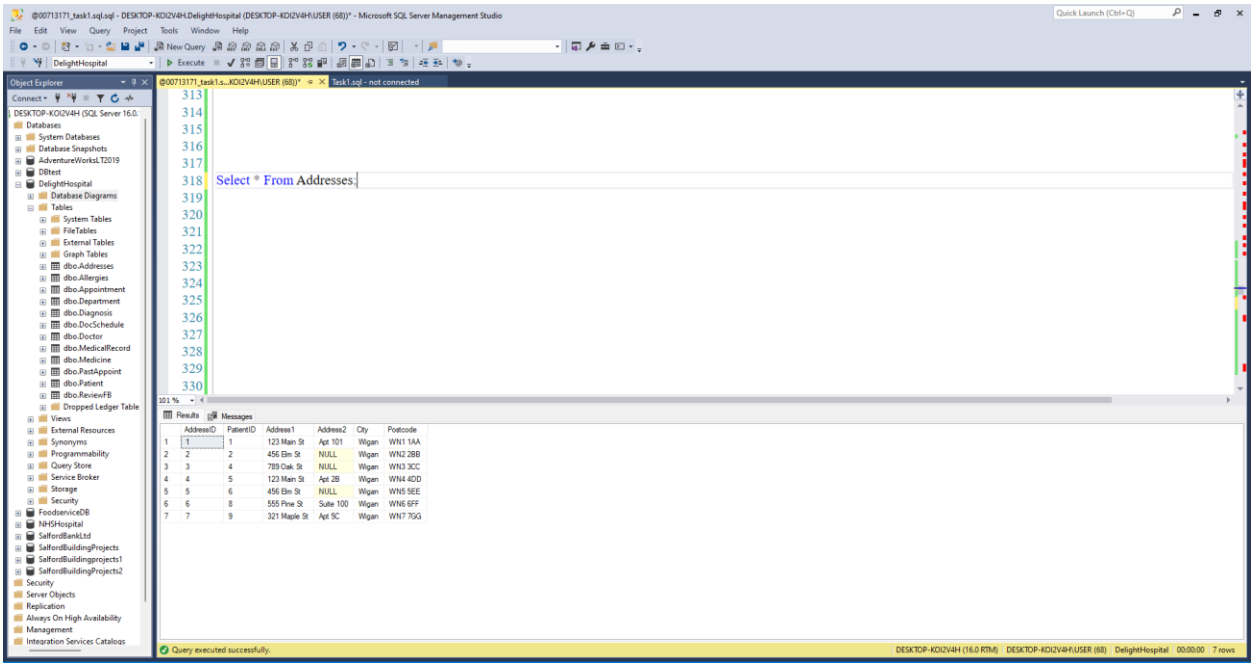



**Figure 1.2.9:** Here is the result for address

## Allergies inserted record:

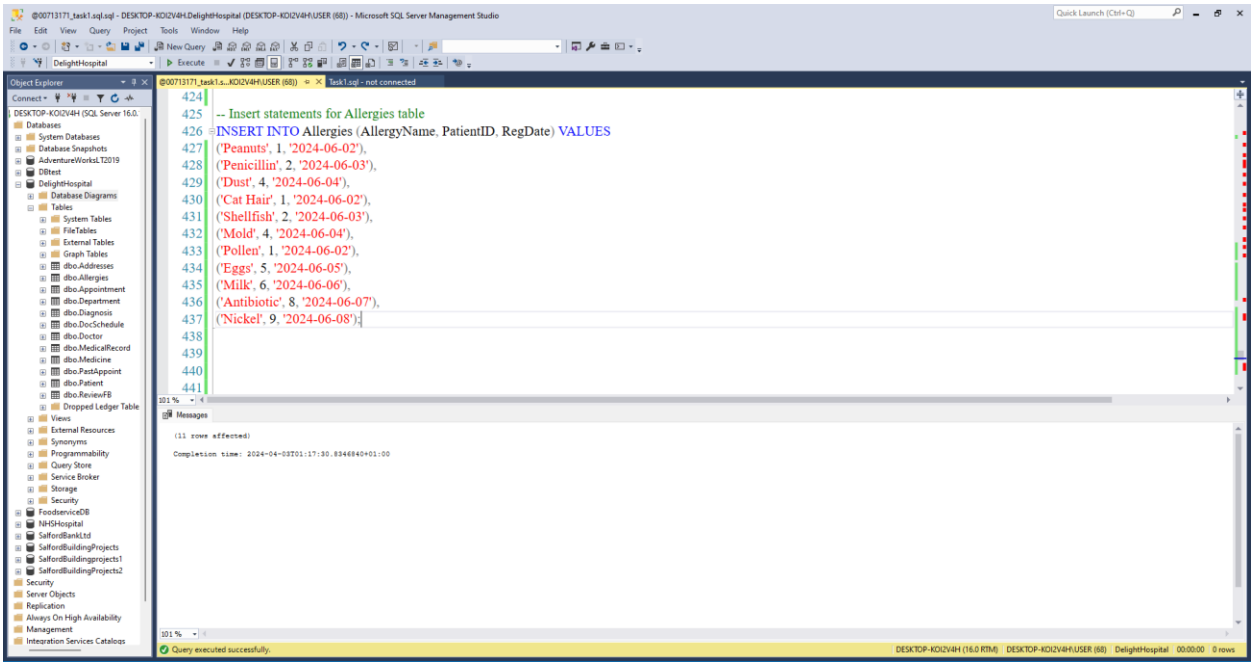



**Figure 1.3.0:** This screen short shows my written Insert TSQL query for Allergies table and was returned successful with eleven (11) record.

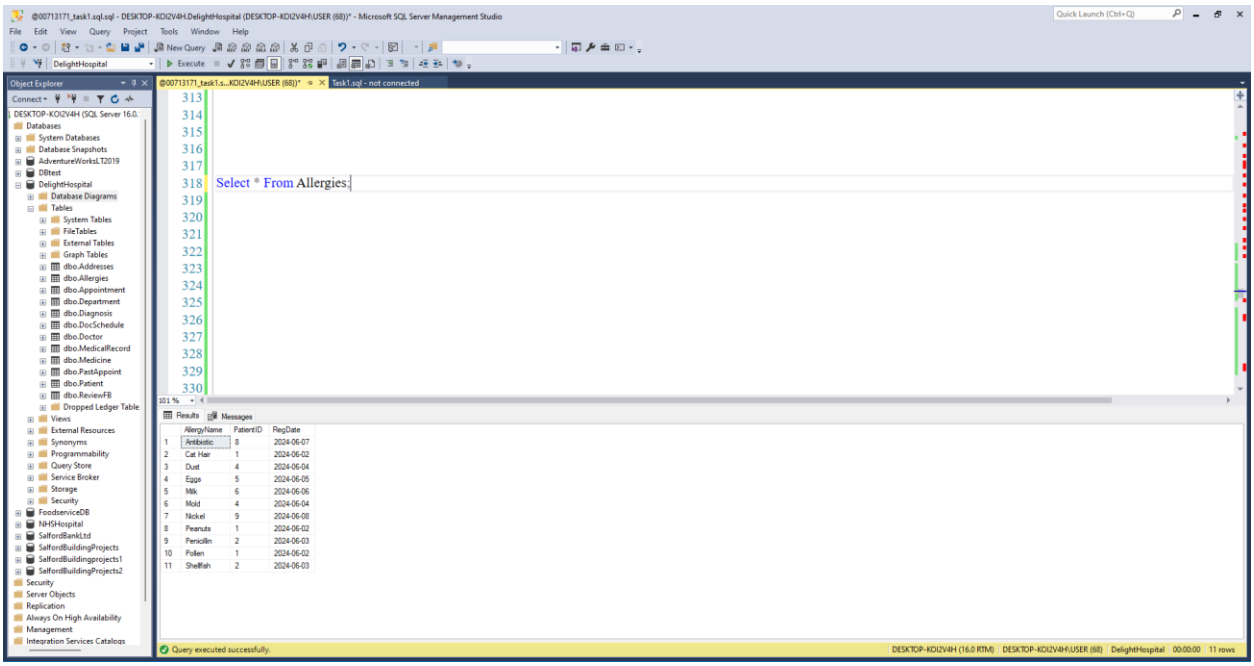



**Figure 1.3.1:** Here is the result for Allergies

## Department inserted record:

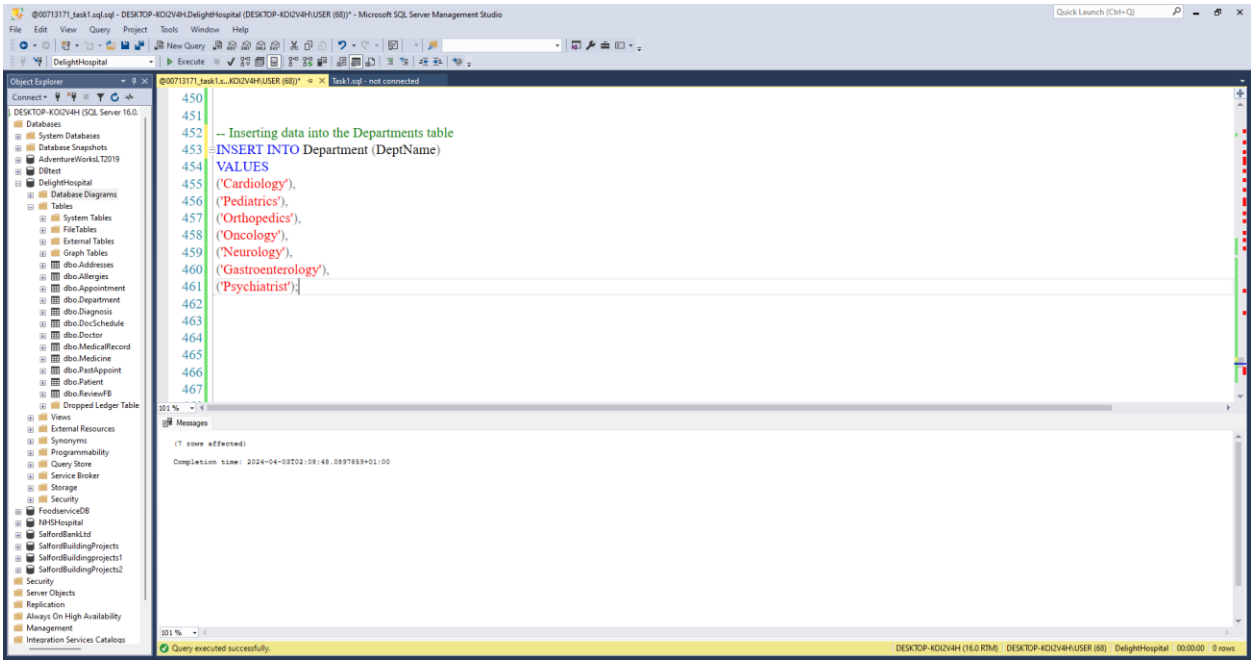



**Figure 1.3.2:** This screen short shows my written Insert TSQL query for Department table and was returned successful with seven (7) record.

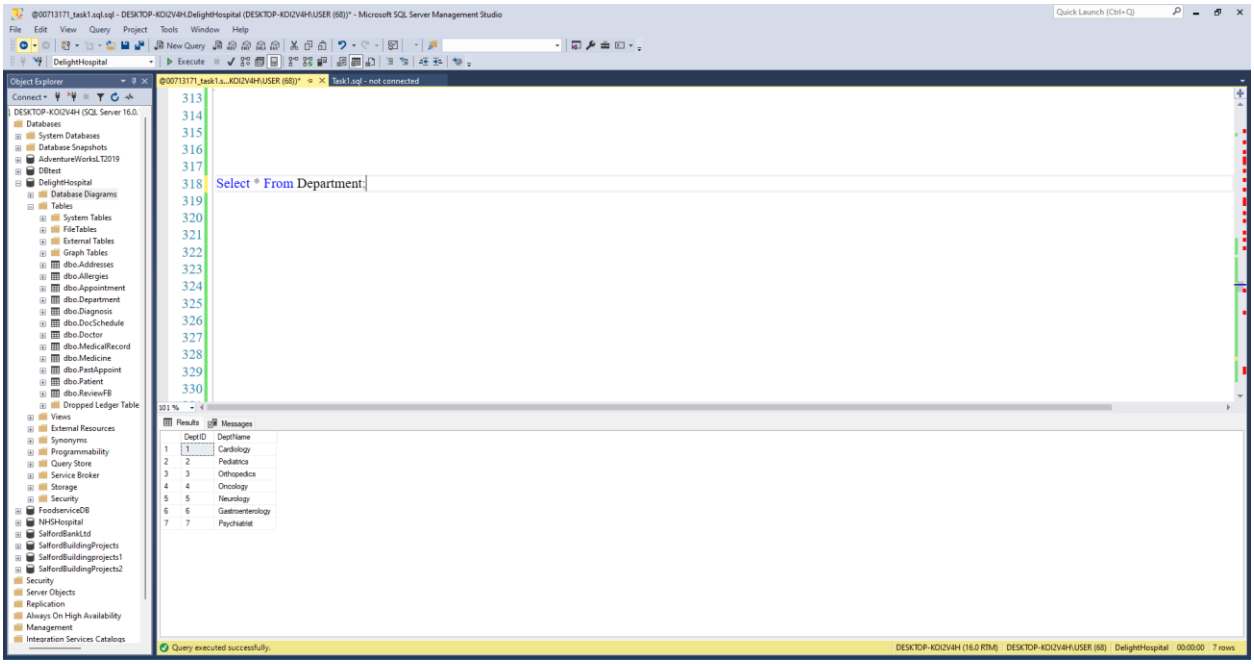



**Figure 1.3.3:** Here is the result for Department

## Doctor inserted record:

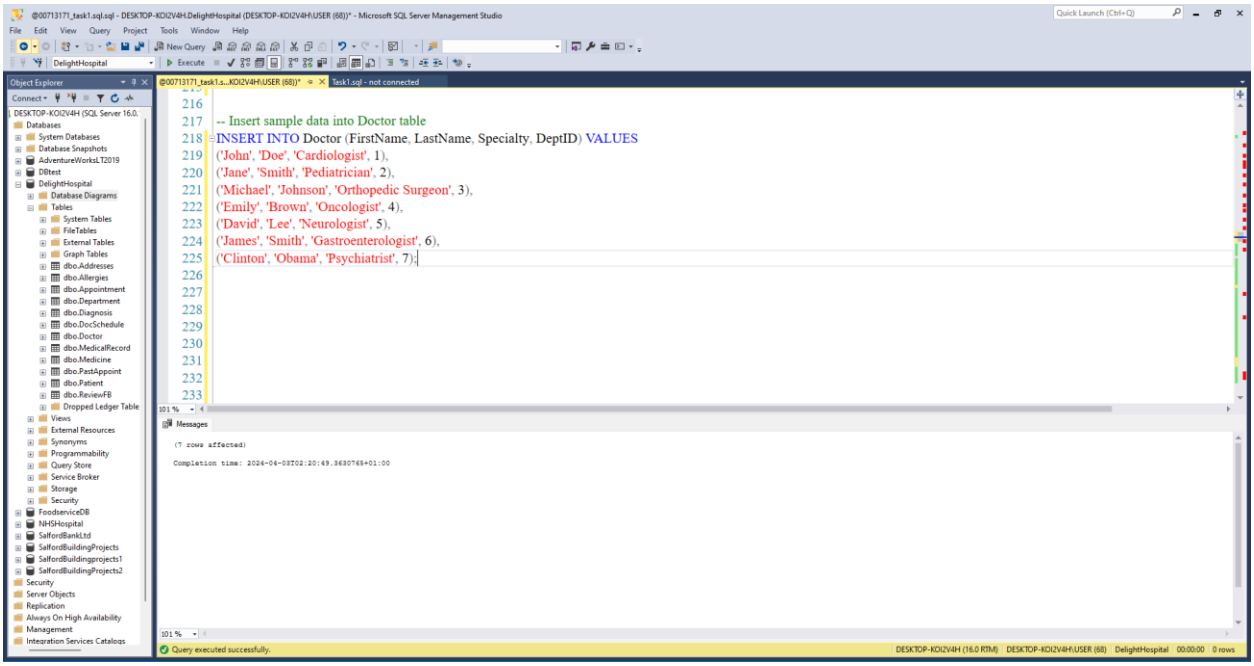


**Figure 1.3.4:** This screen short shows my written Insert TSQL query for Doctor table and was returned successful with seven (7) record.

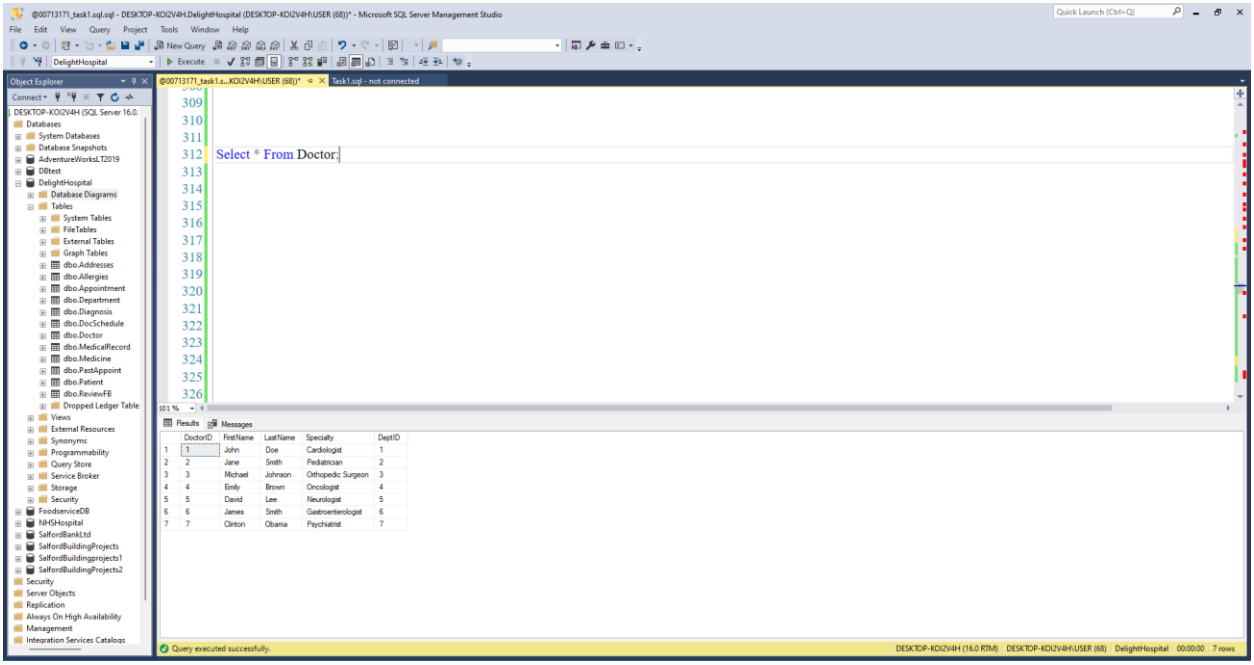



**Figure 1.3.5:** Here is the result for Doctors

## DocSchedule inserted record:

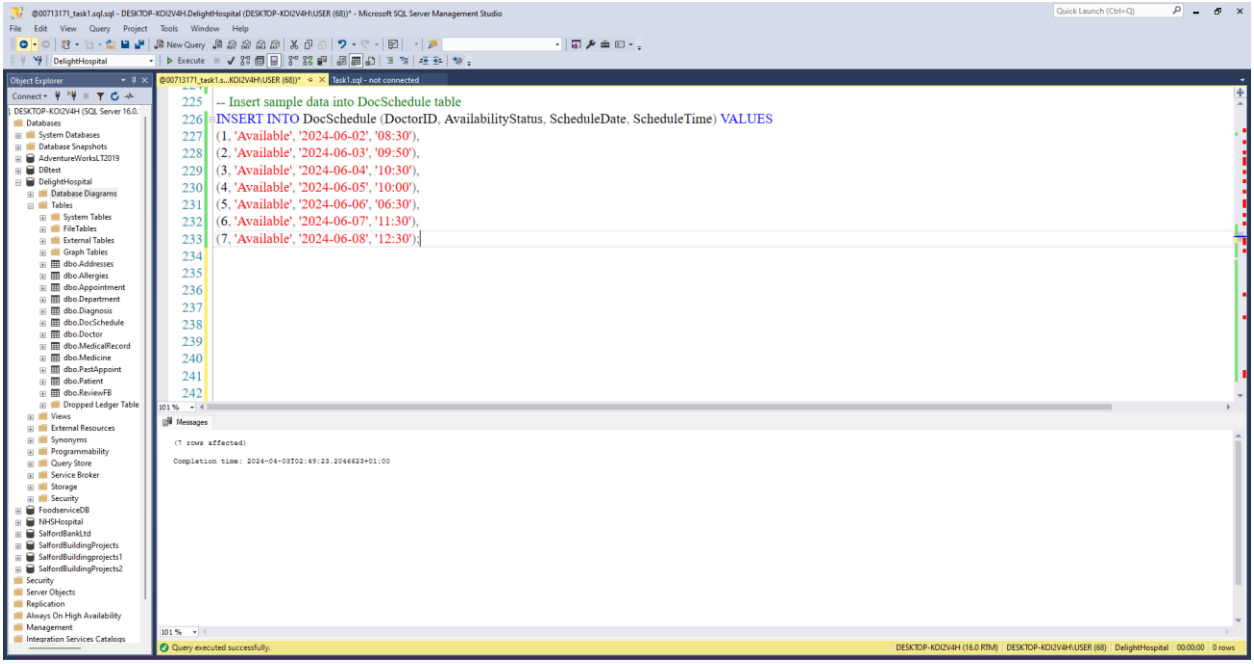


**Figure 1.3.6:** This screen short shows my written Insert TSQL query for DocSchedule table and was returned successful with seven (7) record.

