University of Salford MANCHESTER

ADVANCED DATABASES COURSEWORK

NAME: EMMANUEL AMADI OTUDE

STUDENT ID: @ 00705726

Task 1

Designing a Robust Hospital Management Database System

Introduction:

Task 1 involves designing a hospital management system database, encompassing patient information, medical records, appointments, doctors, and departments. I'll start by analyzing client requirements, identifying entities, attributes, and relationships. Using entity-relationship modelling, I'll design the schema and normalize it to 3NF to reduce redundancy. Implementation in SQL Server Management Studio includes table creation, constraint definition, and relationship establishment. Additional objects like stored procedures and views will be implemented as needed. Prioritizing data integrity, security, and backup measures, I'll document design decisions and provide a detailed report to the client.

Part 1

A. Database Design and Normalization

Entity: Attributes

- Departments: (DepartmentID, DepartmentName)
- Doctors: (DoctorID, FirstName, LastName, DepartmentID, EmailAddress, TelephoneNumber, Username, Password, Specialty)
- Patients: (PatientID, FirstName, LastName, DateOfBirth, Insurance, Username, Password, EmailAddress, TelephoneNumber, DateLeft)
- Addresses: (AddressID, PatientID, Address1, Address2, City, Postcode)
- Appointments: (AppointmentID, PatientID, DoctorID, AppointmentDate, Status)
- MedicalRecords: (RecordID, PatientID, DoctorID, DateOfVisit, Allergies)
- Updates: (UpdateID, RecordID, Diagnoses, Medicines, MedicinePrescribedDate)
- Availability: (AvailabilityID, DoctorID, Date, TimeSlot, AvailabilityStatus)
- Reviews: (ReviewID, DoctorID, PatientID, Review, Rating)
- Feedbacks: (FeedbackID, PatientID, Feedback, FeedbackDate)

Relationships:

Departments - Doctors:

- This is a one-to-many (1:N) relationship, where the "DepartmentID" in the Doctors table serves as a foreign key referencing the "DepartmentID" in the Departments table.

Patients - Addresses:

- This is a one-to-many (1:N) relationship, where the "PatientID" in the Addresses table serves as a foreign key referencing the "PatientID" in the Patients table.

Patients - Appointments:

- It's a one-to-many (1:N) relationship, where the "PatientID" in the Appointments table serves as a foreign key referencing the "PatientID" in the Patients table.

Doctors - Appointments:

- A one-to-many (1:N) relationship, where the "DoctorID" in the Appointments table serves as a foreign key referencing the "DoctorID" in the Doctors table.

Patients - MedicalRecords:

- This is a one-to-many (1:N) relationship, where the "PatientID" in the MedicalRecords table serves as a foreign key referencing the "PatientID" in the Patients table.

Doctors - MedicalRecords:

- This is a one-to-many (1:N) relationship, where the "DoctorID" in the MedicalRecords table serves as a foreign key referencing the "DoctorID" in the Doctors table.

MedicalRecords - Updates:

- This is a one-to-many (1:N) relationship, where the "RecordID" in the Updates table serves as a foreign key referencing the "RecordID" in the MedicalRecords table.

Doctors - Availability:

- One-to-many (1:N) relationship, where the "DoctorID" in the Availability table serves as a foreign key referencing the "DoctorID" in the Doctors table.

Doctors - Reviews:

- This is a one-to-many (1:N) relationship, where the "DoctorID" in the Reviews table serves as a foreign key referencing the "DoctorID" in the Doctors table.

Patients - Reviews:

- This is a one-to-many (1:N) relationship, where the "PatientID" in the Reviews table serves as a foreign key referencing the "PatientID" in the Patients table.

Patients - Feedbacks:

- This is a one-to-many (1:N) relationship, where the "PatientID" in the Feedbacks table serves as a foreign key referencing the "PatientID" in the Patients table.

Normalization:

First Normal Form (1NF): Looking at our initial design, all tables satisfy the requirements of 1NF as each column contains atomic values, and there are no repeating groups.

Second Normal Form (2NF): If any non-key attribute is dependent on only a portion of the primary key, it should be moved to a separate table.