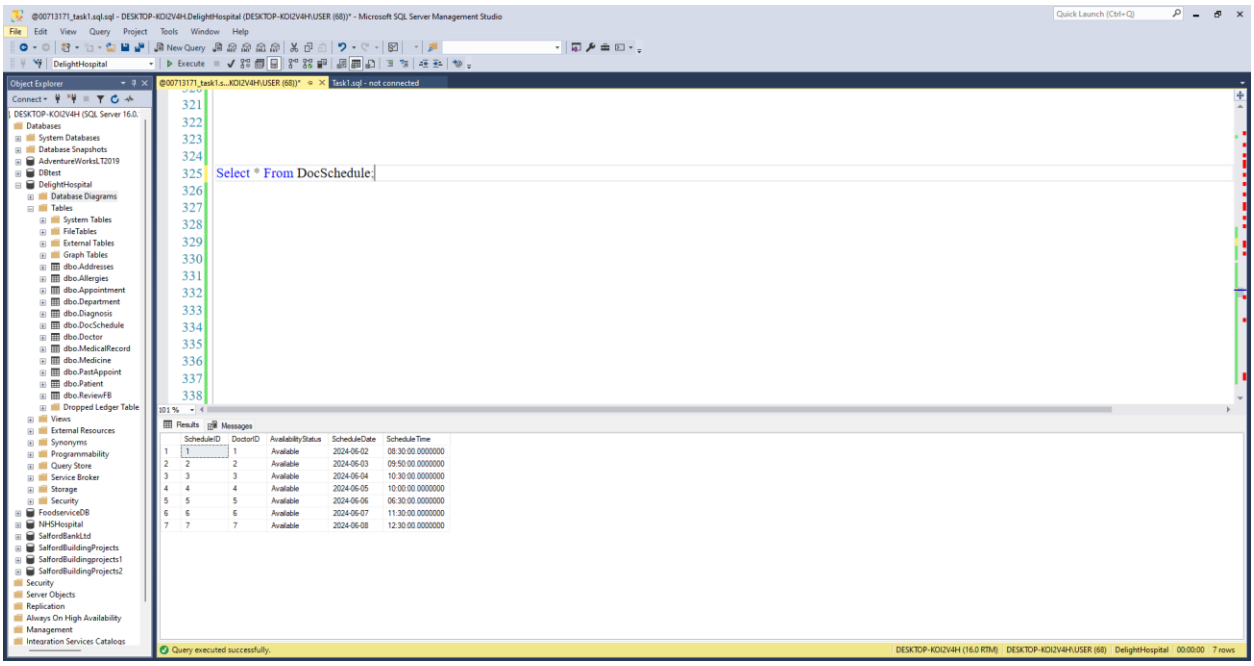



**Figure 1.3.7:** Here is the result for DocSchedule

## Appointment inserted record:

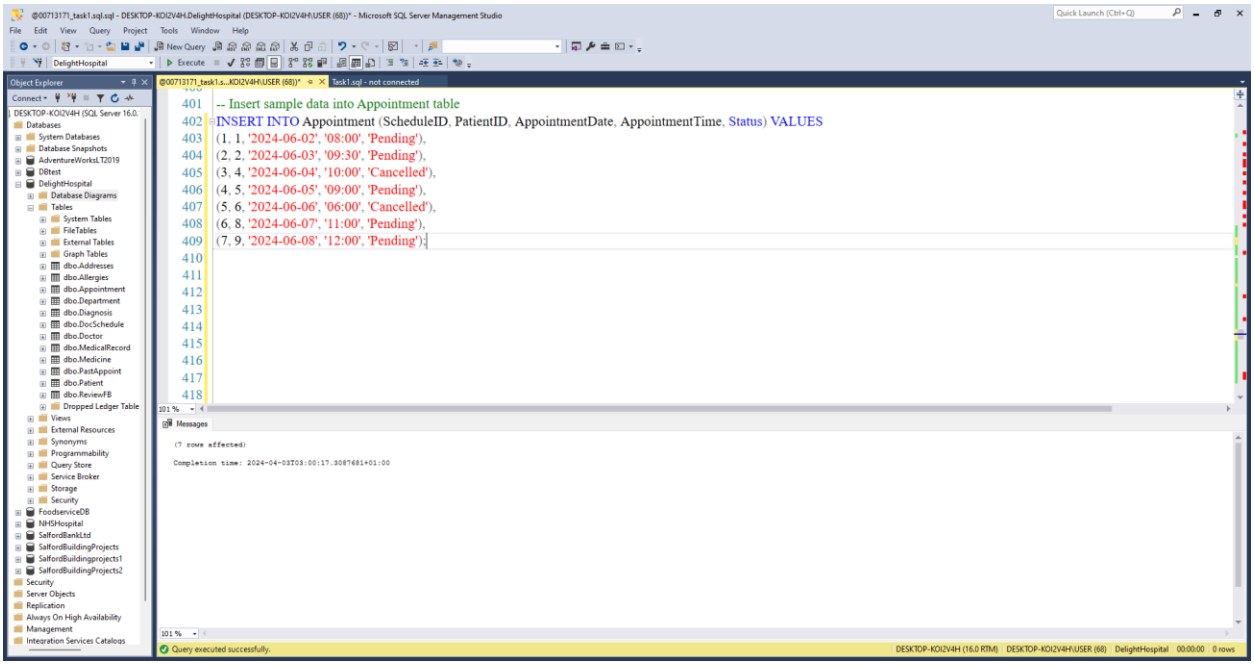



**Figure 1.3.8:** This screen short shows my written Insert TSQL query for Appointment table and was returned successful with seven (7) record.

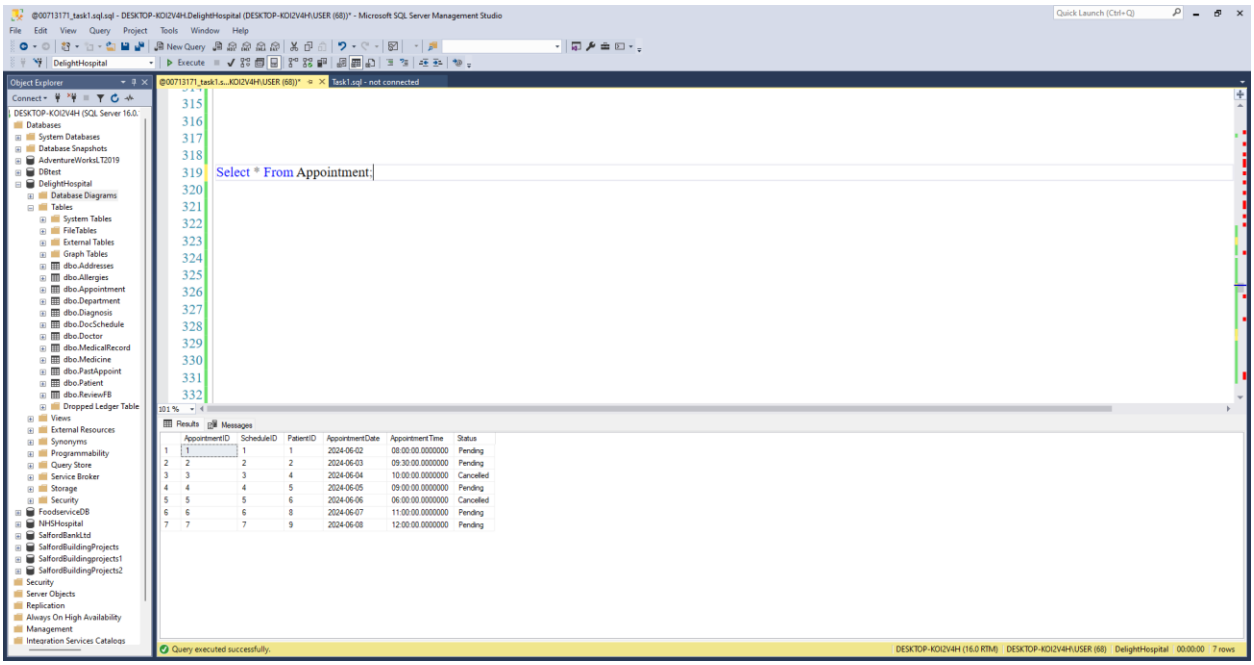



**Figure 1.3.9:** Here is the result for Appointment

**PastAppoint inserted record:**

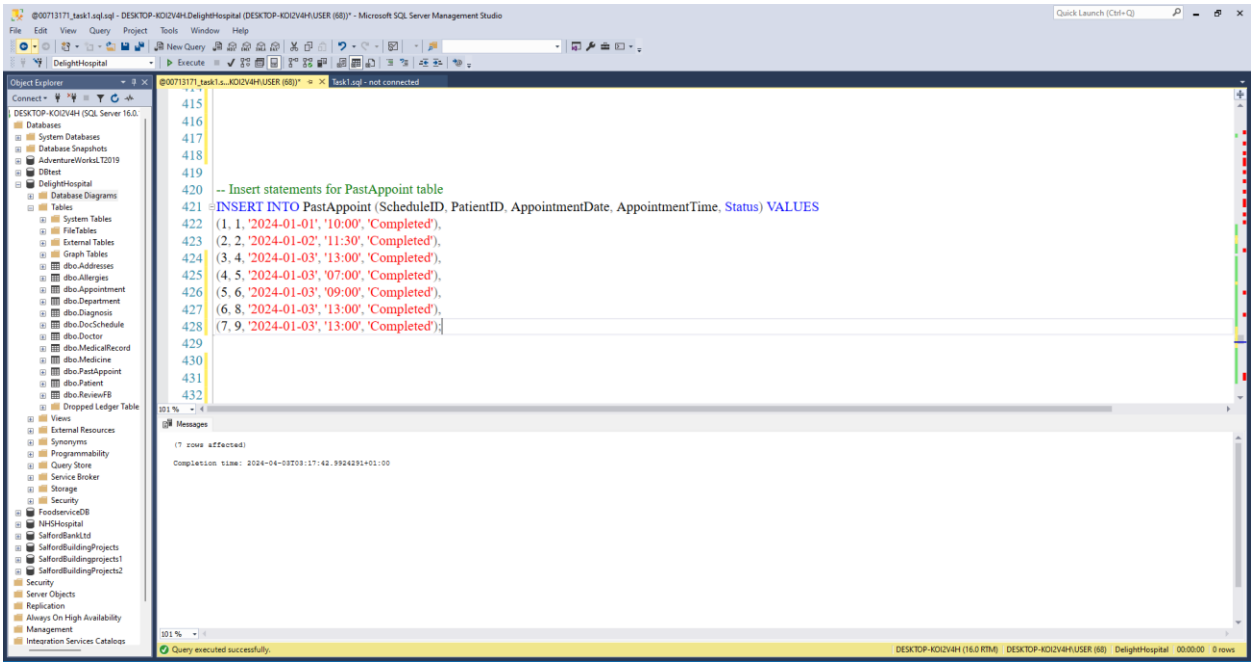



**Figure 1.4.0:** This screen short shows my written Insert TSQL query for PastAppoint table and was returned successful with seven (7) record.

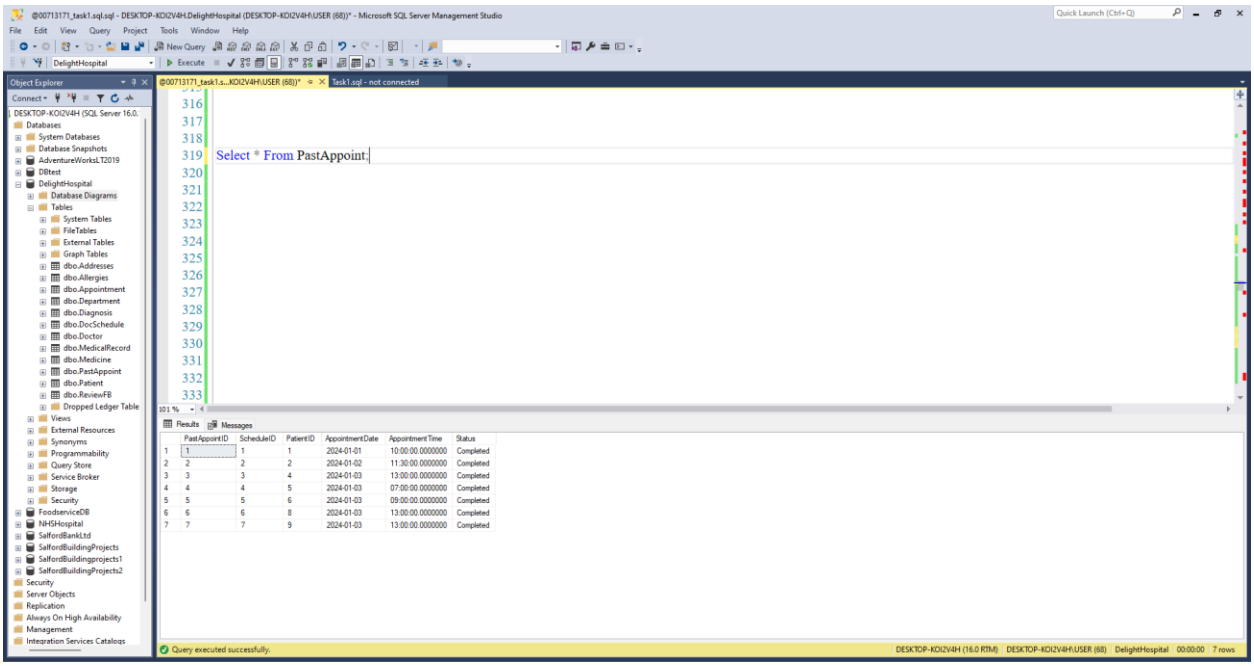



**Figure 1.4.1:** Here is the result for address

**ReviewFB inserted record:**

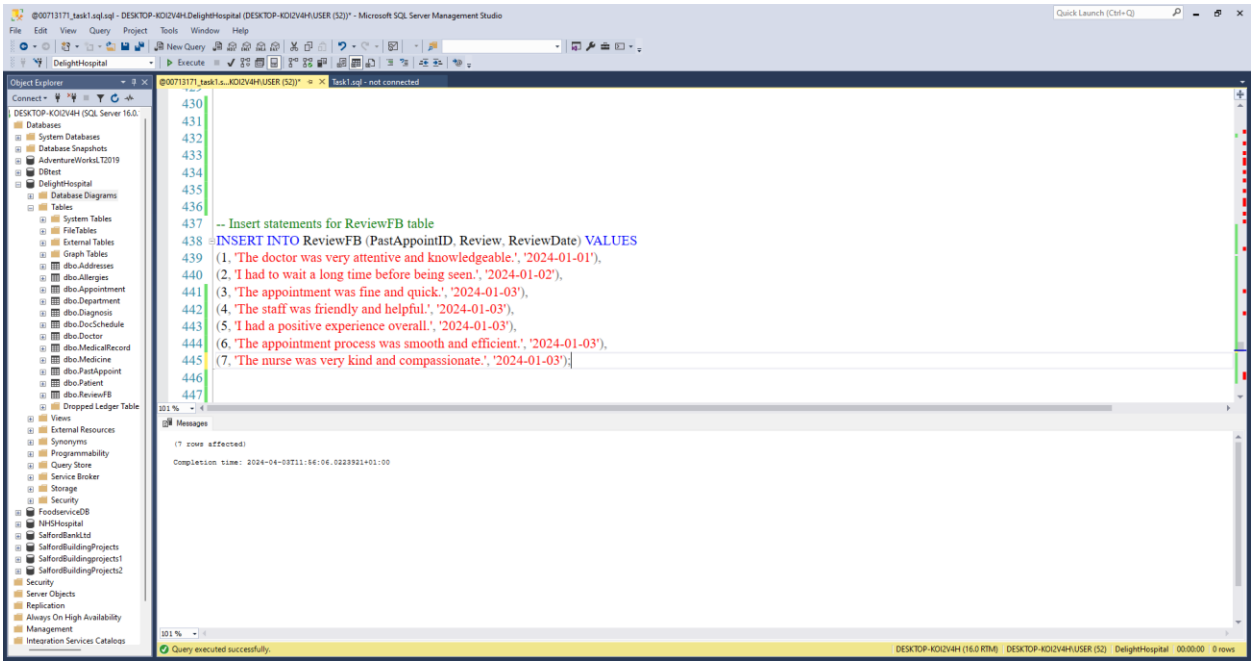



**Figure 1.4.2:** This screen short shows my written Insert TSQL query for **ReviewFB** table (review feedback record) and was returned successful with seven (7) record.

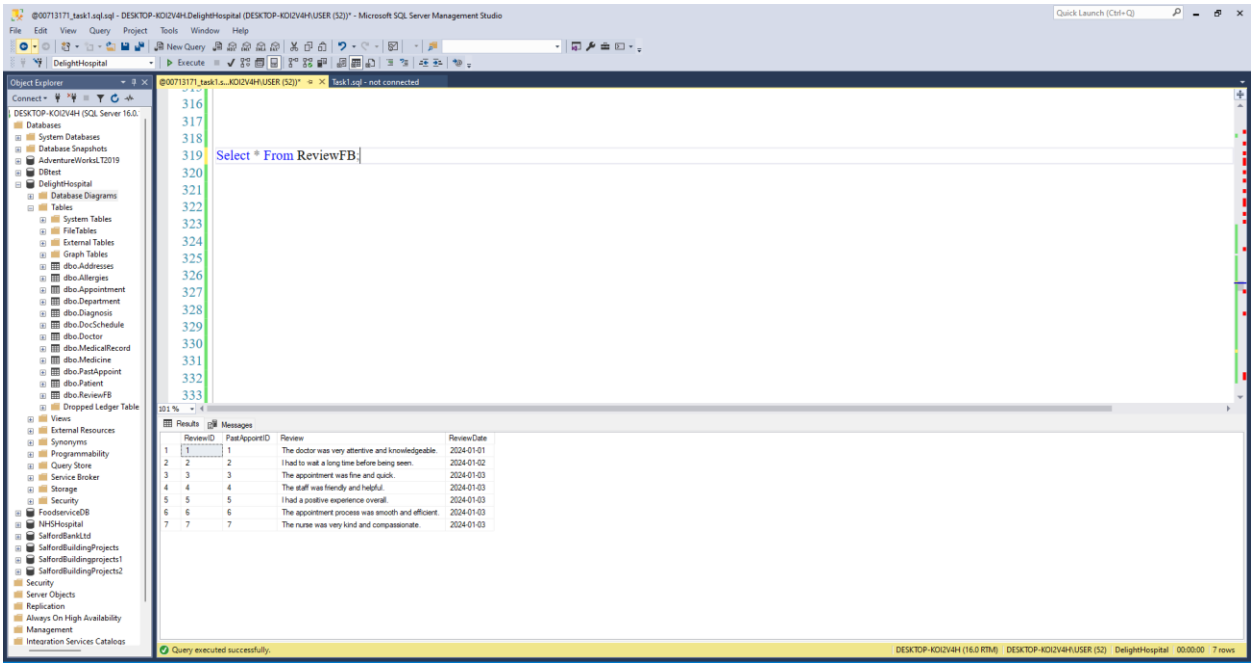



**Figure 1.4.3:** Here is the result for address

## MedicalRecord inserted record:

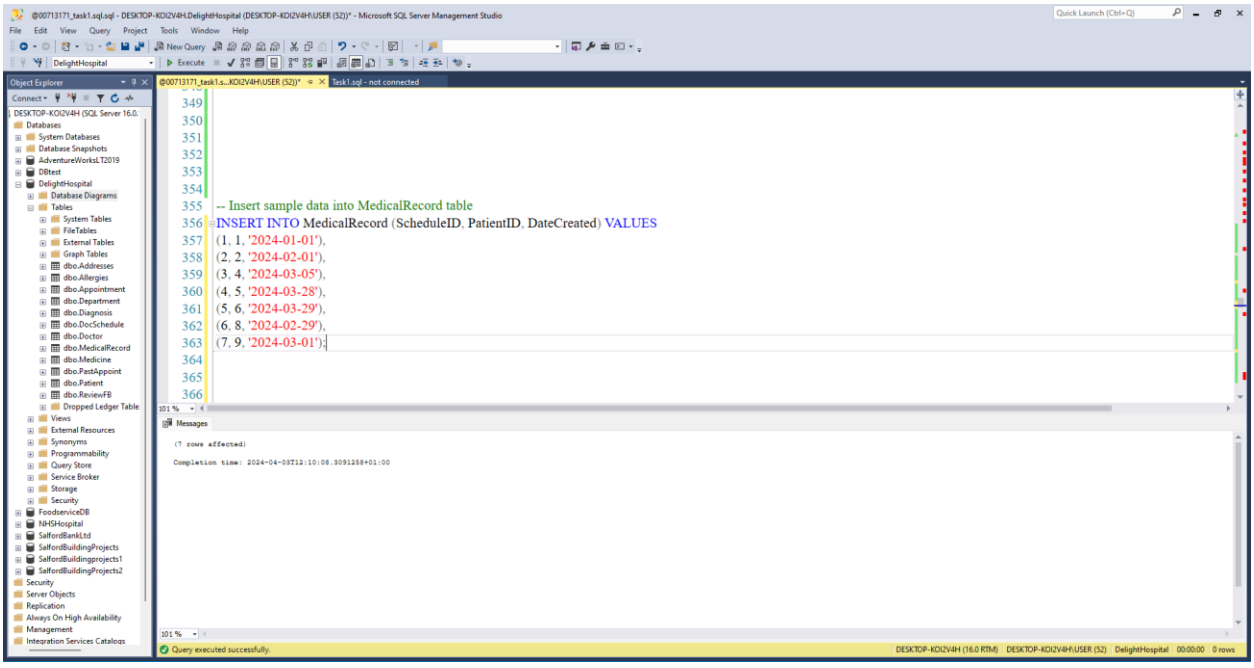



**Figure 1.4.4:** This screen short shows my written Insert TSQL query for **MedicalRecord** table and was returned successful with seven (7) record.

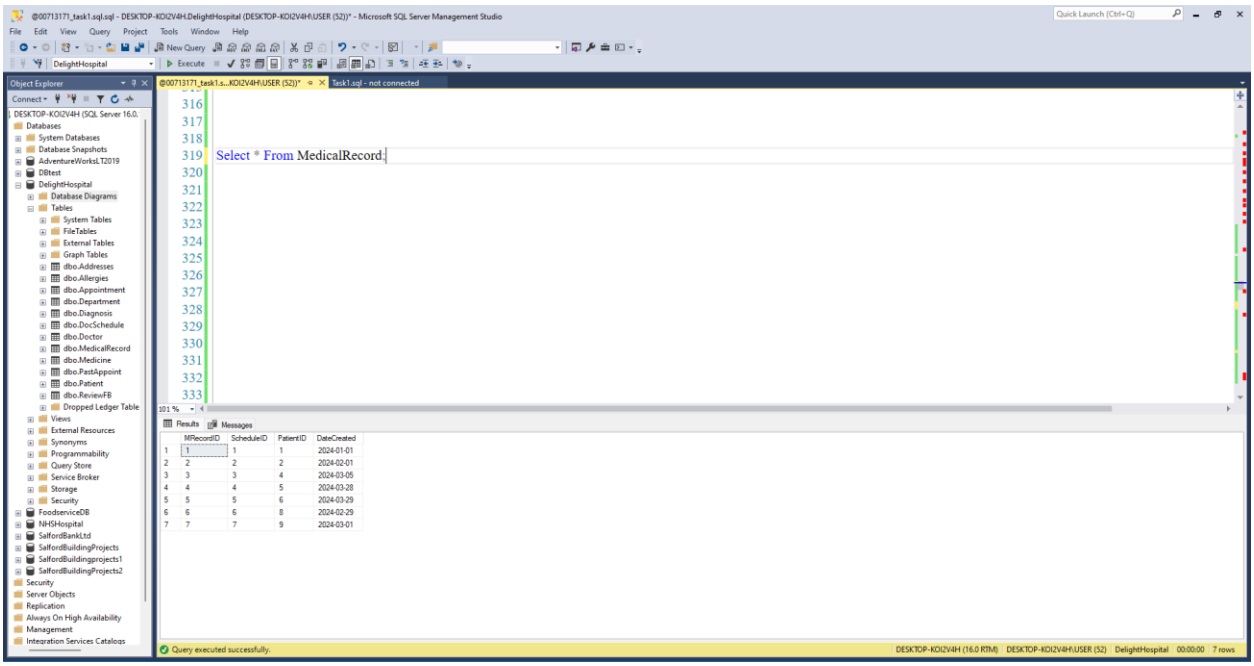



**Figure 1.4.5:** Here is the result for MedicalRecord

## Diagnosis inserted record:

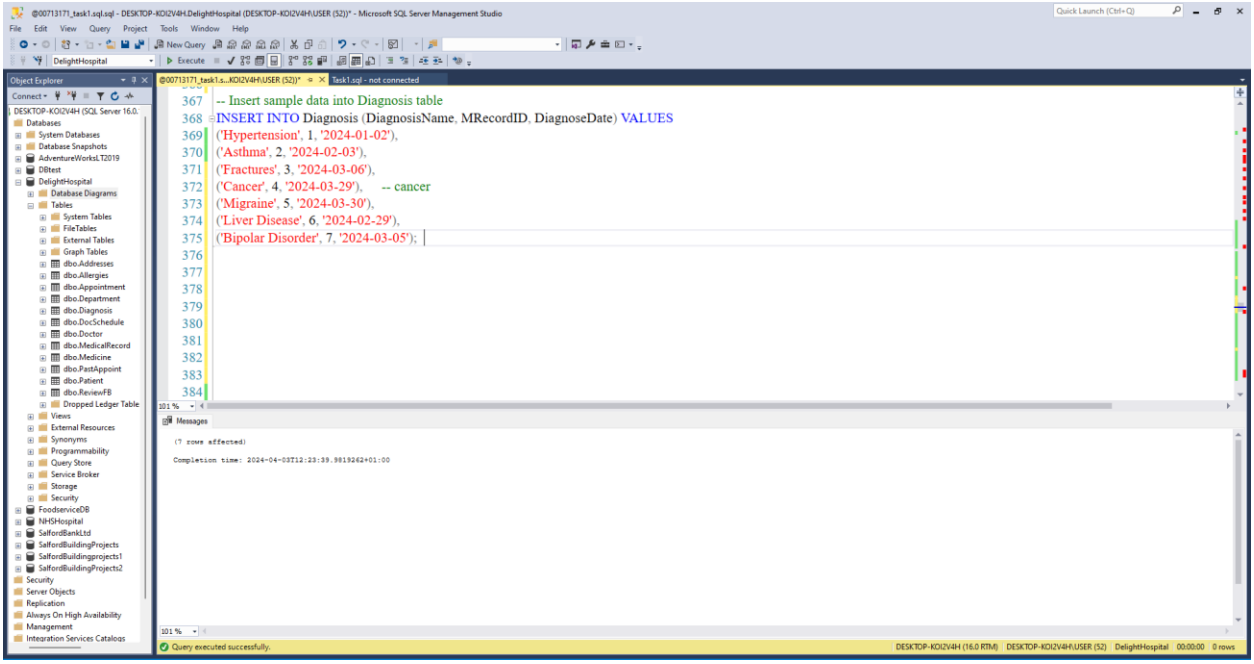


**Figure 1.4.6:** This screen short shows my written Insert TSQL query for Diagnosis table and was returned successful with seven (7) record.

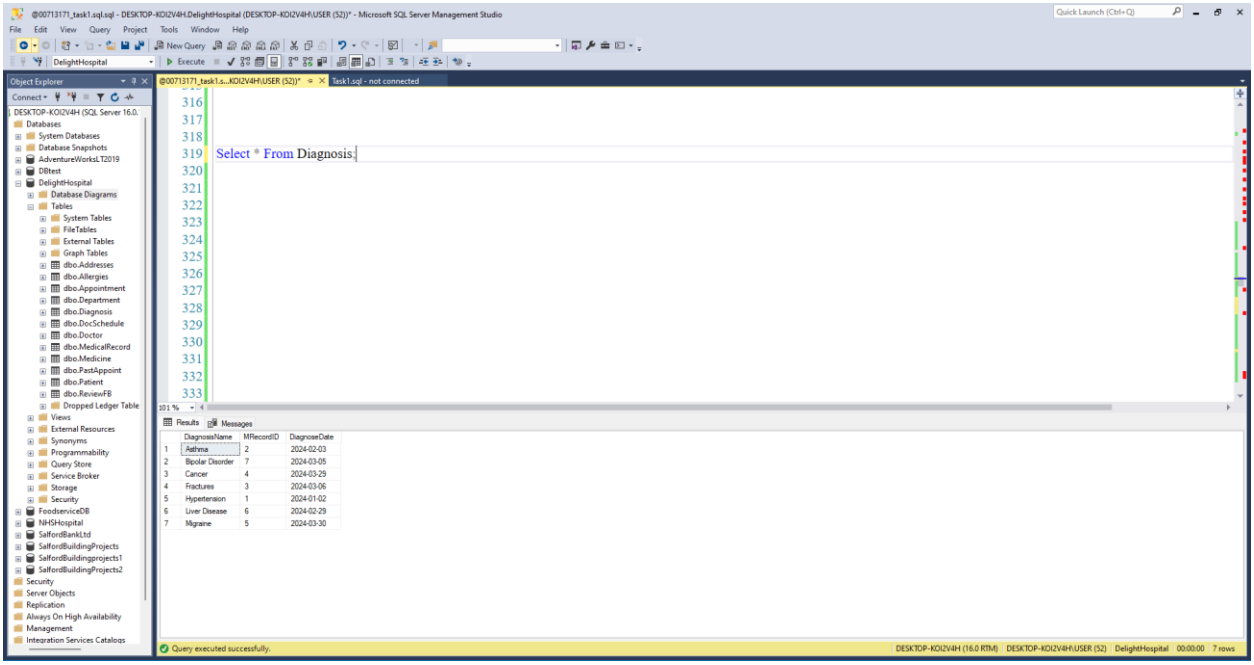



**Figure 1.4.7:** Here is the result for Diagnosis

## Medicine inserted record:

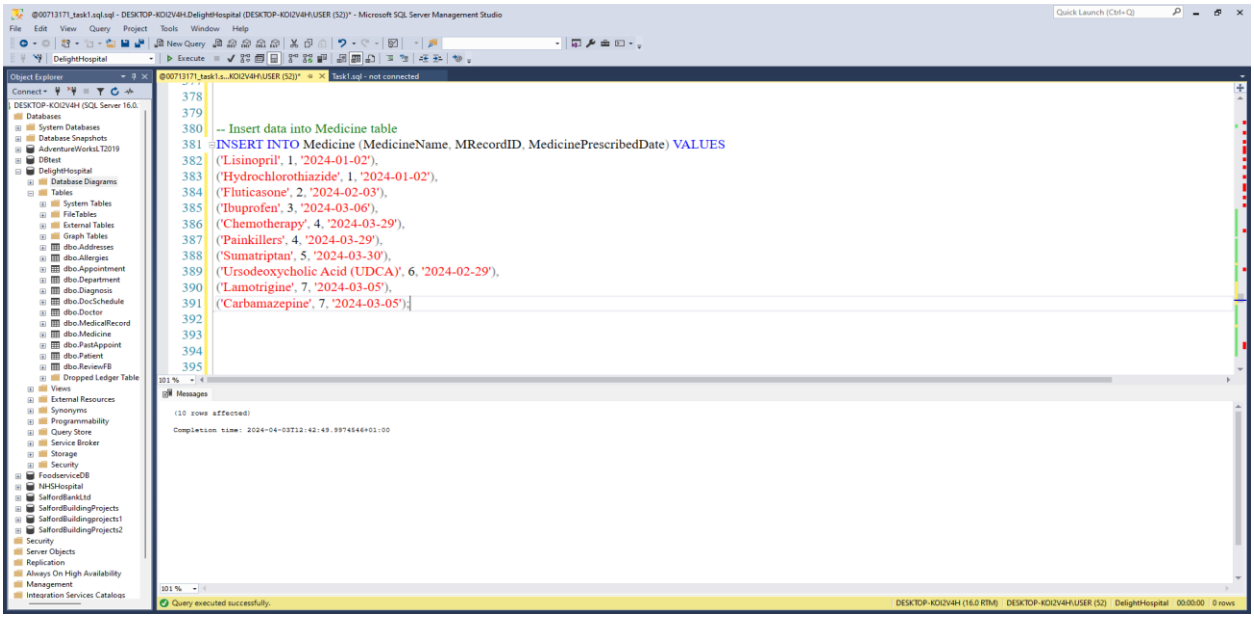


**Figure 1.4.8:** This screen short shows my written Insert TSQL query for Medicine table and was returned successful with seven (7) record.

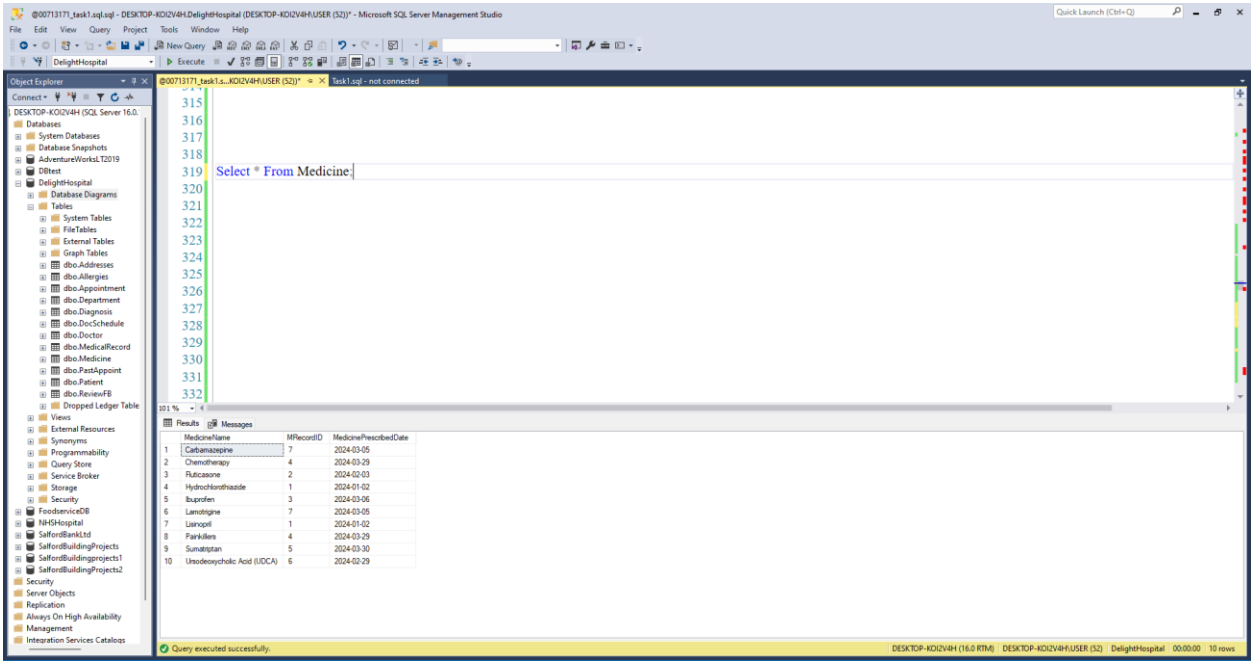



**Figure 1.4.9:** Here is the result for Medicine

**PART 2:**

**Q**2. Adding constraint to check the appointment date is not in the past:

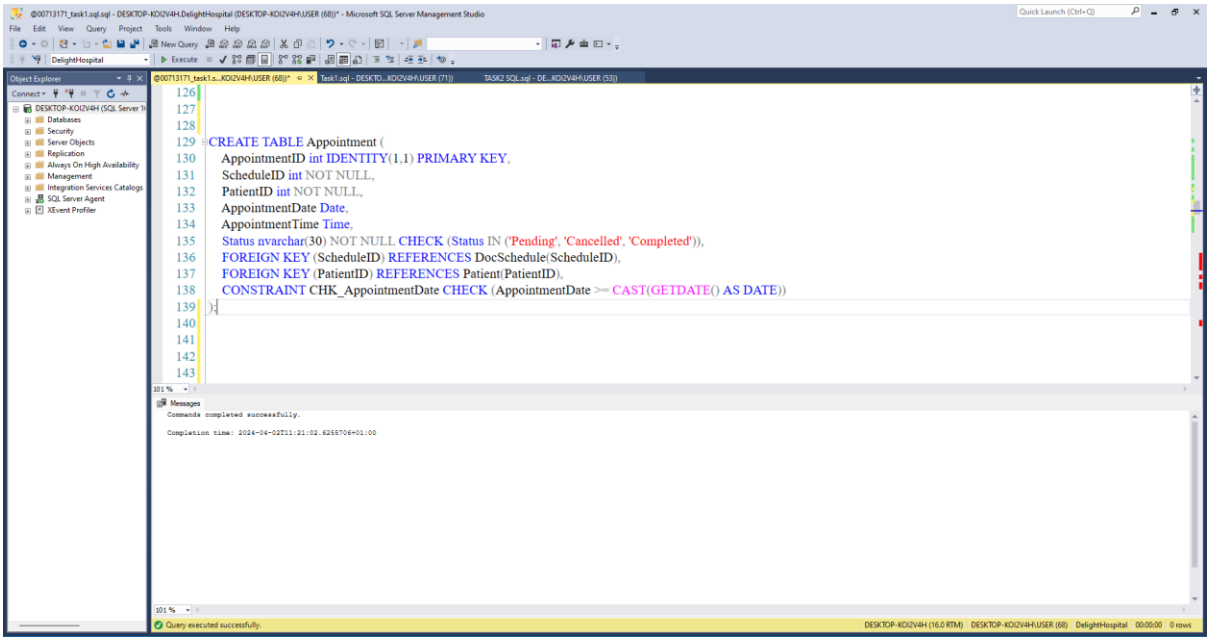



**Figure 1.5.0: CHECK constraint:** The AppointmentDate is guaranteed to be present (i.e. larger than or equal to the current date) by the CHECK constraint **CHK_AppointmentDate**.

**Q**3.

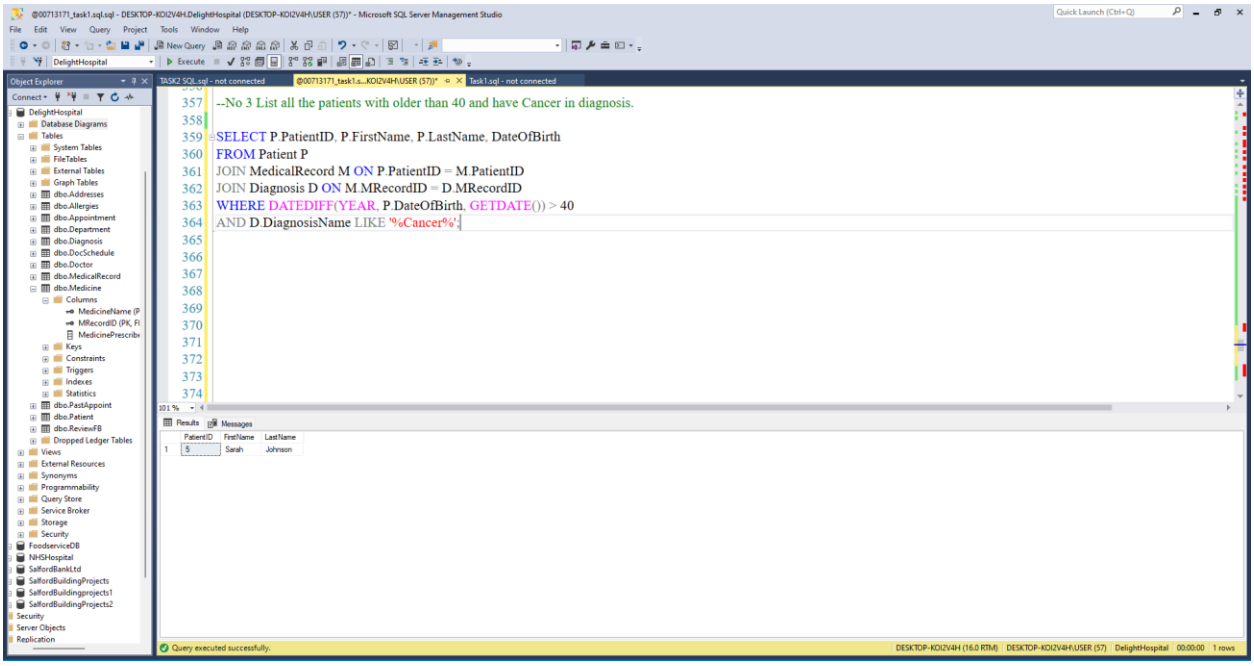



**Figure 1.5.1:** This TSQL query above answers question two (3) by using select statement to list some patient details including names and date of birth from patient table aliased with “P” and using “JOIN” to join MedicalRecord based on PatientID in other to join Diagnosis table using MRecordID and finally use the WHERE clause to filter the rows needed. Result from the executed query is derived by getting difference between system date using “GETDATE()” and “DateOfBirth” but with additional condition using LIKE “%Cancer%” to get patient who also have Cancer as string.

**Q**4a.

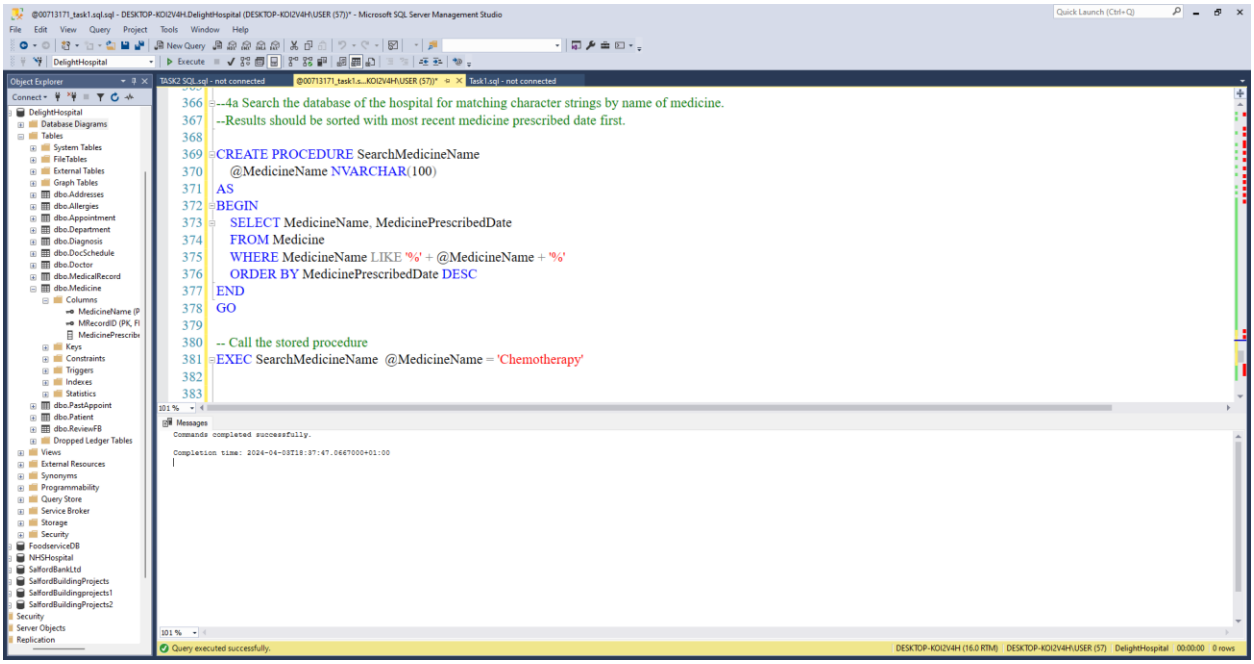



**Figure 1.5.2:** I named the stored procedure as "**SearchMedicineName**" intended to fetch records from the Medicine table. To retrieve the "MedicineName" and "MedicinePrescribedDate" columns from the Medicine table, a "SELECT" query is used at the beginning. Only the records having a "MedicineName" that contains the input string (given by the @MedicineName parameter) are included, with the WHERE clause's filtering of the data. To ensure that the most current prescriptions display first the "ORDERBY" clause is introduced and "DESC" also used to put the result in descending order based on the "MedicinePrescribedDate" column. The Stored procedure runs successfully:

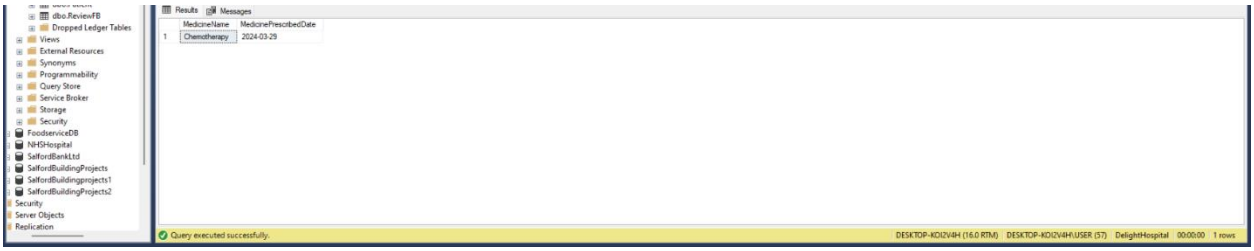


**Figure 1.5.3:** The **@MedicineName** argument is passed to the value 'Chemotherapy', and used the "**EXEC**" keyword to search using the stored procedure.

**Q**4b.

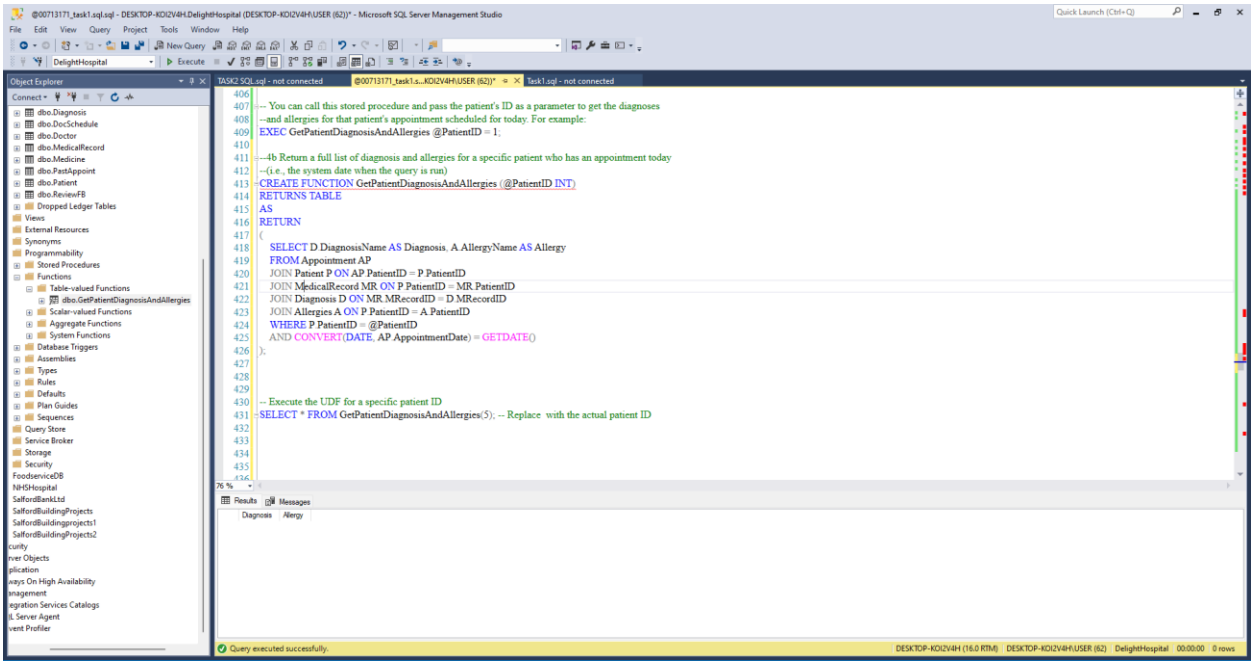


**Figure 1.5.4:** At the explorer window we can see the User defined function installed. The function accepts a specific patient ID as input and outputs a table containing the patient's diagnosis and allergy details. To obtain the results, I use the select statement to run the function name with the intended patient ID it was successful but there was no appointment today.