Analyzing each table:

- **Patients**: All attributes (**FirstName**, **LastName**, **AddressID**, **DateOfBirth**, **Insurance**, **Username**, **Password**, **DateLeft**) are fully functionally dependent on the primary key (**PatientID**).
- **Addresses**: All attributes (**AddressID**, **PatientID**, **Address1**, **Address2**, **City**, **Postcode**) are fully functionally dependent on the primary key (**AddressID**).
- **Doctors**: All attributes (**DoctorID**, **FirstName**, **LastName**, **DepartmentID**, **EmailAddress**, **TelephoneNumber**, **Username**, **Password**, **Specialty**) are fully functionally dependent on the primary key (**DoctorID**).
- **Departments**: All attributes (**DepartmentID**, **DepartmentName**) are fully functionally dependent on the primary key (**DepartmentID**).
- **Appointments**: All attributes (**AppointmentID**, **PatientID**, **DoctorID**, **AppointmentDate**, **AppointmentTime**, **Status**) are fully functionally dependent on the primary key (**AppointmentID**).
- **MedicalRecords**: All attributes (**RecordID**, **PatientID**, **Allergies**, **DateOfVisit**) are fully functionally dependent on the primary key (**RecordID**).
- **Updates**: All attributes (**UpdateID**, **RecordID**, **DoctorID**, **Diagnoses**, **Medicines**, **MedicinePrescribedDate**) are fully functionally dependent on the primary key (**UpdateID**).

- **Availability**: All attributes (**AvailabilityID**, **DoctorID**, **Date**, **TimeSlot**, **AvailabilityStatus**) are fully functionally dependent on the primary key (**AvailabilityID**).
- **Reviews**: All attributes (**ReviewID**, **DoctorID**, **PatientID**, **Review**, **Rating**) are fully functionally dependent on the primary key (**ReviewID**).
- **Feedbacks**: All attributes (**FeedbackID**, **PatientID**, **Feedback**, **FeedbackDate**) are fully functionally dependent on the primary key (**FeedbackID**).

Third Normal Form (3NF): In 3NF, a table should be in 2NF, and no transitive dependencies should exist.

Upon analyzing the tables:

Patients: The **AddressID** attribute is functionally dependent on the **PatientID** (primary key), so there's no transitive dependency.

Doctors: The **DepartmentID** attribute is functionally dependent on the **DoctorID** (primary key), so there's no transitive dependency.

Appointments: There are no transitive dependencies.

MedicalRecords: All attributes are directly dependent on the primary key (**RecordID**), so there are no transitive dependencies.

Updates: All attributes are directly dependent on the primary key (**UpdateID**), with no transitive dependencies.

Availability: There are no transitive dependencies.

Reviews: There are no transitive dependencies.

Feedbacks: There are no transitive dependencies.

Addresses and **Departments**: These tables have no non-prime attributes other than the primary key, no transitive dependencies.

Therefore, all tables are in 3NF as there are no transitive dependencies.