## Q4c. **Updating the details for an existing doctor**

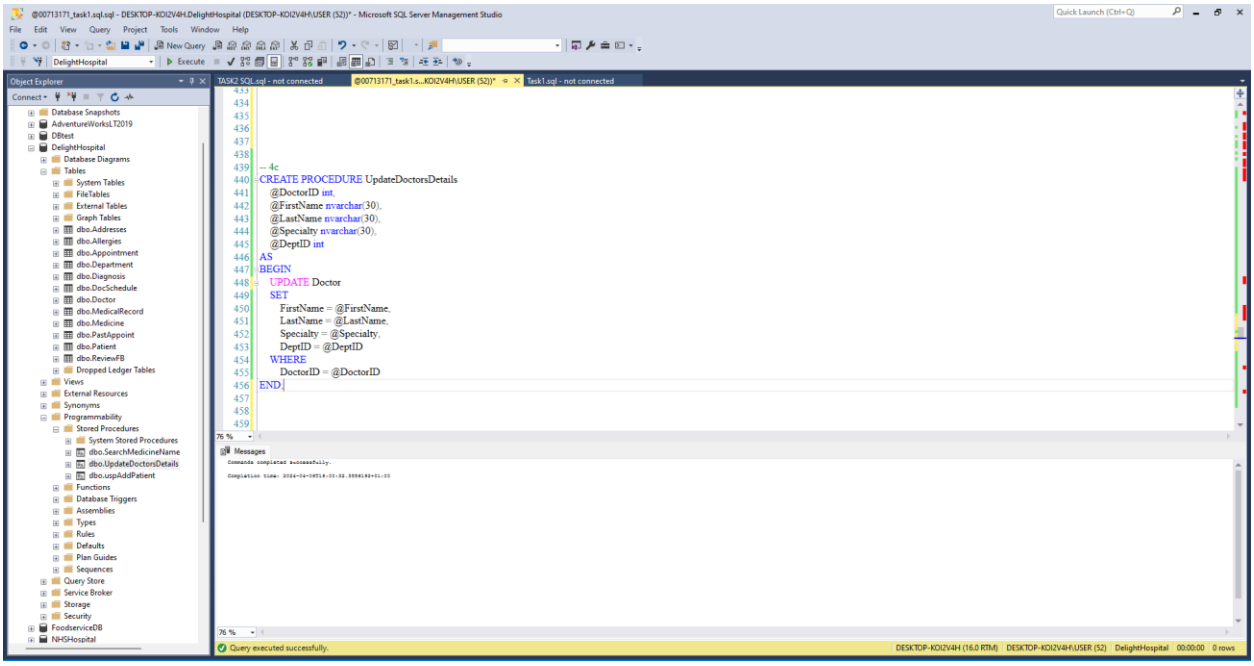



**Figure 1.5.5:** I created the stored procedure to update Doctor records by first giving it a name "UpdateDoctorDetails". I took certain parameter for example doctors ID = (@DoctorID) etc.

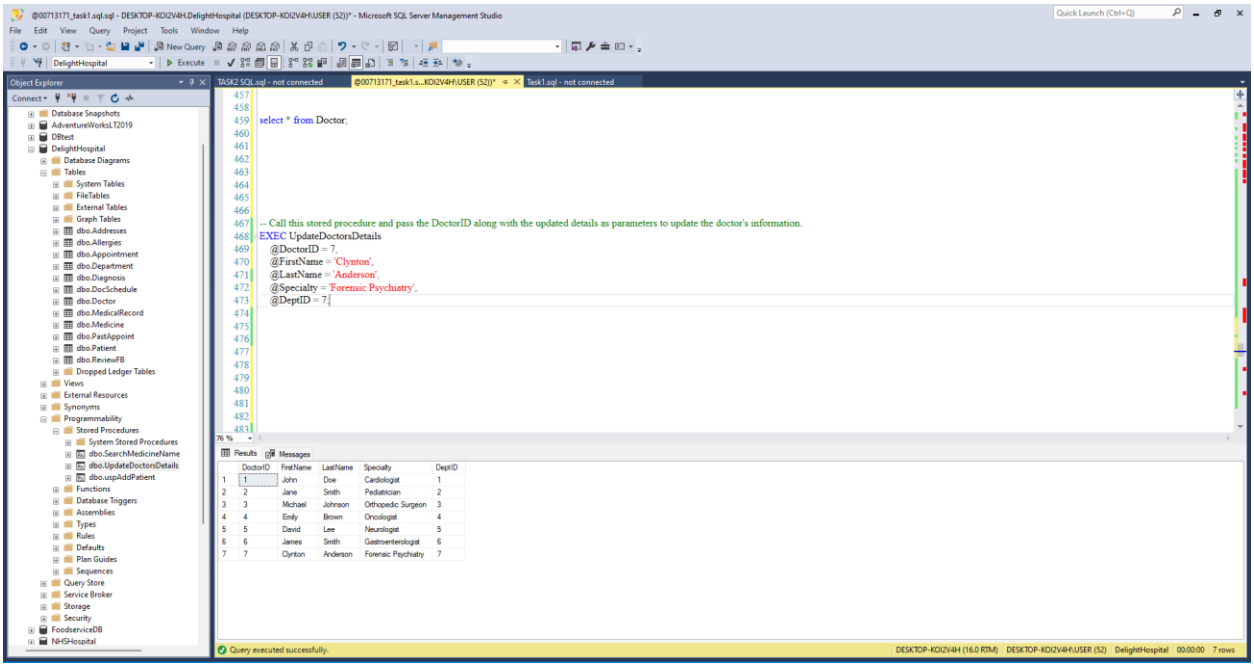



**Figure 1.5.6:** Then the stored procedure updates the supplied data for the designated doctor ID by invoking the stored procedure using **EXEC** as shown in the slide above. I updated the record of doctor with Doctor ID "7" by correcting his name from **Clinton to Clynton**, and updating the specialty from **Psychiatry to Forensic Psychiatry**.

**Q**4d. **Delete the appointment whos' status is already completed.**

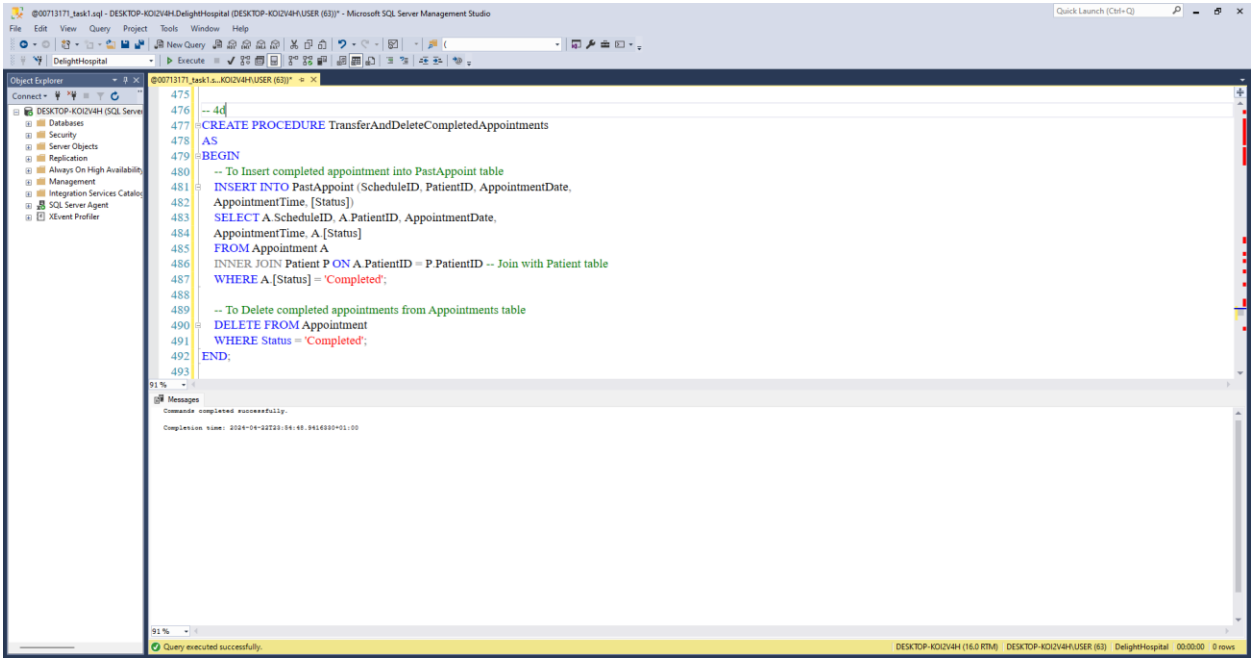



**Figure 1.5.7:** After giving the stored procedure a name, I've linked the Patient and the Appointment table together using INNER JOIN to move the historical record, the "INSERT INTO" statement ensures the authenticity of related patients by transferring completed appointments from the Appointment table to the PastAppoint table. Immediately after the record is moved it is then deleted from the Appointment table.

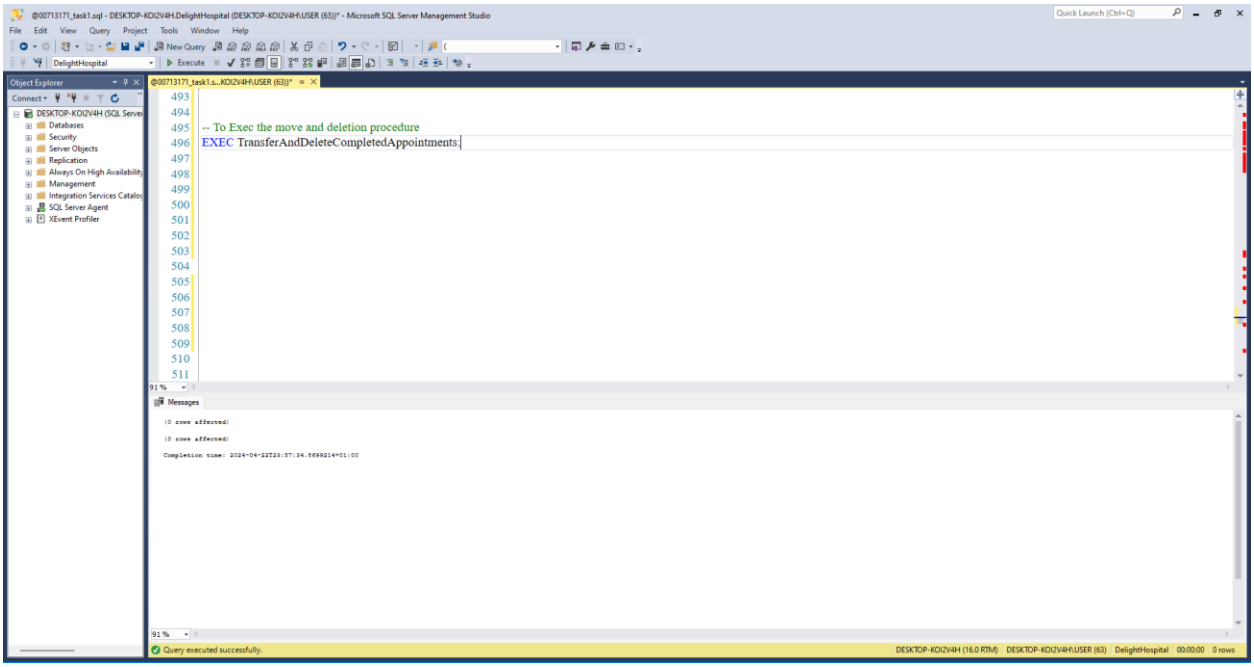



**Figure 1.5.8:** Above is the TSQL **EXEC** command query that runs the '**TransferAndDeleteCompletedAppointment**' stored procedure although no completed appointment at this time due and date in future.

## Q5. **Creating View**

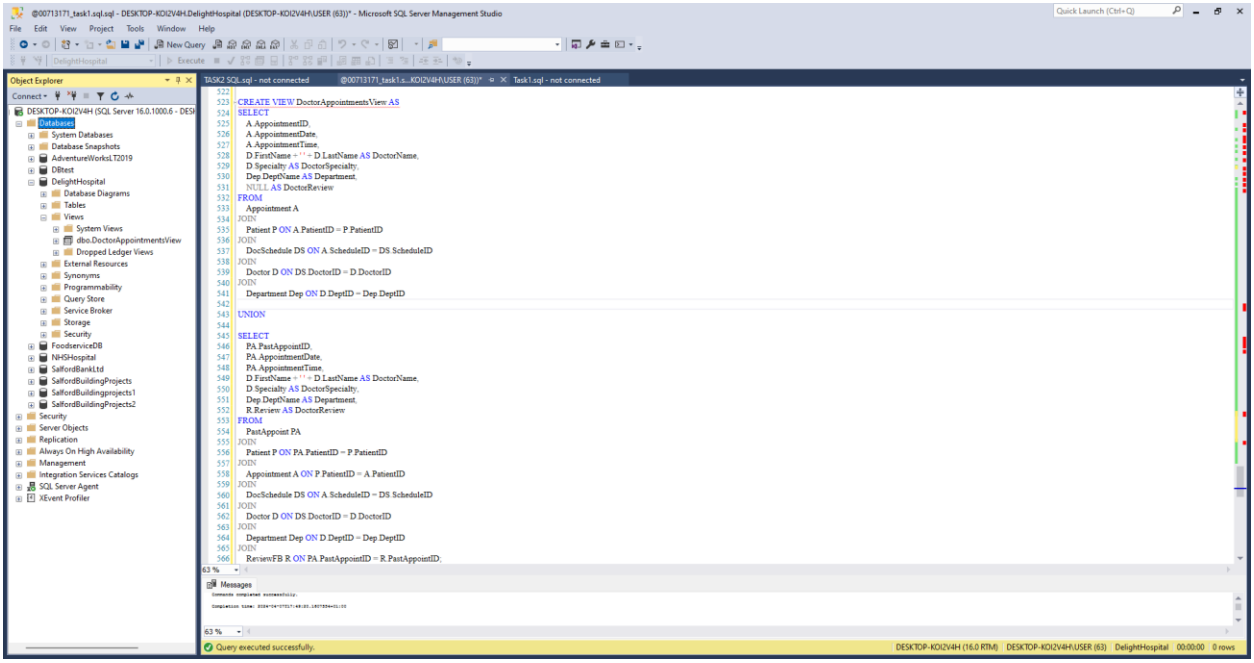



Figure 1.5.9: In this view query I've used the UNION operator to combine results of two (2) SELECT statements to become one result set. First query retrieve details of Appointment table while the second returns record of Pastappointment, they select similar query only for the second query that includes review/feedback for doctors from past appointments. I also ensured datatypes are the same to avoid error, finally I aliased NULL value for Doctor review on the first select statement to align the structure of the result sets for the UNION operation with the second select statement.

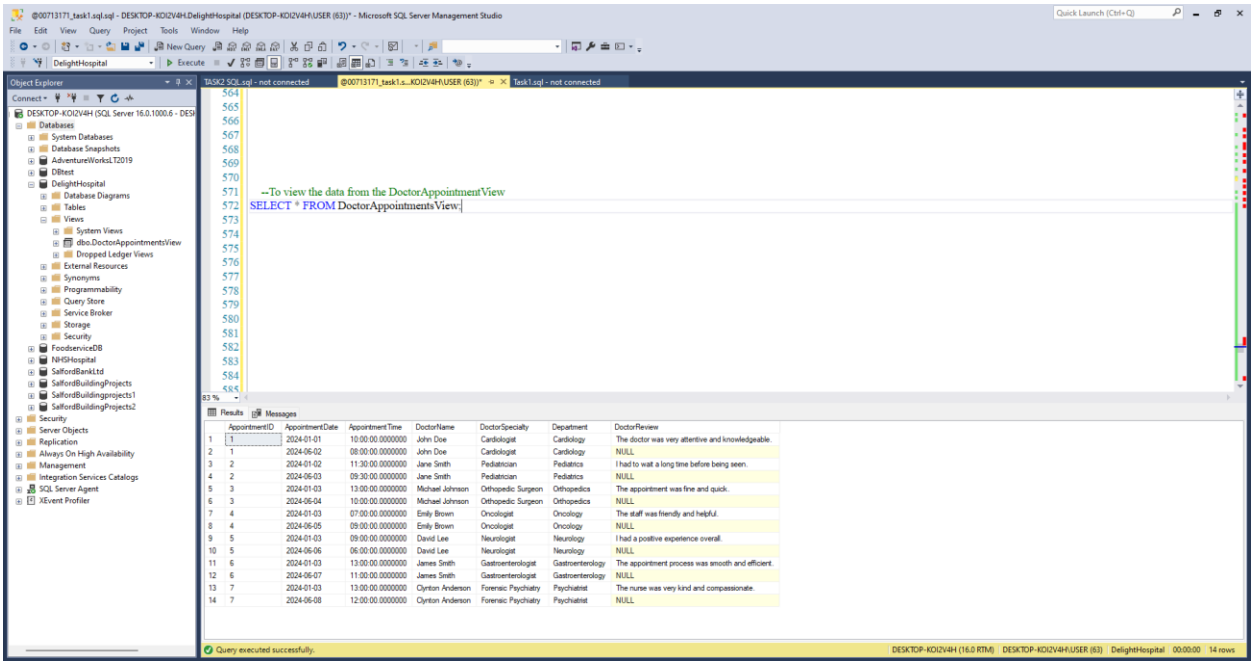



**Figure 1.6.0:** Above is the TSQL SELECT statement to view the stored View table with name "**DoctorAppointmentView**".

**Q**6. Creating a trigger so that the current state of an appointment can be changed to available when it is cancelled:

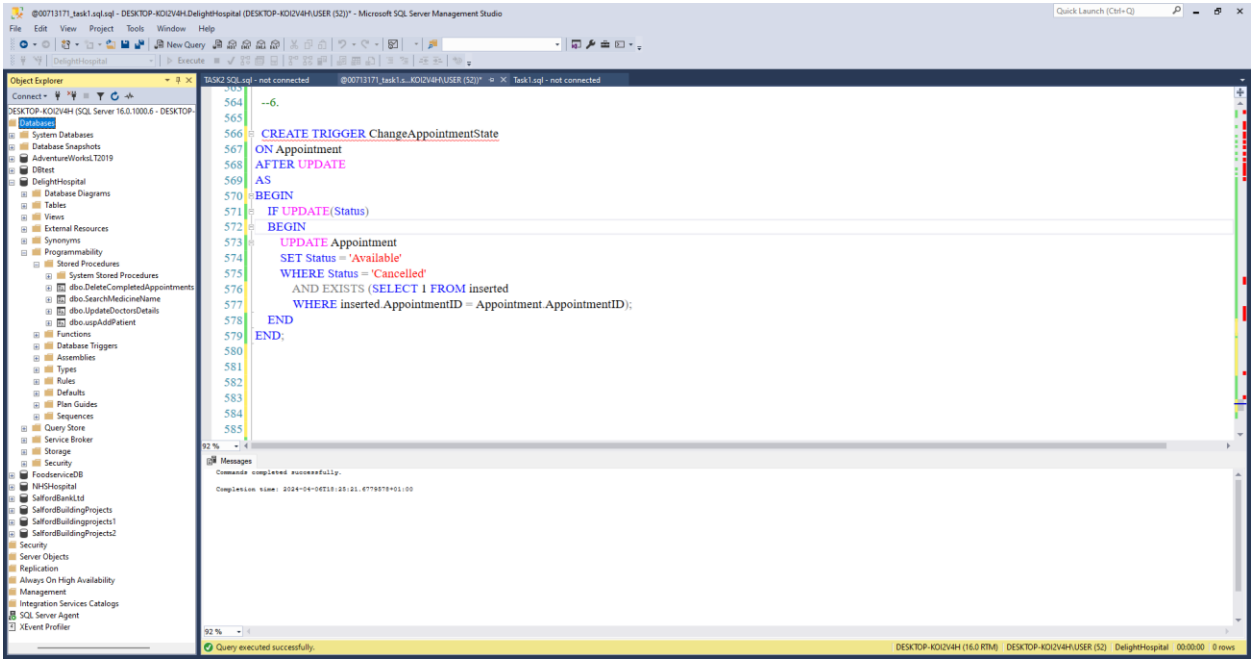



**Figure 1.6.1:** The TSQL Trigger above checks if the status column has been updated to "Cancelled" and then automatically updates the states of the cancelled appointment to "Available". It guards against inadvertent adjustments to other appointments and guarantees that only the appointments that are being updated are impacted.

**Q**7. Write a query that allows the hospital to identify the number of completed appointments with the specialty of doctors as '**Gastroenterologist'.**

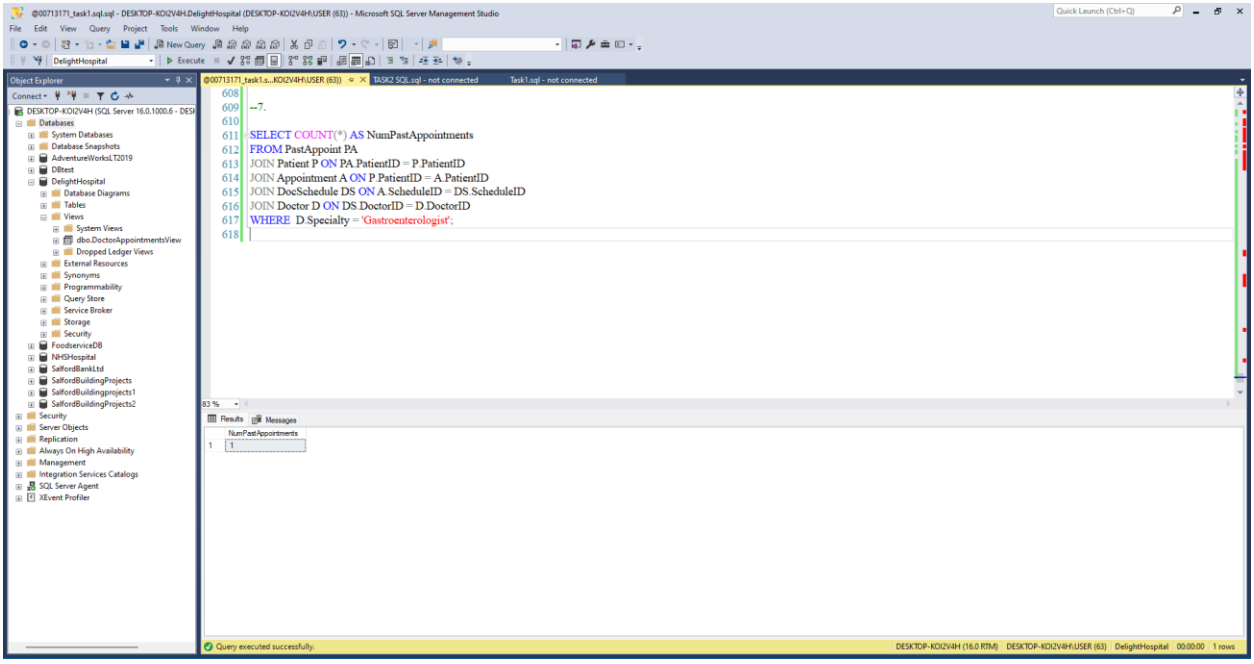



**Figure 1.6.2:** This SQL code connects tables like PastAppoint, Patient, Appointment, DocSchedule, and Doctor to track previous appointments for patients with gastrointestinal specialists.

**Q**8.

**EXPLAINING WHAT I HAVE CONSIDERED IN YOUR DATABASE FOR:**

**DATA INTEGRITY AND CONCURRENCY**

Primary and Foreign keys: In my tables I've established connections between data points, preserving referential integrity by ensuring distinct identity for each record and persistent linking of related material

I've established check constraints to ensure data consistency and cleanliness, such as correct email format and secure passwords, to maintain a clean and secure database. Be committed to continuous surveillance, examining and verifying the data to identify potential issues promptly and ensuring the reliability and trustworthiness of information's.

**DATA SECURITY**

The importance of using object permissions and database schemas to enhance the security of **DelightHospital** database;

Using the Database schema to organize and manage object like tables, stored procedures, functions, and views, streamlining administration and ownership. They also allow for granular permission control, granting permissions to all objects within the schema, simplifying security configuration and reducing administration overhead.

"View-level security" regulates and limits access for their roles. Combining table-level and view-level security can improve security without compromising data usefulness, thereby preserving **Patient** privacy. Some and more Stored procedure can be utilized to restrict access to data while maintaining security by enclosing business logic within these procedures, thereby reducing the risk of unwanted access.

Finaly to protect against SQL injection attacks, its crucial to implement string escaping techniques and built-in methods like PHP's **mysql_real_escape_string()**. Avoid keeping Java/JSP scripts accessible over HTTP, store configuration files in secure locations, and limit database access to trusted clients at the port level or through database capabilities.

## DATABASE BACKUP AND RECOVERY

Steps In Performing a Full Database Backup Using SSMS

In providing my client with advice and guidance on backup and recovery I've shown him steps in carrying out the process using SSMS

1. I've created a folder called **DelightHospital_Backup** in the **'C:'** drive, then I opened my SSMS and connected to an instance of SQL Server then expanded the server tree and then expand the database folder as seen in **Figure 1.6.3** below

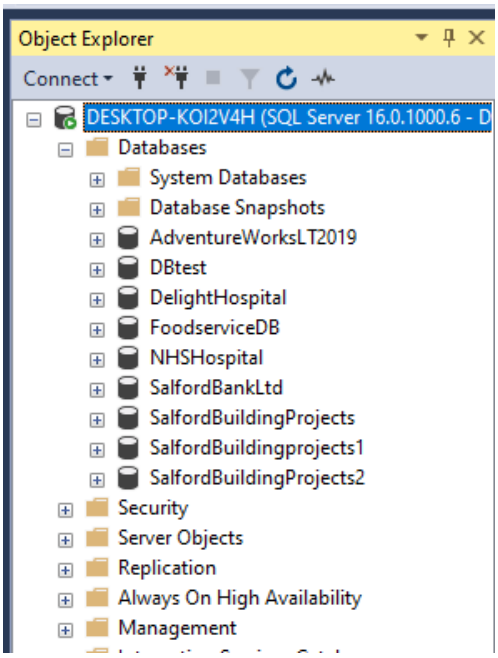



**Figure 1.6.3**

2. I right clicked the DelightHospital database and selected Tasks and then Backup as seen in **Figure 1.6.4** below

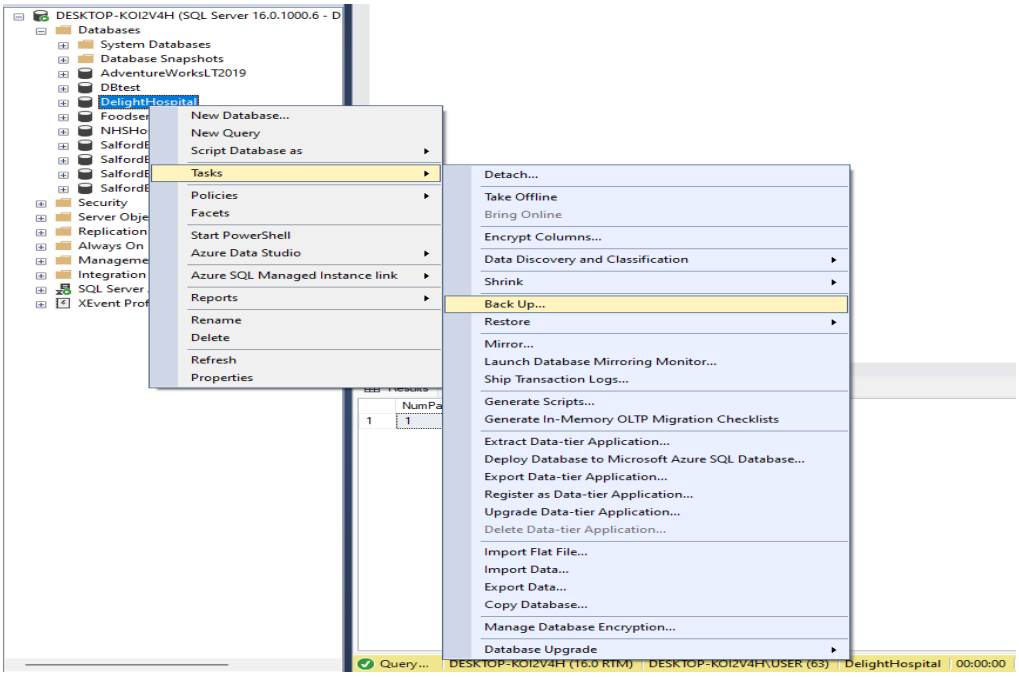



**Figure 1.6.4:**

3. I've left my backup type as full also confirmed that I'm on the right database and then I selected Disk instead of URL, I clicked remove to replace the automatically included location and then added my backup location as shown below in **Figure 1.6.5**:

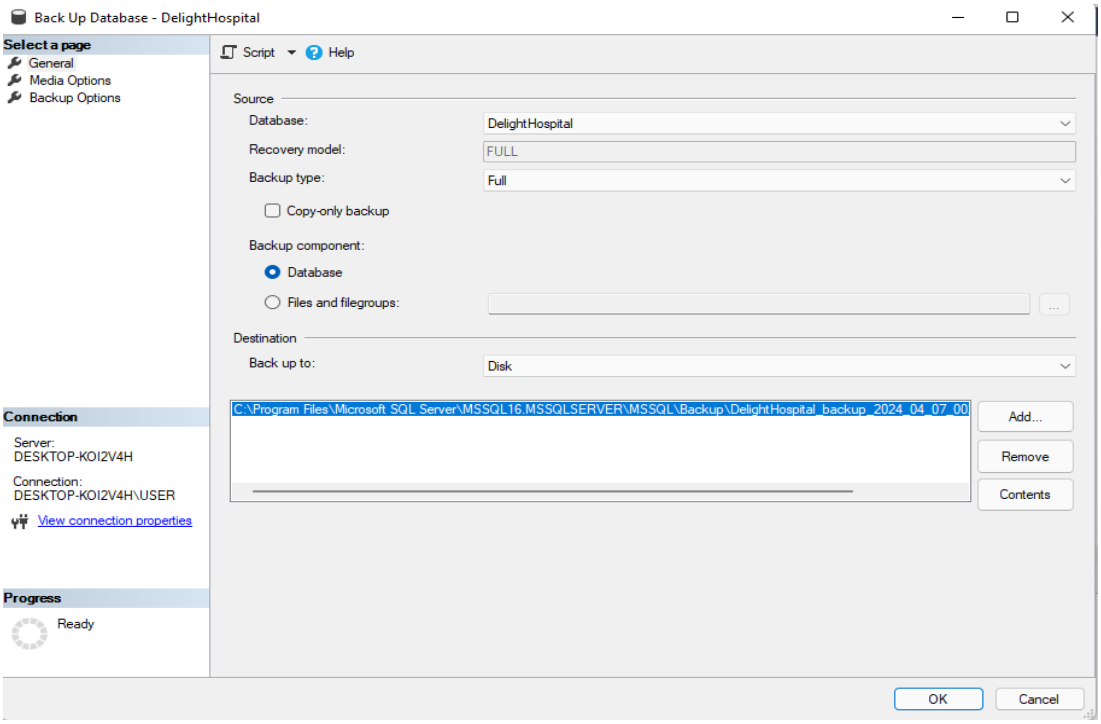



**Figure 1.6.5**

4. The dialog box below called “Select Backup Destination” shows when I click ok, now on this box I clicked the ellipsis (…) after selecting File name as shown below in **Figure 1.6.6**:

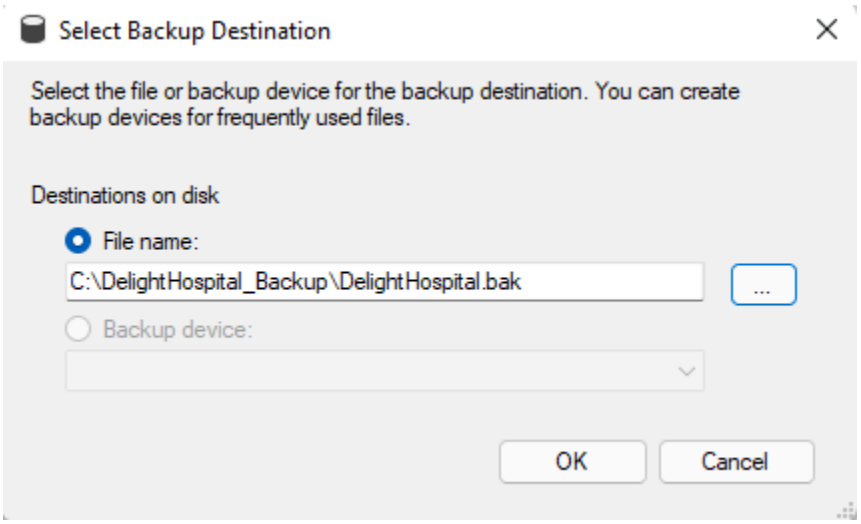



**Figure 1.6.6**

5. In the Locate Database Files dialog box, I located my backup file and then for the File name below I typed **DelightHospital.bak** and then clicked ok twice as seen in **Figure 1.6.7**.

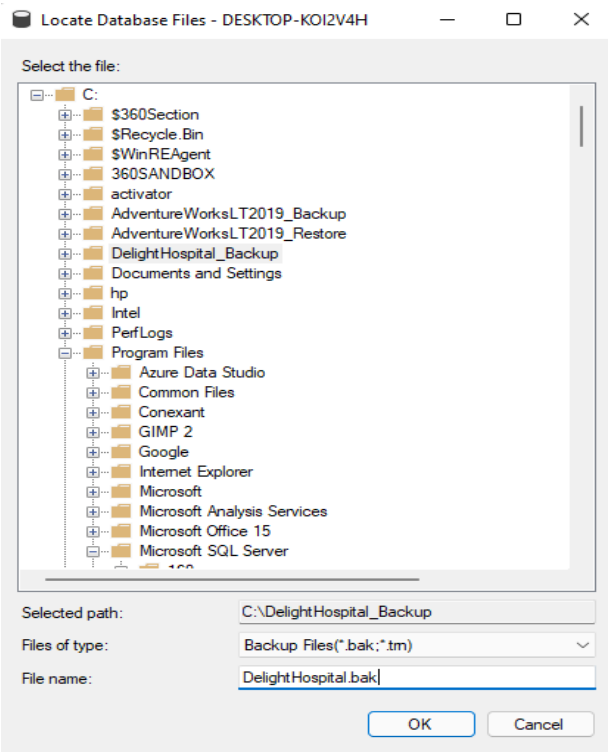



**Figure 1.6.7**

6. In the Select a Page pane of the Back Up Database dialogue box, I selected **Media Options.** In the first section I picked **Overwrite All Existing Backup Sets option**. I left the reliability section as seen in **Figure 1.6.8** Below.

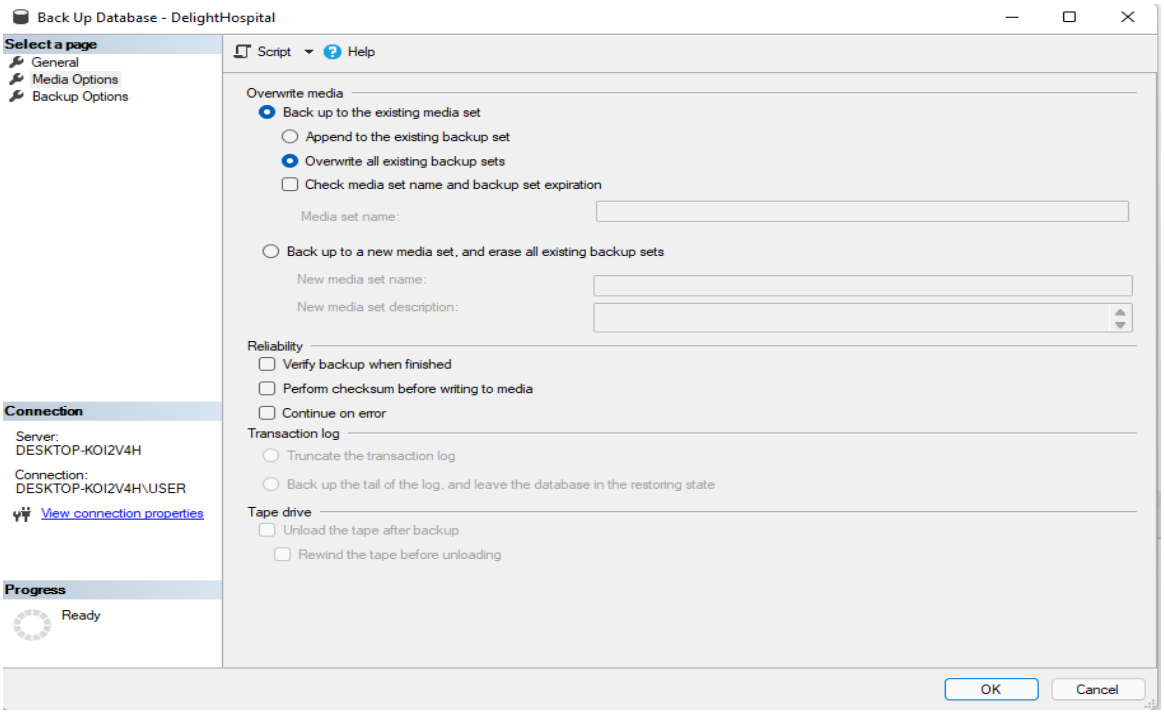

Back Up Database - DelightHospital
Select a page
General
Media Options
Backup Options
Script
Help
Overwrite media
Back up to the existing media set
Append to the existing backup set
Overwrite all existing backup sets
Check media set name and backup set expiration
Media set name:
Back up to a new media set, and erase all existing backup sets
New media set name:
New media set description:
Reliability
Verify backup when finished
Perform checksum before writing to media
Continue on error
Transaction log
Truncate the transaction log
Back up the tail of the log, and leave the database in the restoring state
Tape drive
Unload the tape after backup
Rewind the tape before unloading
Connection
Server:
DESKTOP-KOI2V4H
Connection:
DESKTOP-KOI2V4H\USER
View connection properties
Progress
Ready
OK
Cancel

**Figure 1.6.8**

7. In this stage I selected ‘compress backup’ as seen in **Figure 1.6.9** Below.

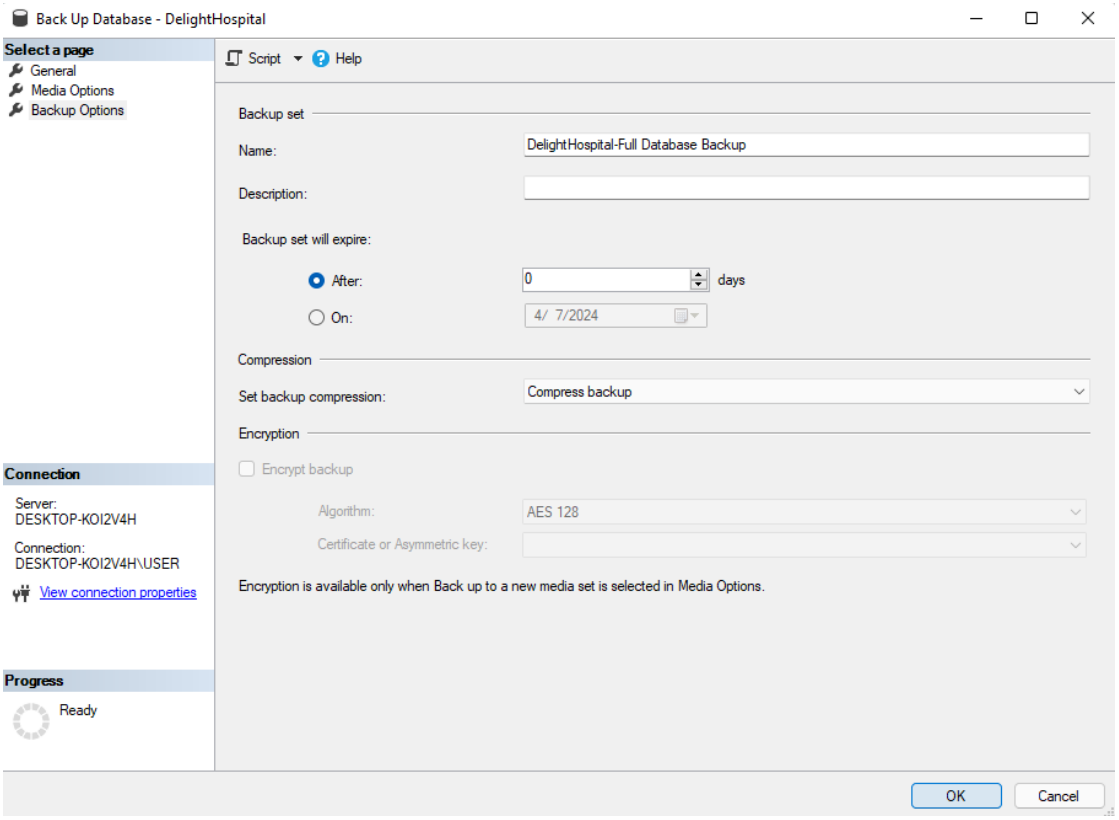



**Figure 1.6.9**

8. MY back Up is complete after the step above and then I can find my back up in the folder location as seen in **Figure 1.7.0** Below.

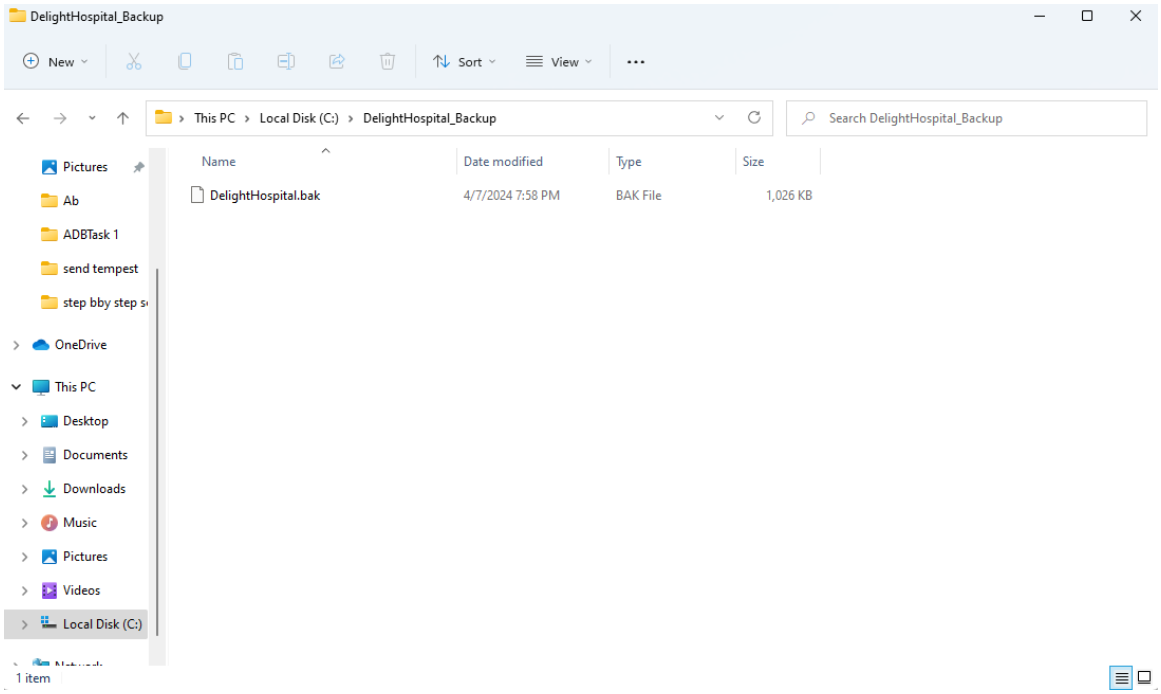

DelightHospital_Backup
New
Sort
View
This PC › Local Disk (C:) › DelightHospital_Backup
Search DelightHospital_Backup
Pictures
Ab
ADBTask 1
send tempest
step bby step s
OneDrive
This PC
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
Name
Date modified
Type
Size
DelightHospital.bak
4/7/2024 7:58 PM
BAK File
1,026 KB
1 item

**Figure 1.7.0**

STEPS FOR RESTORING THE DELIGHTHOSPITAL DATABASE IS SHOWN FROM ONE SCREENSHORT TO THE OTHER;

1. Right click on the Database, Task, Restore, Database as seen in the **Figure 1.7.1**:

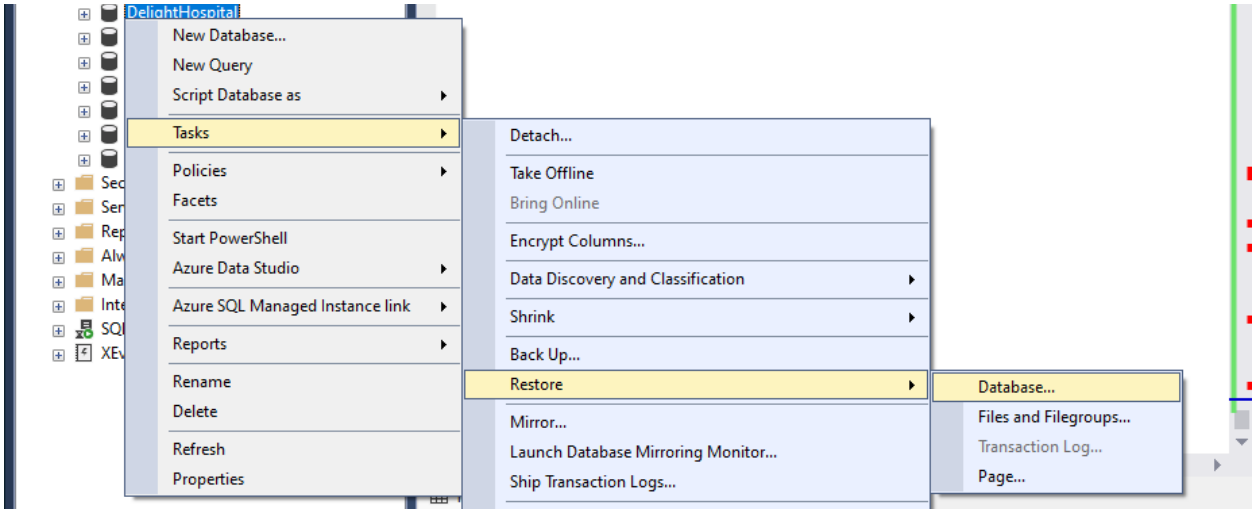



**Figure 1.7.1**

Select Device option, click on the "**…**" button, as seen in the **Figure 7.2** below:

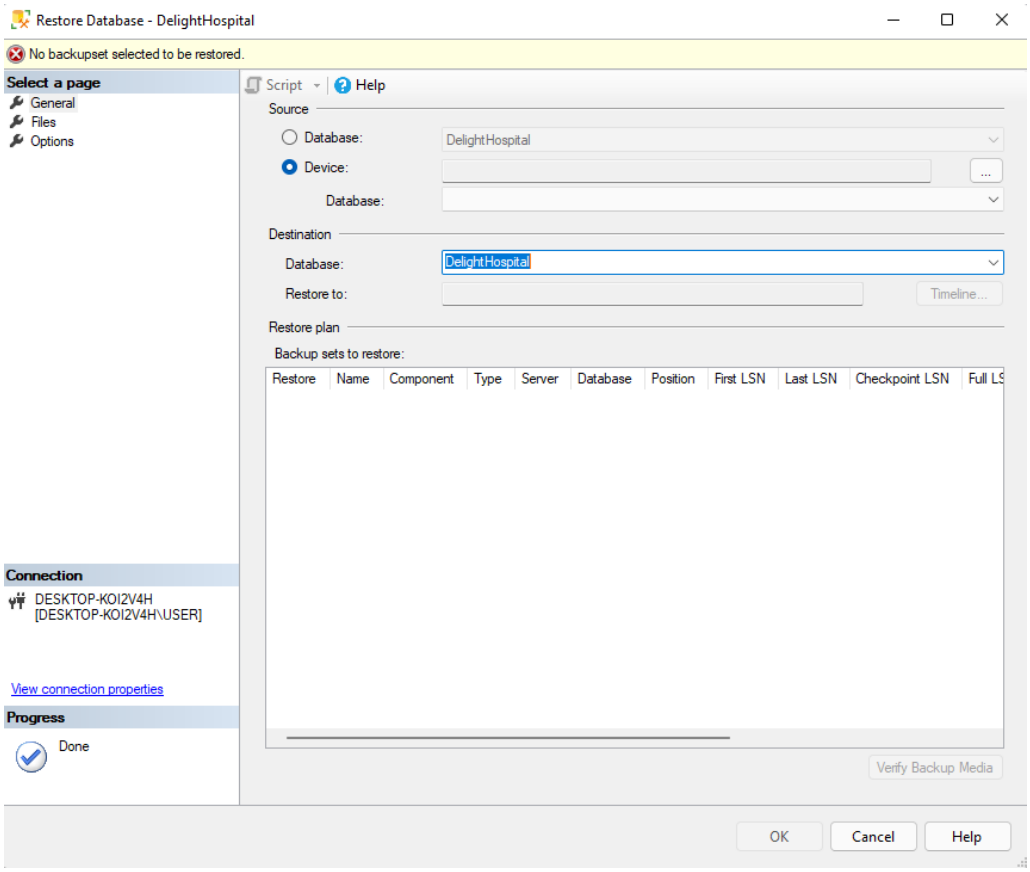



**Figure 1.7.2**

2. **Select backup devices** dialog box option then select File for Backup media type as seen in **Figure 7.3** Below. Click on the **ADD** button.