B. SQL Studio Management System (SSMS)

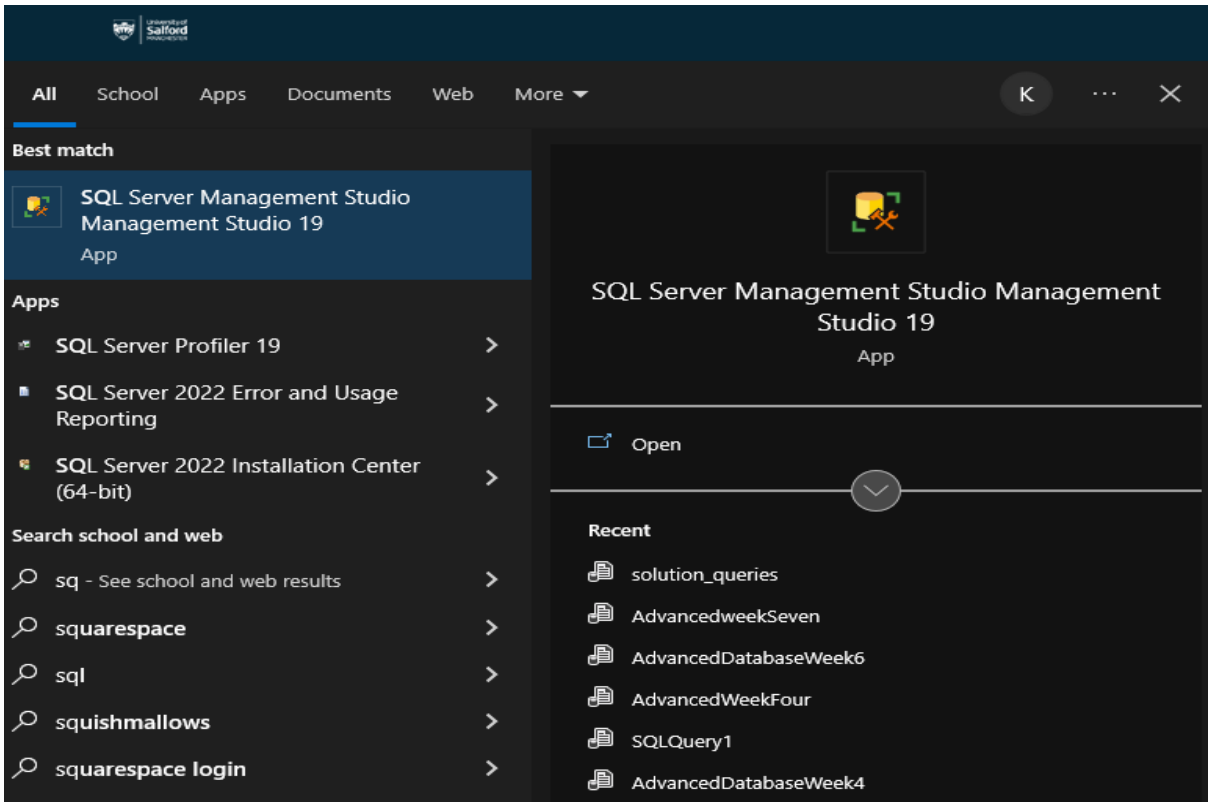


Fig 2.1.1

Searching for "SQL Server Managenment Studio"

Launch SSMS by typing the word "SQL Server Management Studio" into the Windows search box.

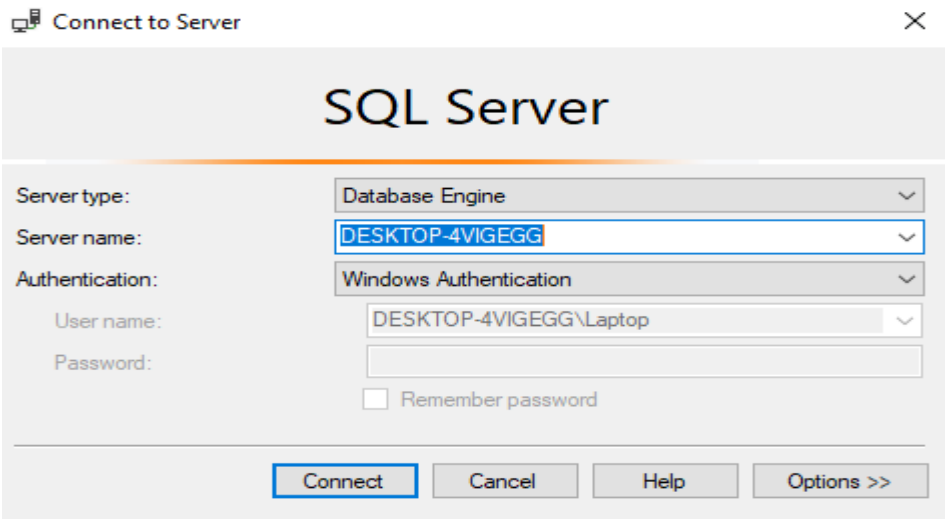


Fig 2.1.2

Connecting to the server

C. Using the CREATE DATABASE statement below:

create database HospitalManagementSystem;

Check if it has been successfully created by refreshing the object Explorer and clicking the plus sign next to Databases to expand. Please note that if the code has been executed, you won't be able to execute a second time as it now already exists.

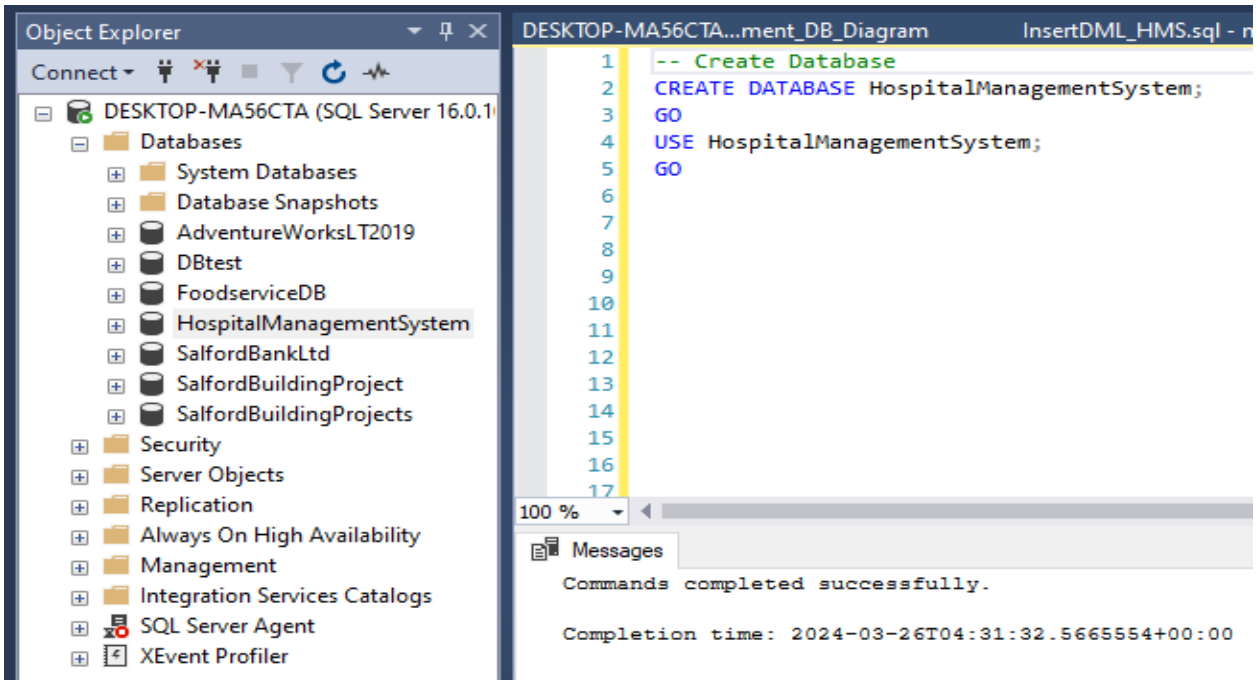


Fig 2.1.3 **Sets the current database for use**.

use HospitalManagementSystem;

This sets the current database for use and running, query changes from the default one.