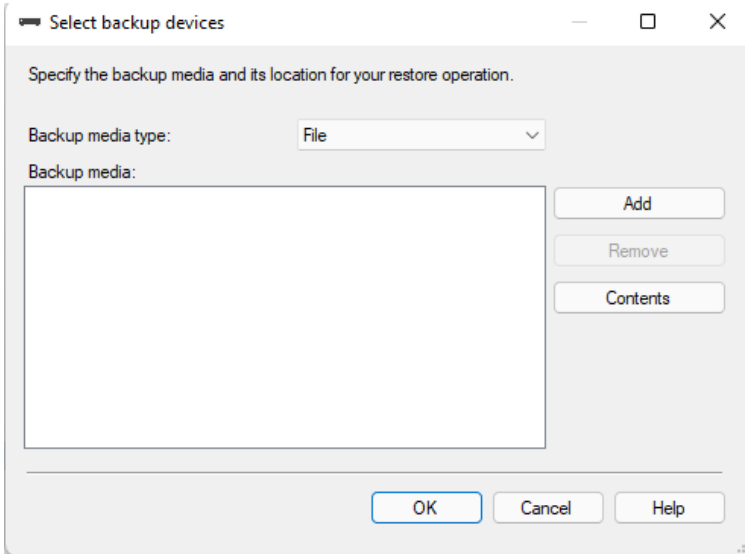



**Figure 1.7.3**

3. Select the **DelightHospital.bak** file from Local drive as shown in the diagram below

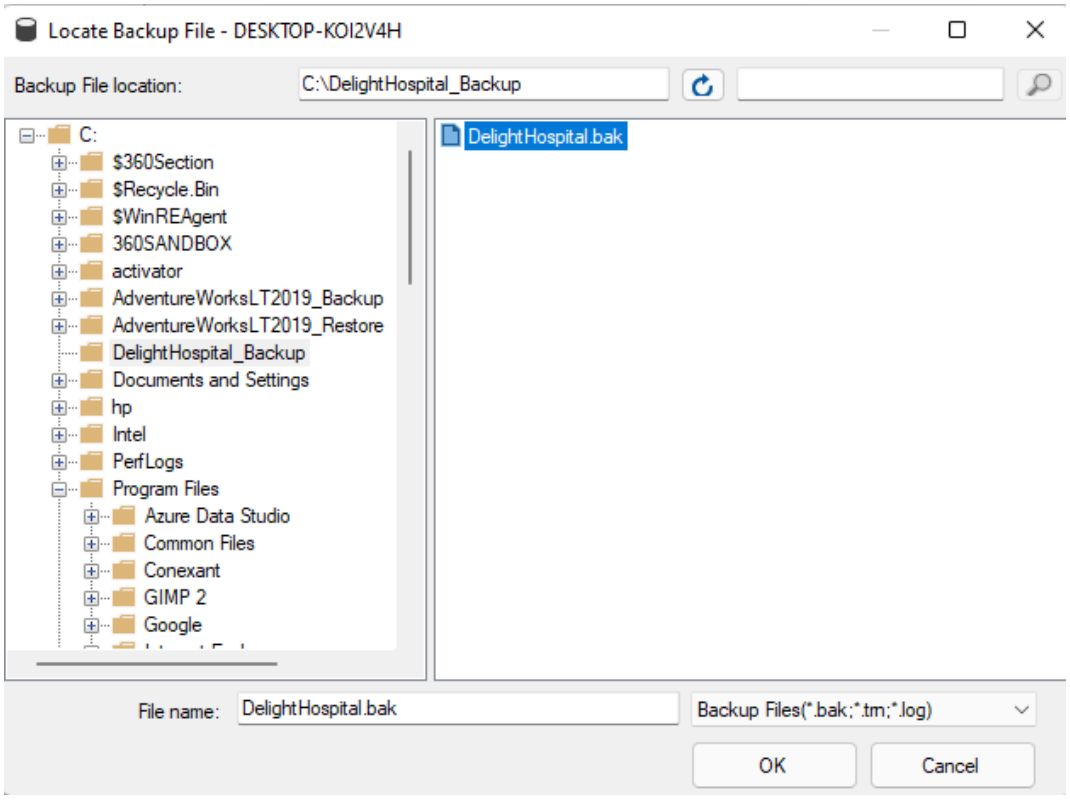



**Figure 1.7.4**

4. Select the **Close Existing Connections to Destination Database** option in the Server Connections section. Click **OK**

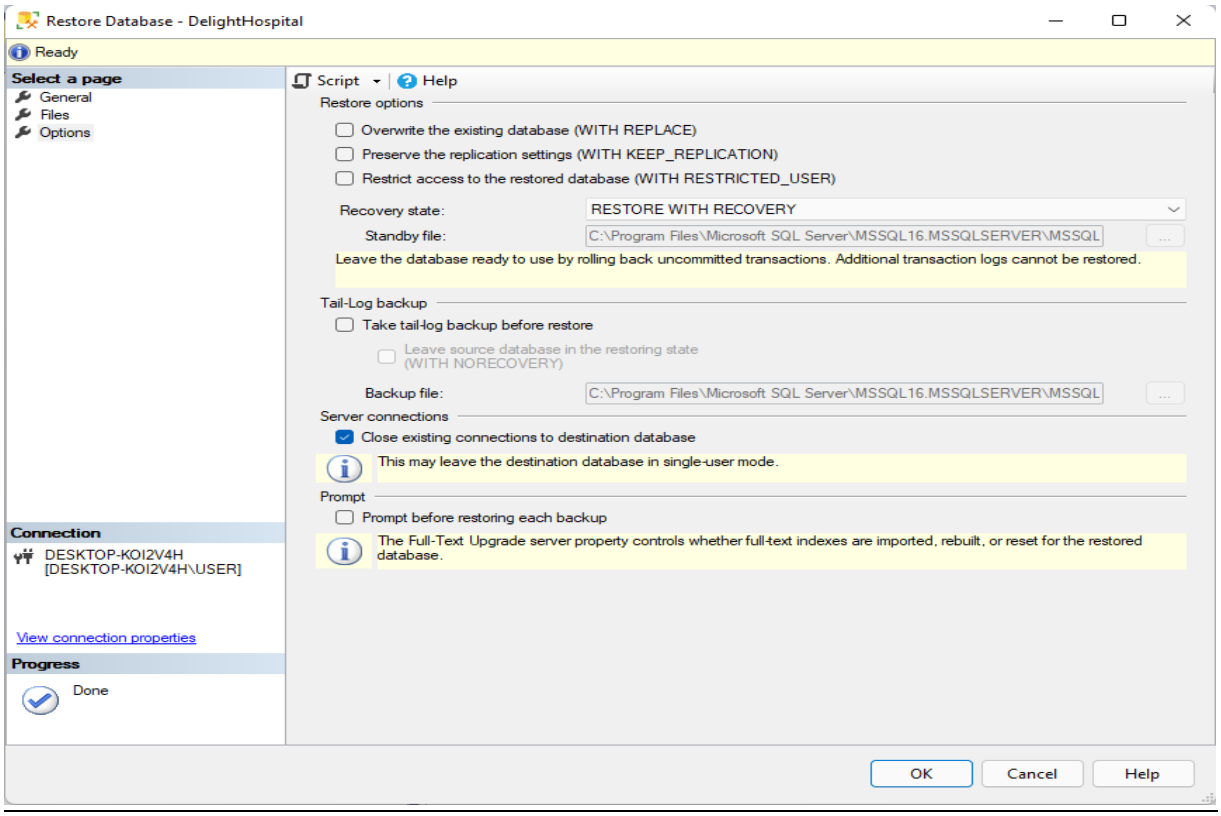



**Figure 1.7.5**

5. Click **OK**

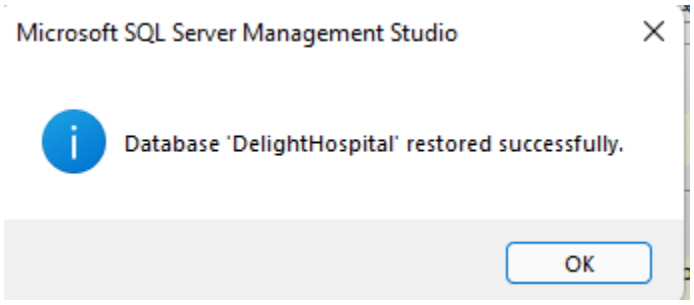



**Figure 1.7.6**

# CONCLUSION

The **DelightHospital** database is a significant advancement in data integrity, security protocols, and operational optimization. It efficiently manages patient, appointment, medical record, and staff information while maintaining confidentiality. The database structure includes tables with defined properties, ensuring quality and dependability of information through primary keys, foreign keys, and constraints.

The database is secure through strict procedures, including the use of hashing techniques to store passwords and improve access control and confidentiality. Database schemas, object permissions, and views ensure users have access to only the necessary data for their roles. The database's resilience is ensured through strong backup and recovery protocols, with regular testing to ensure data regular in case of crises or system failures.

# TASK TWO REPORT (2)

## INTRODUCTION

The job requires me to serve as a database consultant for a food service business. Construction of a database and data import from four related CSV files into separate tables are the goals. Restaurant cuisines, ratings, customers, and other information are all included in the provided CSV files.

I'm going to accomplish this goal by doing the following steps:

- A brand-new database called "FoodserviceDB" will be made using SQL.
- The contents of the tables are imported from the CSV files unchanged.
- All tables have the necessary primary keys added to them. However, referential integrity is protected by properly defining and maintaining foreign keys.
- A database diagram is made to illustrate the relationships between the tables in order to provide a clear image of the database structure.

The database will be properly created, the necessary data will be imported, and the tables will be correctly linked and ready for further analysis according to the preciously discussed methods.

## PART 1:

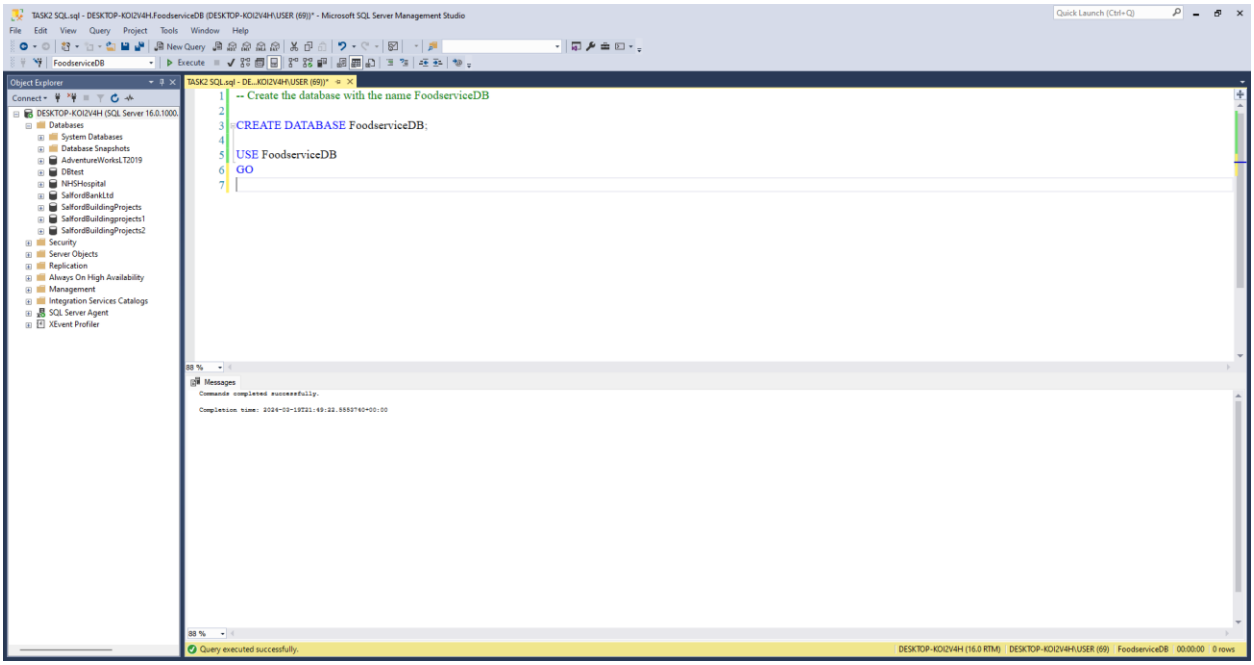



**Figure 2.1.0:** In this first step I have written a TSQL code to create a database with name **FoodServiceDB** and with the 'USE' query to change from the current database to the just created database.

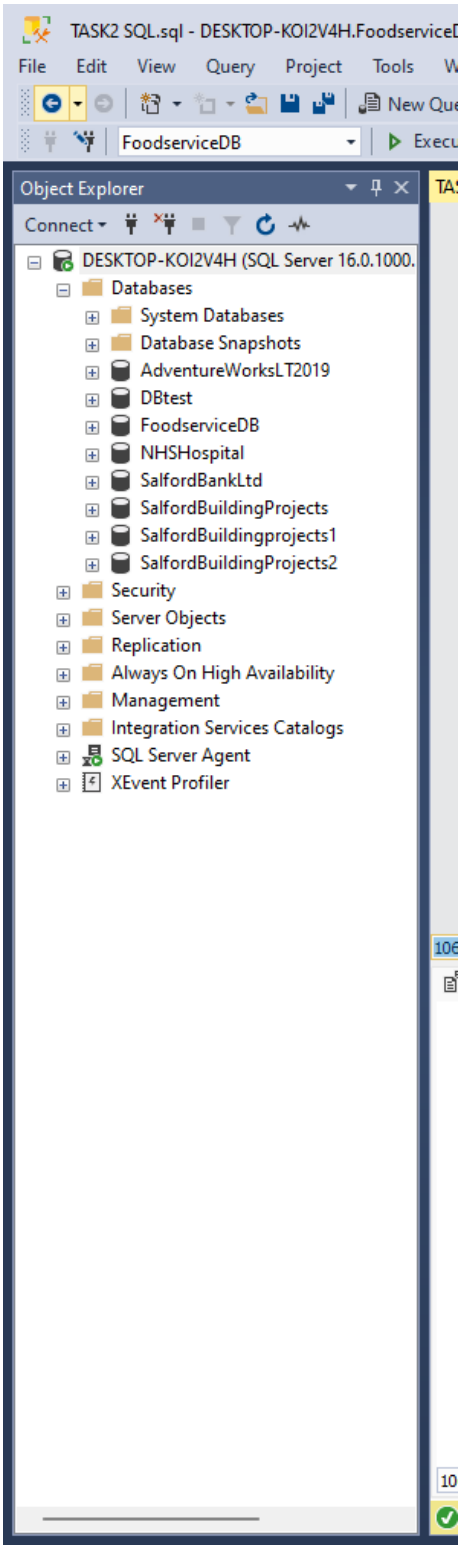



**Figure 2.1.1:** This figure shows us the successful creation of the FoodServiceDB database.

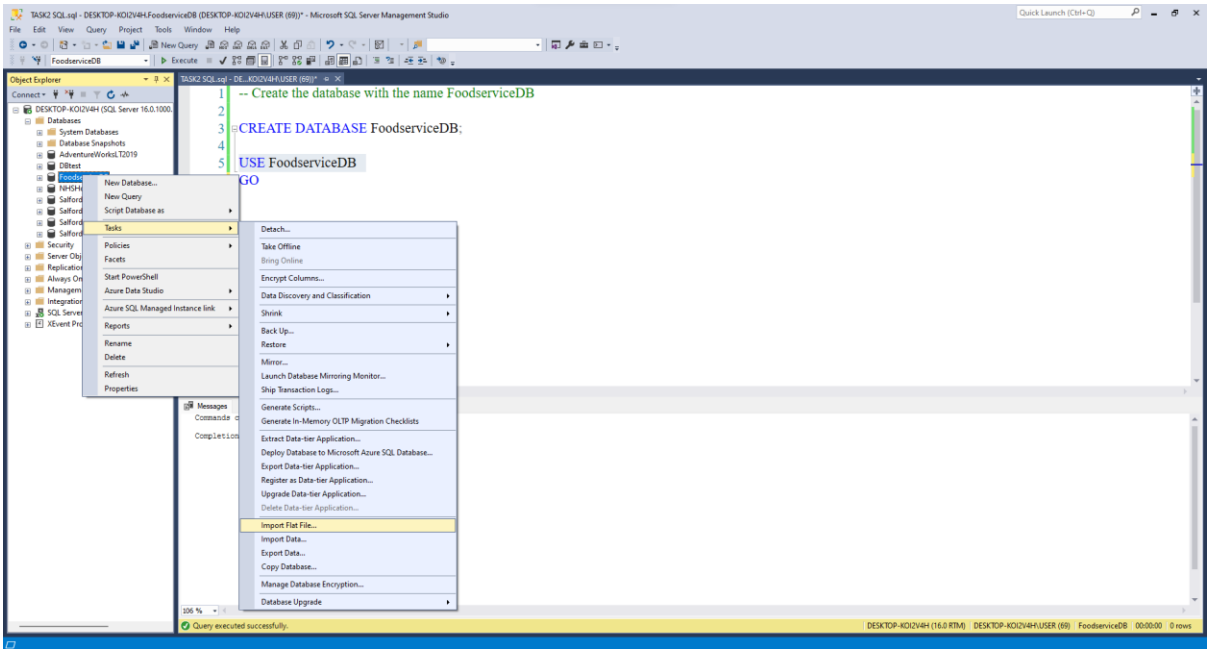



**Figure 2.1.2:** Step by step procedure to move/import files into the FoodServiceDB database

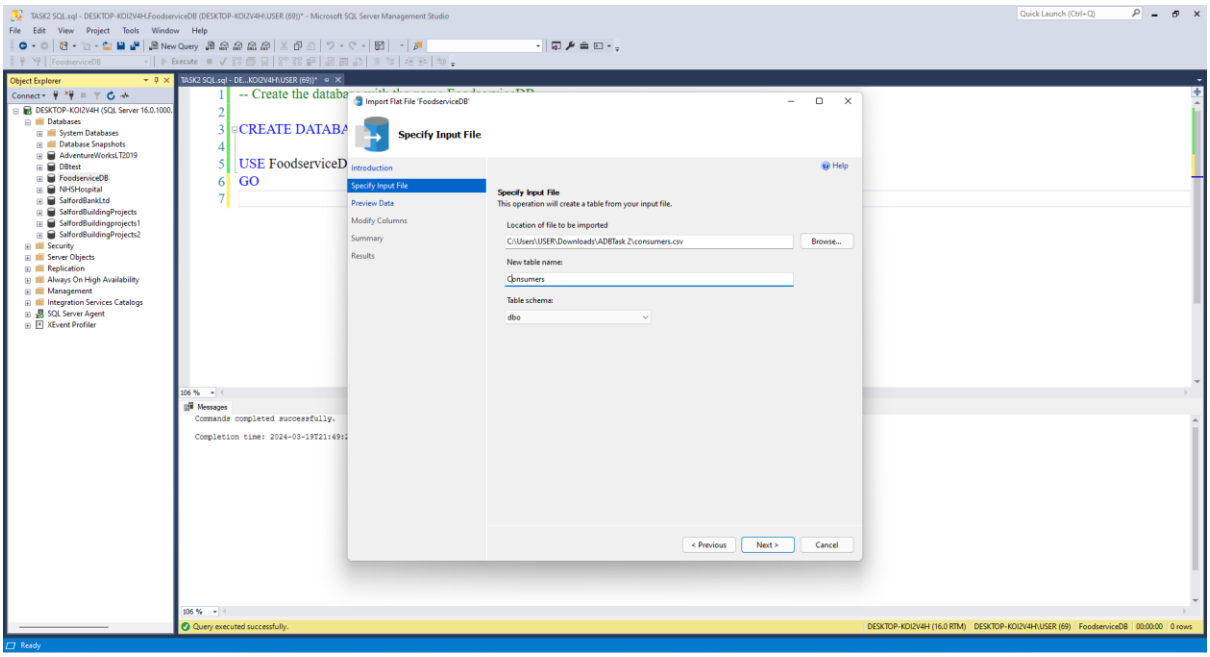



**Figure 2.1.3:** In this step while importing the consumers.csv file I rewrote the table name as requested in the assessment.

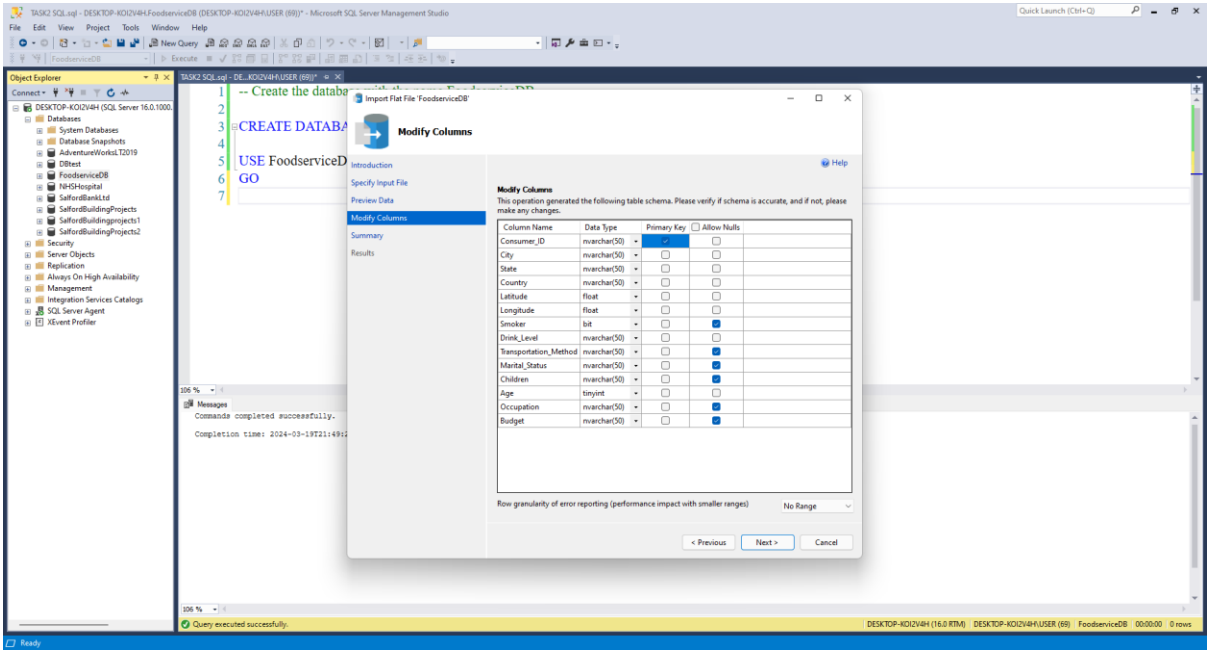



**Figure 2.1.4:** I imported the Consumers file into a table and selected the Consumer_ID as primary key

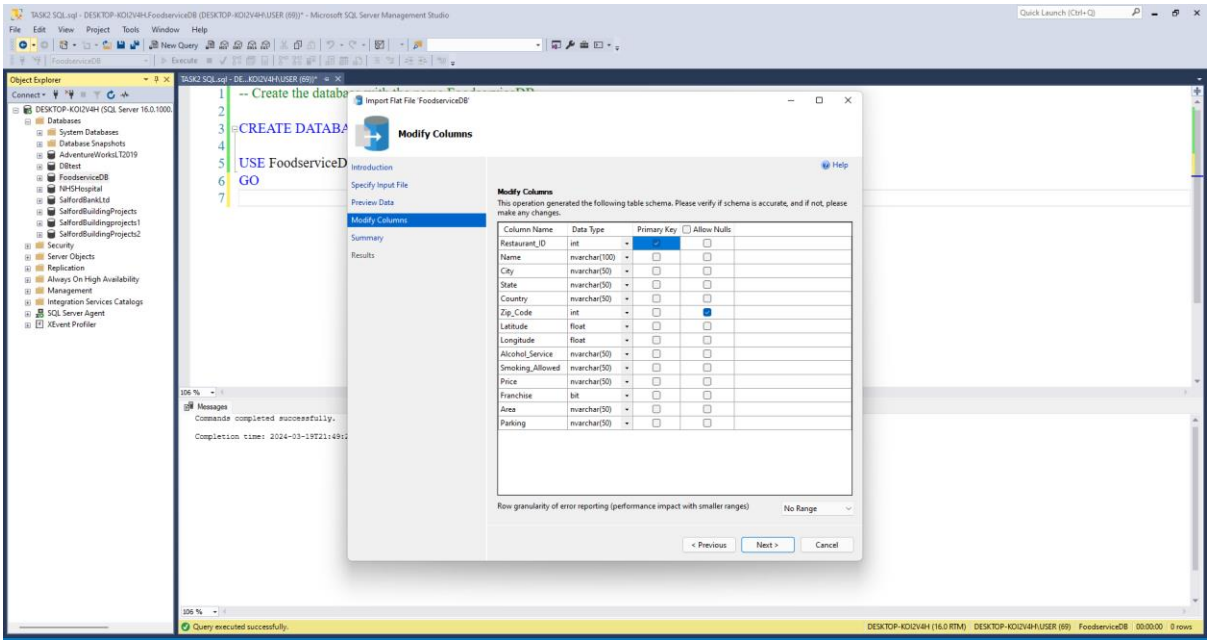



**Figure 2.1.5** I imported the restaurant file into a table and as done in **Figure 2.1.3**, I rewrote the table name and then selected the Restaurant_ID as primary key.

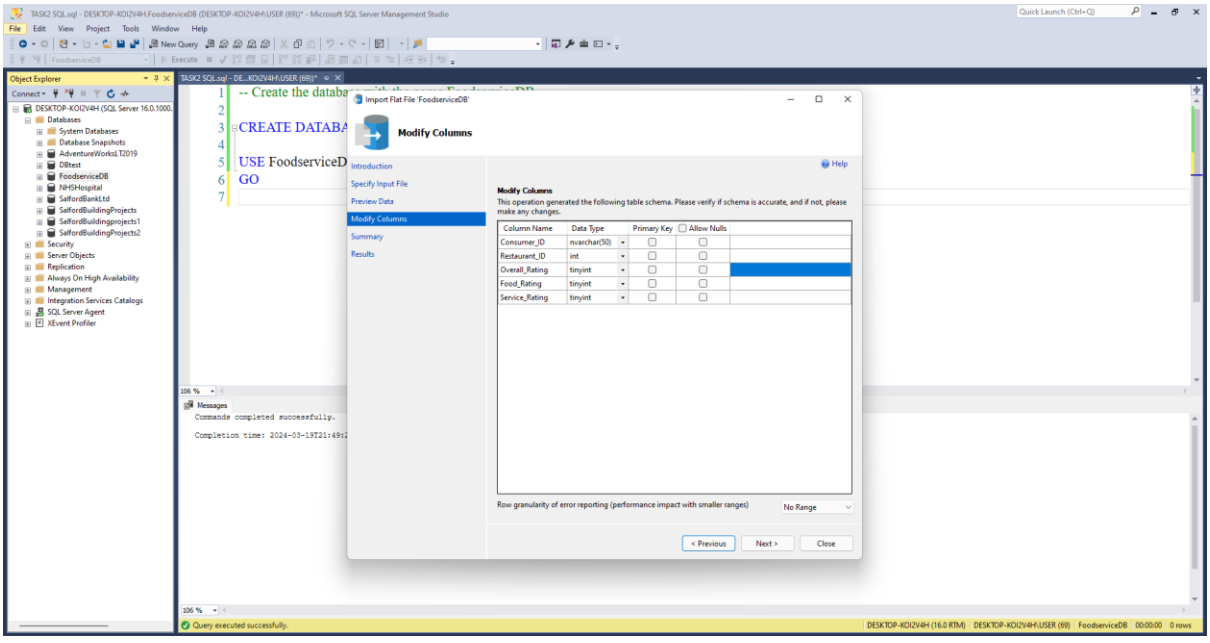



**Figure 2.1.6** I imported the ratings file into a table and as done in **Figure 2.1.3**, I rewrote the table name to Ratings.

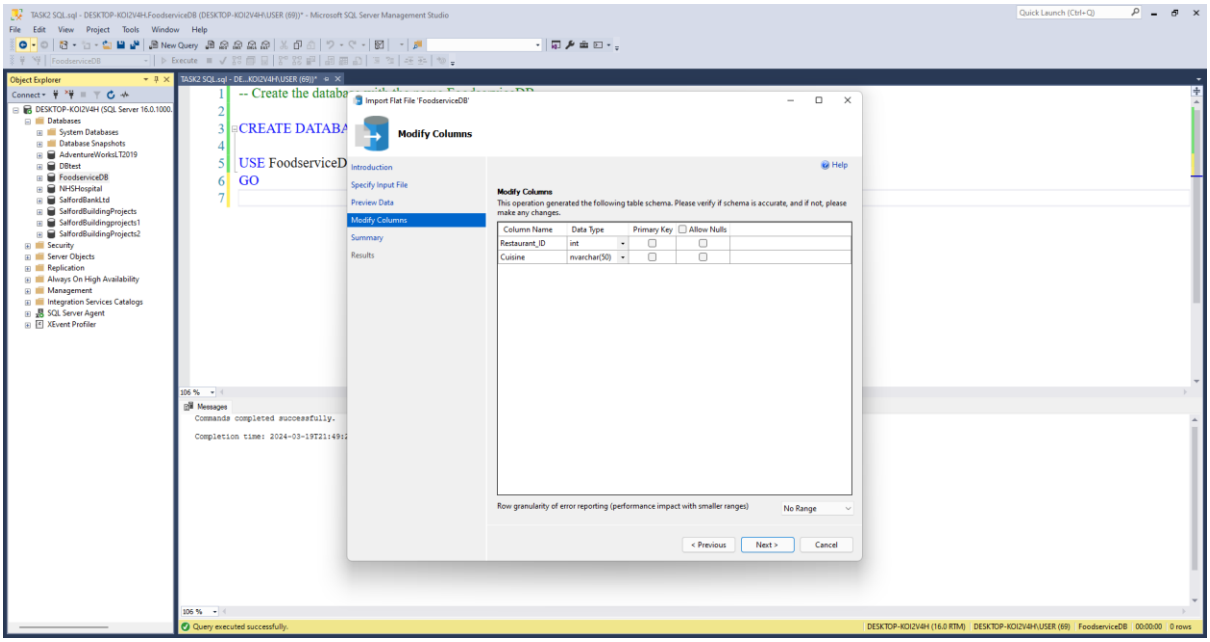



**Figure 2.1.7** I imported the restaurant_cuisines file into a table and as done in **Figure 2.1.3**, I rewrote the table name to Restaurant_Cuisines.

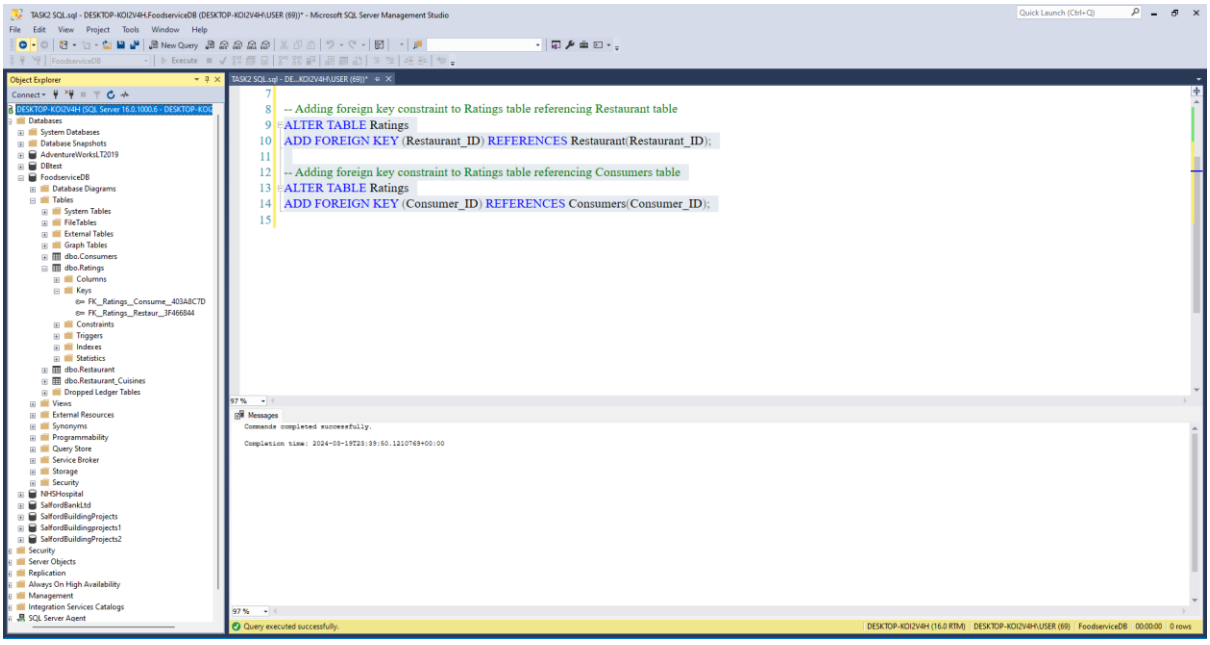



**Figure 2.1.8** Adding the foreign key constraint for the Ratings table

**Explain in details clement**

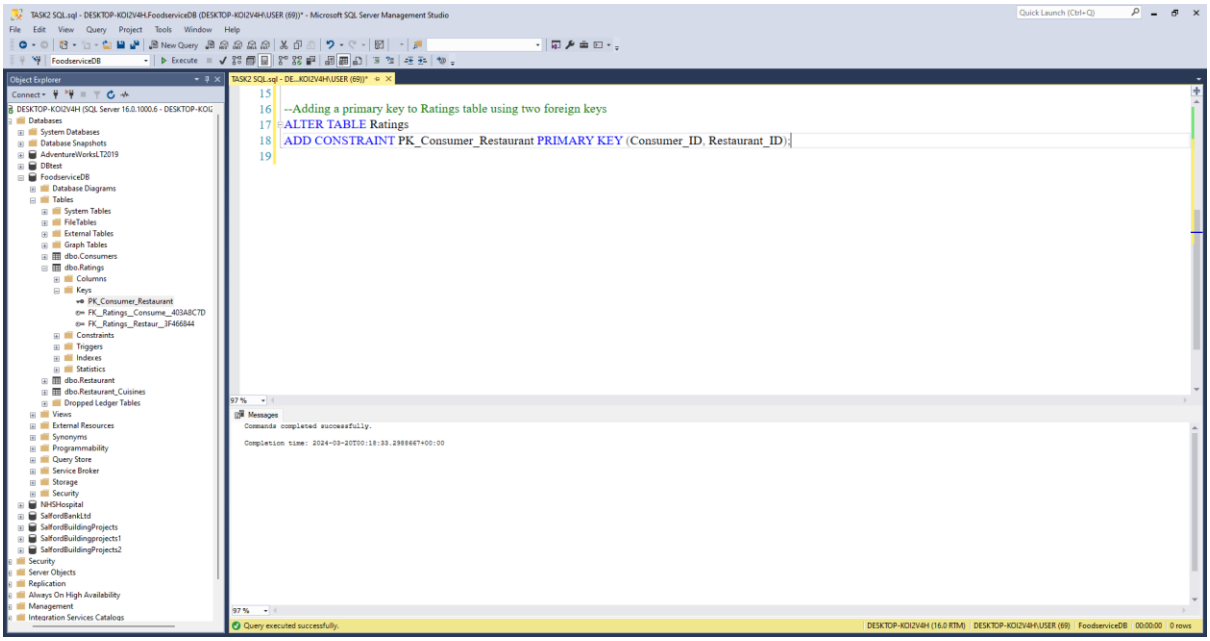



**Figure 2.1.9:** The purpose of the **ALTER** query here is to use the Consumer_ID and Restaurant_ID columns as composite keys to add a primary key constraint to the Ratings table. According to this method, every combination of Restaurant_ID and Consumer_ID ought to identify a record in the Ratings table in a distinct way.

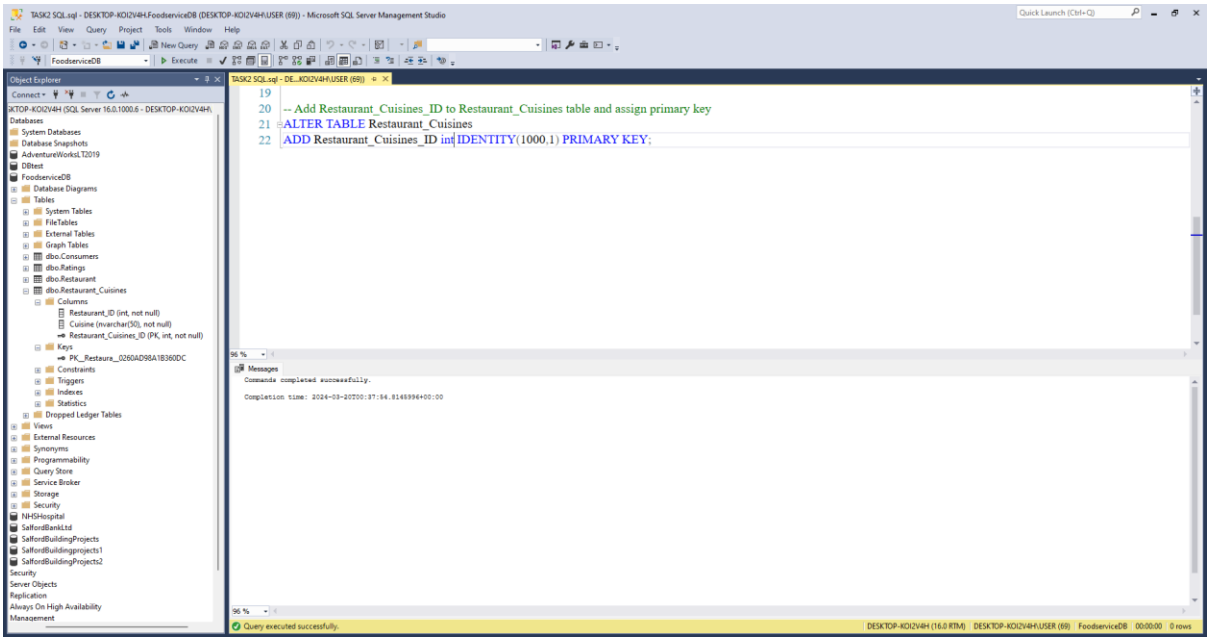


**Figure 2.2.0:** Adding a new column (Restaurant_Cuisines_ID) with a primary key into the Restaurant_Cuisines table

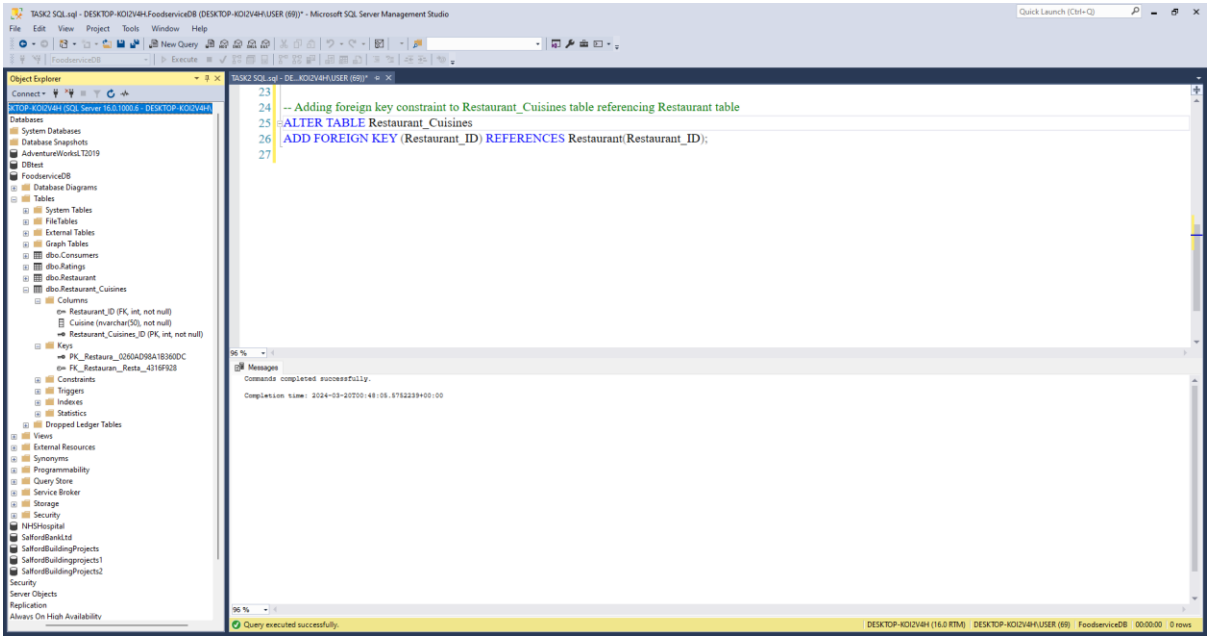



**Figure 2.2.1:** Adding foreign key constraint for the Restaurant_Cuisines table to reference the Restaurant table.

## Database Diagram

All tables contained in this database have been adequately joined with the required constraints such as foreign keys which has been used to reference or link together related tables so as to query accurate result. Below is the database diagram **Figure 2.2.2**, showing the visual representation of each table and their relation.

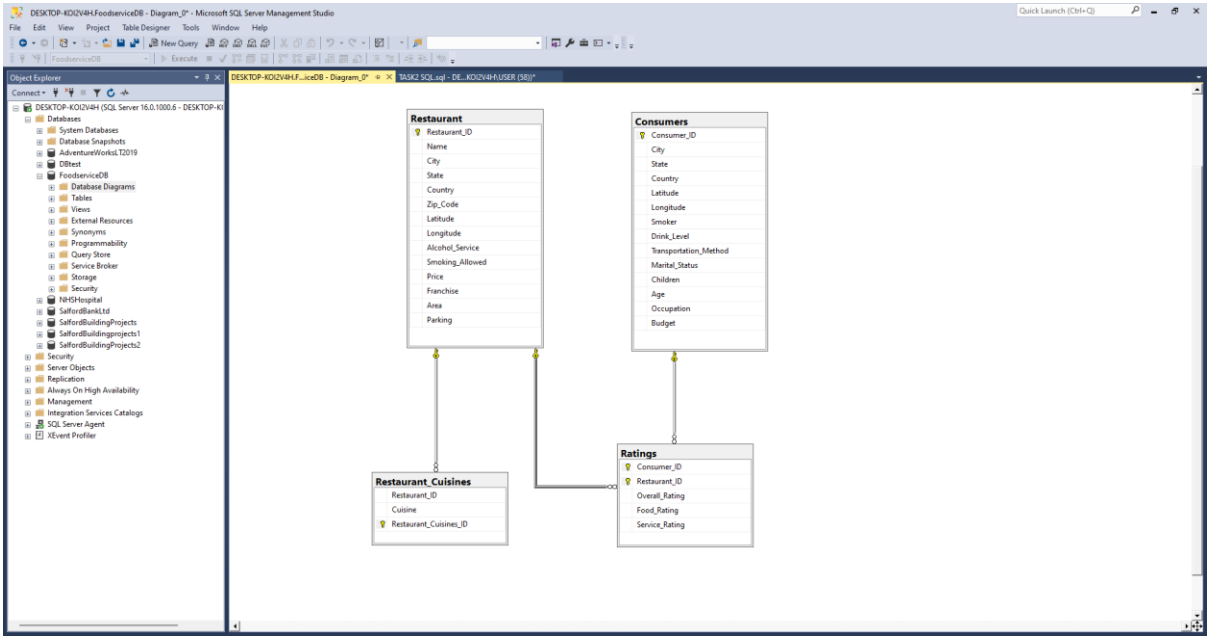



**Figure 2.2.2:** The Microsoft SQL Server Management Studio database diagram above shows the tables, their names, primary keys and the relationships between them.

Each table has a unique identifier (Restaurant_id as primary key, Consumer_id as primary key for Restaurant and Consumer tables respectively, 'Restaurant_id and Consumer_id as **Foreign keys** and used to create **Composite** primary key in Ratings table, **Restaurant_Cuisines_id** as primary key to distinguish individual records within the Restaurant_cuisines table.

**Restaurant Table:**

One-to-many relationship with the "Restaurant_Cuisines" table based on Restaurant_id.

One-to-many relationship with the "Ratings" table based on Restaurant_id.

**Consumers Table:**

One-to-many relationship with the "Ratings" table based on Consumer_id.

**Ratings Table:**

Many-to-one relationship with the "Restaurant" table based on Restaurant_id.

Many-to-one relationship with the "Consumers" table based on Consumer_id.

**Restaurant_Cuisines Table:**

One-to-many relationship with the "Restaurant" table based on Restaurant_id.

## Part 2:

### Answer to Question 1.

1. In order to retrieve or list all restaurants with a medium range price, serving Mexican food in an open area, firstly I have identified the area, and price to be columns under the restaurants table while the Mexican food under the cuisine column is found in the restaurant_cuisines table. In this case the restaurant table is joined with the restaurant_cuisines table.


**Figure 2.2.3:** The TSQL to list all restaurants with a medium range price, serving Mexican food in an open area.

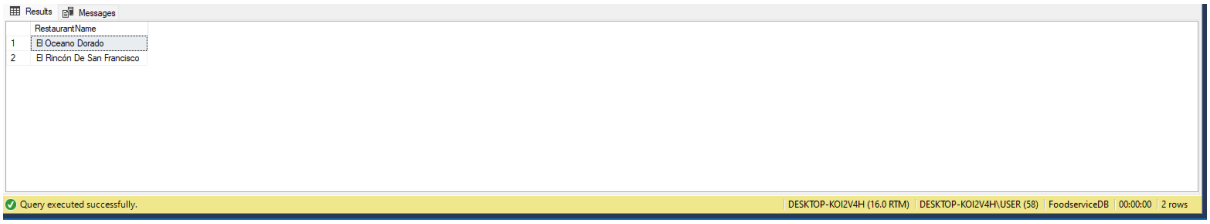



**Figure 2.2.4:** Result gotten from the query in **Figure 2.2.3**

### Answer to Question 2.

2. In Comparing the results of the total number of restaurants who have the overall rating as 1 serving Italian food with Mexican food. Using count Distinct keyword to count the number of restaurants since a restaurant can have ratings multiple times from consumers. This time the restaurants, ratings and restaurant_cuisines tables are joined together and the where clause will be used to filter the joined result set.


**Figure 2.2.5:** This TSQL query shows the total number of restaurants serving Mexican food with overall rating of 1

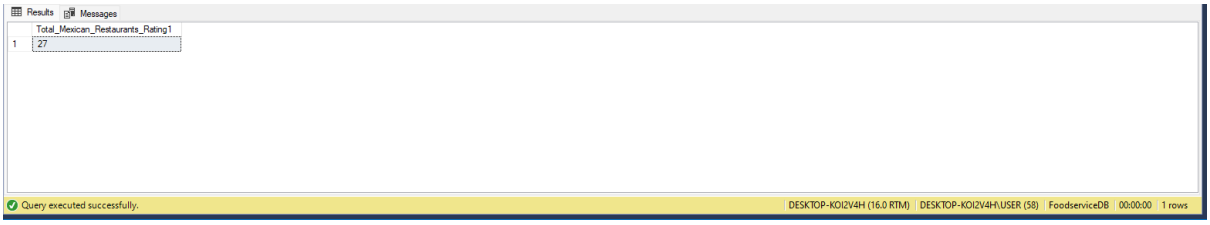



**Figure 2.2.6:** Result gotten from the query in **Figure 2.2.5**

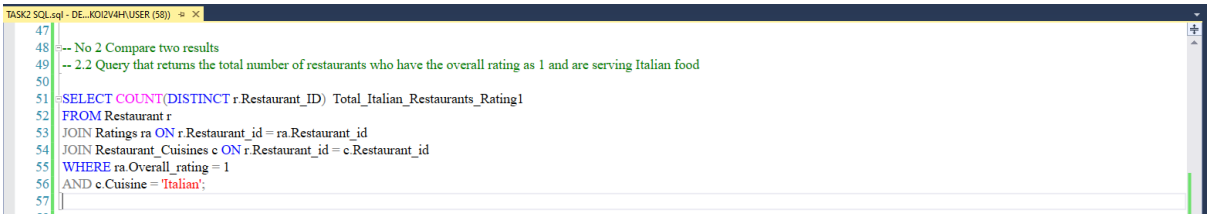



**Figure 2.2.7:** This TSQL query shows the total number of restaurants serving Mexican food with overall rating of 1

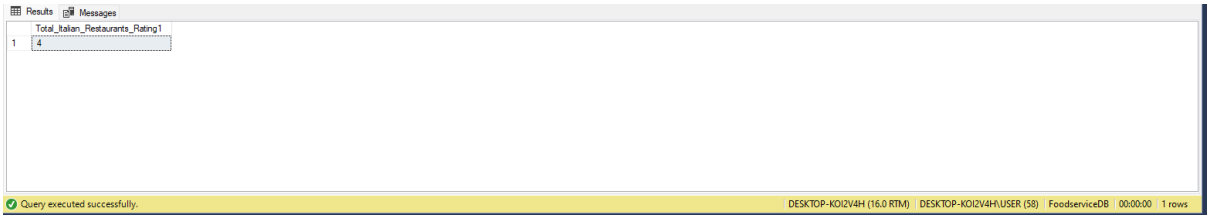



**Figure 2.2.8:** Result gotten from the query in **Figure 2.2.7**

| **Total number of restaurants** | **Overall rating of 1** |
|---|---|
| Mexican food | 27 |
| Italian food | 4 |

In restaurants with an overall rating of one, Mexican food is more common than Italian food, based on the comparison. This could indicate that there are just more Mexican eateries in the dataset or that patrons of lower-rated businesses tend to prefer Mexican food. This might be the result of varying consumer expectations, societal factors, or individual preferences. When thinking about menu options, customer satisfaction tactics, and general business decisions, restaurant owners and managers can benefit greatly from an understanding of these dynamics.

In general, the study draws attention to possible distinctions between Italian and Mexican eating experiences with regard to client satisfaction within the dataset. In order to gain a deeper comprehension of the observed discrepancies in ratings, this could spur further investigation into elements such as food quality, service standards, or the preference of a customer to a particular cuisine.

**Answer to Question 3.**

3. The average of consumers who have given a service rating of 0 is calculated by utilizing the AVG function and performing a join between the consumers and Ratings tables

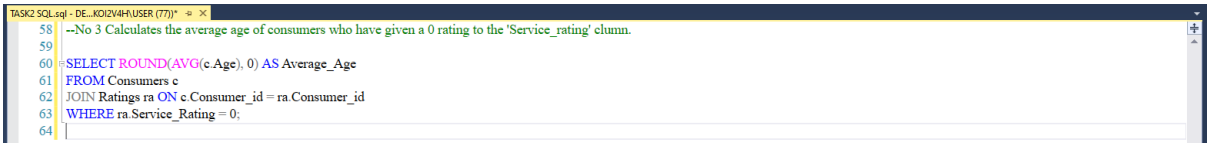



**Figure 2.2.9:** This TSQL query shows the average age of consumers giving restaurants a service rating of 0.

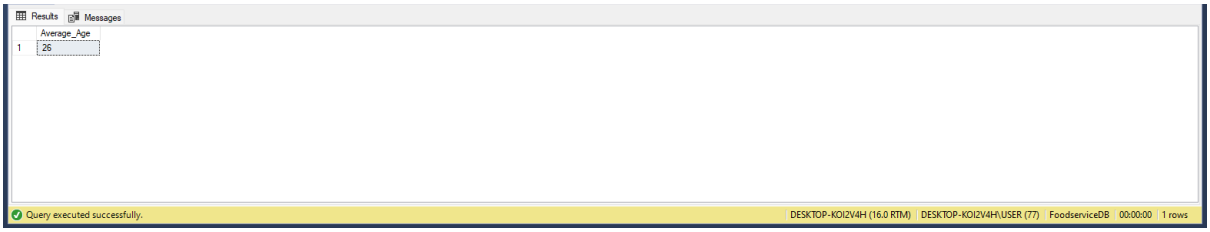



**Figure 2.3.0:** Result from the executed query in **Figure 2.2.9**

**Answer to Question 4.**

4. Having a composite primary key constraint to the Ratings table with the columns Consumer_ID and Restaurant_ID which are foreign keys corresponding the primary keys in the Consumer and Restaurant tables, the join conditions is used to first join the Restaurant table with the Ratings table and then the Ratings table with the Consumer table, after which we filter the consumers who are the youngest by using a subquery in the WHERE clause to find the minimum age, then we sort the results based on the food rating in the descending order.


**Figure 2.3.1:** This query returns the restaurants ranked by the youngest consumer

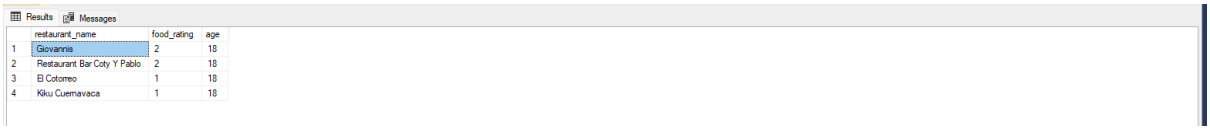



**Figure 2.3.2:** Result from the query in **Figure 2.3.1**

**Answer to Question 5.**

5. The stored procedure '**UpdateServiceRatingWithParking**' modifies the '**Service_rating**' entries in the 'Ratings' table. It specifically targets restaurants with available parking, identified by values of 'yes' or 'public' in the 'Parking' column of the 'Restaurant' table. The procedure sets the service the service rating for these restaurants to '2' upon completion. It displays confirming the update of service ratings for restaurants with parking availability.

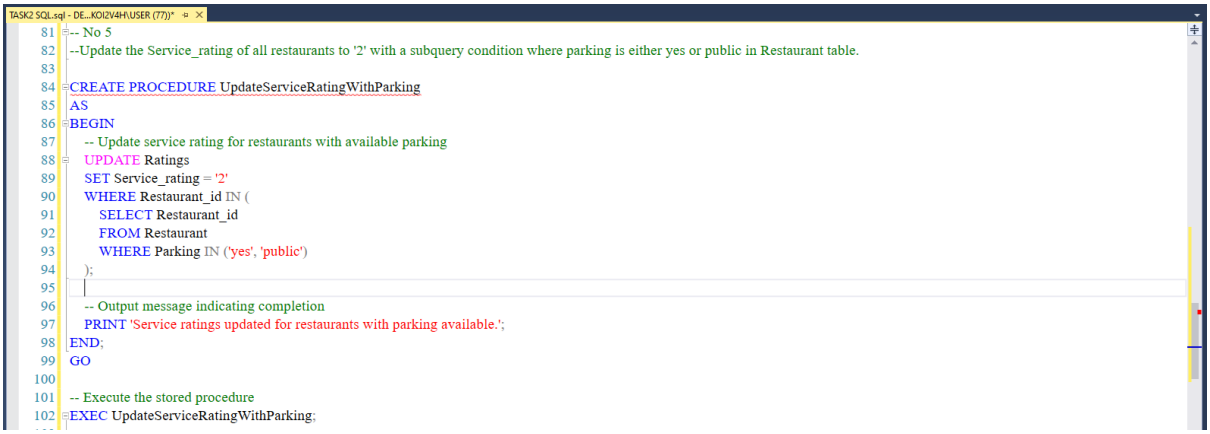



**Figure 2.3.3:** The TSQL query to write a stored procedure

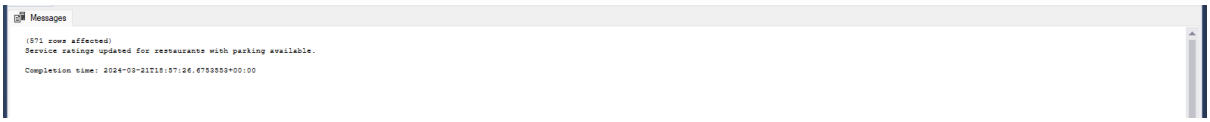
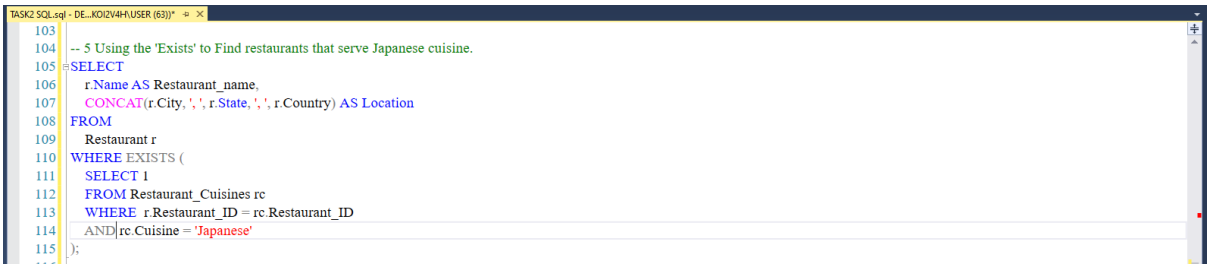



**Figure 2.3.4:** The result shows **571** rows affected and the print statement

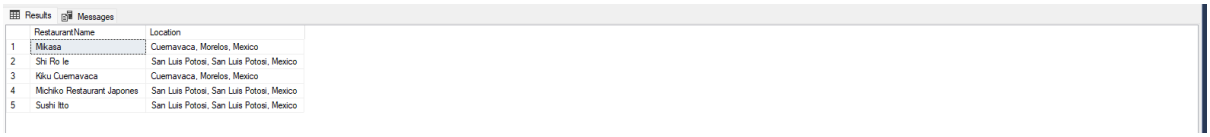
**Answer to Question 6.**

(i) Using the '**EXISTS**' clause to check for the existence of at least one row in the Restaurant_Cuisines table where the restaurant serves Japanese cuisine, this was done by matching the Restaurant_ID column from the outer query with the Restaurant _ID in the subquery, the outer query selects restaurant names and their location from the Restaurant table



```
103
104  -- 5 Using the 'Exists' to Find restaurants that serve Japanese cuisine.
105  SELECT
106      r.Name AS Restaurant_name,
107      CONCAT(r.City, ', ', r.State, ', ', r.Country) AS Location
108  FROM
109      Restaurant r
110  WHERE EXISTS (
111      SELECT 1
112      FROM Restaurant_Cuisines rc
113      WHERE  r.Restaurant_ID = rc.Restaurant_ID
114      AND rc.Cuisine = 'Japanese'
115  );
```

**Figure 2.3.5:** This TSQL query finds restaurants that serve Japanese cuisine and returns their name and location.



Results | Messages

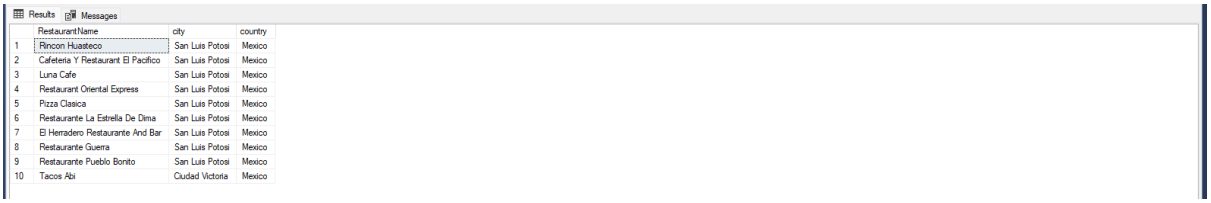
| | RestaurantName | Location |
|---|---|---|
| 1 | Mikasa | Cuernavaca, Morelos, Mexico |
| 2 | Shi Ro Ie | San Luis Potosi, San Luis Potosi, Mexico |
| 3 | Kiku Cuernavaca | Cuernavaca, Morelos, Mexico |
| 4 | Michiko Restaurant Japones | San Luis Potosi, San Luis Potosi, Mexico |
| 5 | Sushi Itto | San Luis Potosi, San Luis Potosi, Mexico |

**Figure 2.3.6:** Here is the result of the query in **Figure 2.3.5**

(ii) Using '**IN**' in this case, the Innermost subquery selects consumers who are younger than 21 years old from the Consumers table, while the middle subquery matches Ratings table with the innermost query using Consumer_id to ensure we get ratings made by consumers who are younger than 21. Finaly the outer query then selects the names, city and country from the Restaurant table where the Restaurant_id's matches in the middle query, to ensure we get restaurants that have been rated by consumers younger than 21

```
SELECT TOP 10 Name AS RestaurantName, city, country
FROM Restaurant
WHERE Restaurant_id IN (
    SELECT Restaurant_id
    FROM Ratings
    WHERE Consumer_id IN (
        SELECT Consumer_id
        FROM Consumers
        WHERE Age < 21
    )
);
```

**Figure 2.3.7:** The TSQL query selects the names of restaurants from the restaurant table that have been rated by consumers younger than 21 years old and then the result limited to TOP ten.



| | RestaurantName | city | country |
|---|---|---|---|
| 1 | Rincon Huasteco | San Luis Potosi | Mexico |
| 2 | Cafeteria Y Restaurant El Pacifico | San Luis Potosi | Mexico |
| 3 | Luna Cafe | San Luis Potosi | Mexico |
| 4 | Restaurant Oriental Express | San Luis Potosi | Mexico |
| 5 | Pizza Clasica | San Luis Potosi | Mexico |
| 6 | Restaurante La Estrella De Dima | San Luis Potosi | Mexico |
| 7 | El Herradero Restaurante And Bar | San Luis Potosi | Mexico |
| 8 | Restaurante Guerra | San Luis Potosi | Mexico |
| 9 | Restaurante Pueblo Bonito | San Luis Potosi | Mexico |
| 10 | Tacos Abi | Ciudad Victoria | Mexico |

**Figure 2.3.8:** Here is the result of the query in **Figure 2.3.7**

(iii) Using '**System Function**', to find the average food rating of restaurants that offer alcohol service to those that do not

```
-- (iii)

--What is the average food rating of restaurants that offer alcohol service compared to those that do not?
    SELECT
    ISNULL(R.Alcohol_Service, 'No Alcohol Service') AS Alcohol_Service_Status,
    AVG(RT.Food_Rating) AS Avg_Food_Rating
FROM
    Restaurant R
LEFT JOIN
    Ratings RT ON R.Restaurant_ID = RT.Restaurant_ID
GROUP BY
    R.Alcohol_Service;
```

**Figure 2.3.9** This query uses **ISNULL** to determine a restaurant's average (**AVG**) meal rating by considering factors like alcohol service. It helps restaurant management make informed decisions about menu items by highlighting the relationship between food and drink service ratings.

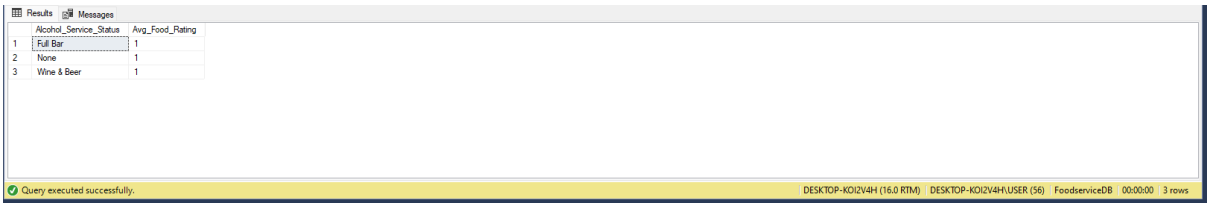



**Figure 2.4.0** Here is the result of the query in **Figure 2.3.9**

(iv) Using **Groupby**, **Having** and **OrderBy**, the TSQL query selects Restaurant_ID and Name which was Alias as RestaurantName from the Restaurant table, uses the AVG() function on the service_rating column to determine the average service rating and then aliased the column as 'AvgServiceRating' from the Ratings table. The query then performs a JOIN operation on Restaurant and Ratings tables, groups the result by Restaurant_ID and Name to calculate the average service rating for each restaurant. Subsequently, it used the HAVING clause to filter out restaurants where the average service rating is less than 3. Finally, the result is ordered by the average service rating in descending order.

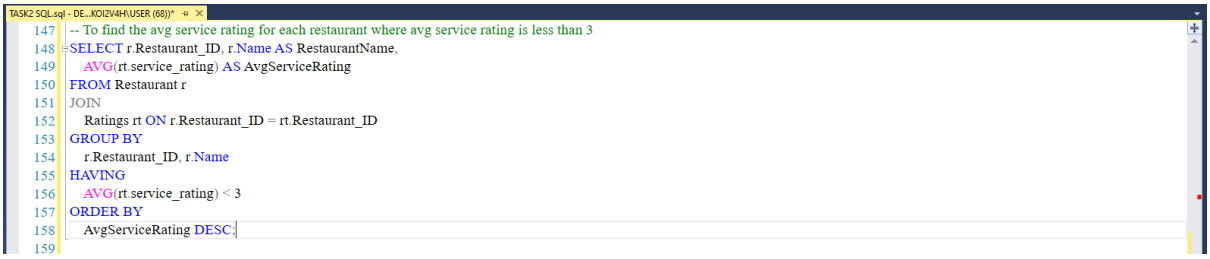
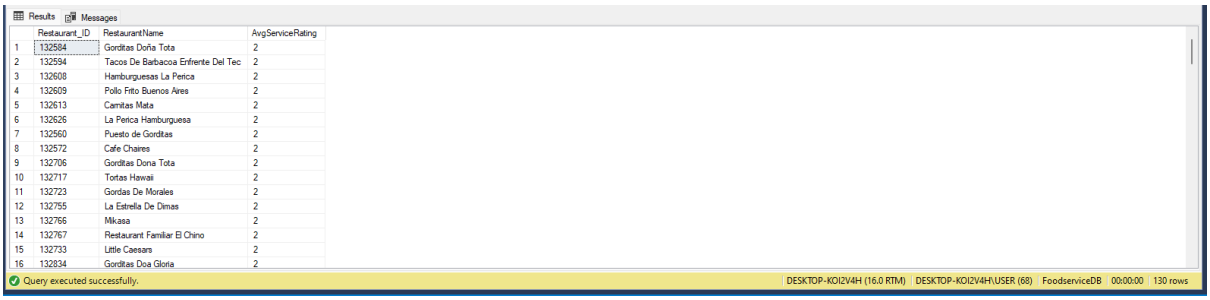



**Figure 2.4.1:** The TSQL query to return Restaurant_ID and Name with AvgServiceRating less than 3 and ordered by the average rating I descending order.



Results Messages

| | Restaurant_ID | RestaurantName | AvgServiceRating |
|---|---|---|---|
| 1 | 132584 | Gorditas Doña Tota | 2 |
| 2 | 132594 | Tacos De Barbacoa Enfrente Del Tec | 2 |
| 3 | 132608 | Hamburguesas La Perica | 2 |
| 4 | 132609 | Pollo Frito Buenos Aires | 2 |
| 5 | 132613 | Carnitas Mata | 2 |
| 6 | 132626 | La Perica Hamburguesa | 2 |
| 7 | 132560 | Puesto de Gorditas | 2 |
| 8 | 132572 | Cafe Chaires | 2 |
| 9 | 132706 | Gorditas Dona Tota | 2 |
| 10 | 132717 | Tortas Hawai | 2 |
| 11 | 132723 | Gordas De Morales | 2 |
| 12 | 132755 | La Estrella De Dimas | 2 |
| 13 | 132766 | Mikasa | 2 |
| 14 | 132767 | Restaurant Familiar El Chino | 2 |
| 15 | 132733 | Little Caesars | 2 |
| 16 | 132834 | Gorditas Doa Gloria | 2 |

Query executed successfully. DESKTOP-KOI2V4H (16.0 RTM) | DESKTOP-KOI2V4H\USER (68) | FoodserviceDB | 00:00:00 | 130 rows

**Figure 2.4.2:** Here is the result from the query in **Figure 2.4.1**

## CONCLUSION

In today's data-fast moving corporate environment, businesses need to analyze huge amounts of information using SQL queries to identify patterns, trends, and areas for improvement. Automating data management can be a cost-effective and efficient solution, as stored procedures ensure consistency and consistency in data protocols. For instance, a stored process can update service ratings based on parking availability.

**Decision Support:** Making wise choices is crucial to succeed in the cutthroat restaurant business. Decision-makers can gain important information from SQL queries into a range of restaurant operations topics, such as menu planning, marketing campaigns, and operational improvement. For instance, restaurants can find-selling items, ascertain which menu items are failing, and modify their menu offers by employing SQL queries to analyze sales data.

**Data Integrity:** Ensuring the quality, consistency, and dependability of database information depends on maintaining data integrity. In order to maintain referential integrity between related tables and lower mistakes and inconsistencies in the database, primary and foreign key constraints are essential. A foreign key constraint, on the other hand, creates a relationship between tables and stops the insertion of erroneous data.

To sum up, stored procedures and SQL queries are essential for organizing and evaluating restaurant data, which helps companies create strategic plans and make wise judgments.