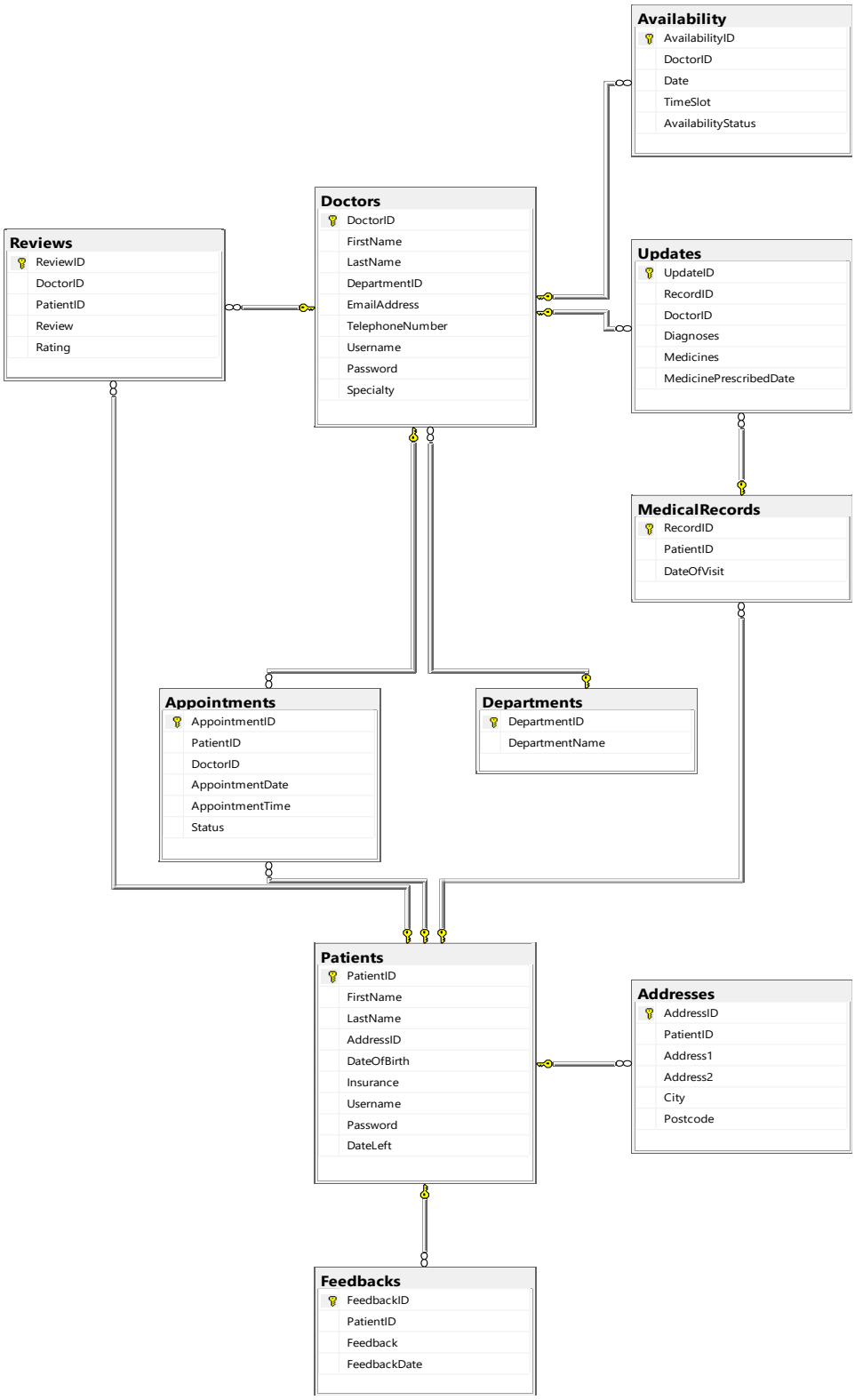


Fig 2.1.4

Database diagram

Database Diagram schema for a Hospital Management System consisting of several tables:

- Patients Table: Connects to Addresses Table via AddressID for patient address association.
- Doctors Table: Links the Departments Table through DepartmentID for doctor-department association.
- Appointments Table: Associated with Patients and Doctors tables via PatientID and DoctorID respectively for patient-doctor appointment correlation.
- Medical Records Table: Related to Patients Table via PatientID and Updates Table via RecordID and Doctors Table via DoctorID for medical visit recording and update tracking.
- Availability Table: Connected to Doctors Table via DoctorID for doctor availability management.
- Reviews Table: Associated with Doctors Table via DoctorID for patient reviews on doctors.
- Feedbacks Table: Linked to Patients Table via PatientID for storing patient feedback entries.

D. TABLE CREATION QUERIES

Patients Table:

The patients' information including their personal details, contact information, and Date left are stored on this table.

```sql
-- Creating Patient Table

CREATE TABLE Patients (
    PatientID INT PRIMARY KEY IDENTITY(1, 1),
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    AddressID INT,
    DateOfBirth DATE NOT NULL,
    Insurance NVARCHAR(50) NOT NULL,
    Username NVARCHAR(50) NOT NULL,
    Password NVARCHAR(50) NOT NULL,
    DateLeft DATE
);
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-12T18:24:59.0529017+01:00
```

Interpretation of table provided:

Patients Table:

- **PatientID INT PRIMARY KEY IDENTITY(1, 1)**: This stores the unique identifier for each patient. It is of type **INT**, meaning it stores whole numbers. With **IDENTITY(1,1)**, it is set to auto-increment starting from 1, ensuring each patient gets a unique ID.
- **FirstName NVARCHAR(50) NOT NULL**: It's for storing the first name of the patient. It is of variable-length Unicode string type and does not allow NULL values.
- **LastName NVARCHAR(50) NOT NULL**: This stores the last name of the patient. It is of variable-length Unicode string type and does not allow NULL values.
- **AddressID INT**: This column stores the ID of the patient's address. It's a Foreign key, referencing the address of the patient in the Addresses table. It is an integer column.
- **DateOfBirth DATE NOT NULL**: Defines a column for storing the date of birth of the patient. It is of date type and does not allow NULL values.
- **Insurance NVARCHAR(50) NOT NULL**: It's a column for storing the insurance information of the patient. It is of variable-length Unicode string type and does not allow NULL values.
- **Username NVARCHAR(50) NOT NULL**: This column stores the username chosen by the patient for login purposes. It is of type **NVARCHAR(50)** a variable-length Unicode string type and does not allow NULL values.
- **Password NVARCHAR(50) NOT NULL**: This column stores the password chosen by the patient for login purposes. It is of type **NVARCHAR(50)** a variable-length Unicode string type and does not allow NULL values.
- **DateLeft DATE**: This defines a column for storing the date when the patient left the hospital. It is of date type and allows NULL values.

```sql
-- Creating Addresses Table

CREATE TABLE Addresses (
    AddressID INT PRIMARY KEY IDENTITY(1, 1),
    PatientID INT,
    Address1 NVARCHAR(50) NOT NULL,
    Address2 NVARCHAR(50),
    City NVARCHAR(50) NOT NULL,
    Postcode NVARCHAR(10) NOT NULL,
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID)
);
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-12T18:24:59.0529017+01:00
```

Addresses Table:

- **AddressID INT PRIMARY KEY IDENTITY(1, 1)**: This line defines the primary key column for uniquely identifying addresses. It is an integer column that automatically increments with each new entry.
- **PatientID INT**: This line defines a column for storing the ID of the patient associated with the address. It is an integer column.
- **Address1 NVARCHAR(50) NOT NULL**: This line defines a column for storing the first line of the address. It is of variable-length Unicode string type and does not allow NULL values.
- **Address2 NVARCHAR(50)**: This line defines a column for storing the second line of the address. It is of variable-length Unicode string type and allows NULL values.
- **City NVARCHAR(50) NOT NULL**: This line defines a column for storing the city of the address. It is of variable-length Unicode string type and does not allow NULL values.
- **Postcode NVARCHAR(10) NOT NULL**: This line defines a column for storing the postcode of the address. It is of variable-length Unicode string type and does not allow NULL values.

Doctor Table:

Stores information about doctors including their department, personal details, and login credentials.

```sql
-- Creating Doctors Table

CREATE TABLE Doctors (
    DoctorID INT PRIMARY KEY IDENTITY(1, 1),
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    DepartmentID INT NOT NULL,
    EmailAddress NVARCHAR(50),
    TelephoneNumber NVARCHAR(15),
    Username NVARCHAR(50) NOT NULL,
    Password NVARCHAR(50) NOT NULL,
    Specialty NVARCHAR(50),
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-12T18:56:47.3498746+01:00
```

- DoctorID: This column serves as the unique identifier for each doctor. It is of type INT and is set to auto-increment starting from 1 (IDENTITY(1,1)).
- FirstName NVARCHAR(50) NOT NULL: This defines a column for storing the first name of the doctor. It is of variable-length Unicode string type and does not allow NULL values.
- LastName NVARCHAR(50) NOT NULL: It's a column for storing the last name of the doctor. It is of variable-length Unicode string type and does not allow NULL values.
- DepartmentID: This column represents the department to which the doctor belongs. It is a foreign key referencing the Departments table's DepartmentID column.
- EmailAddress **NVARCHAR(50)**: This column stores the email address of the doctor. It is of type NVARCHAR(50) a variable-length Unicode string type and allows NULL values.
- TelephoneNumber **NVARCHAR(15)**: This defines a column for storing the telephone number of the doctor. It is of variable-length Unicode string type and allows NULL values.
- Password **NVARCHAR(50) NOT NULL**: This column stores the password for the doctor's account. It is of type NVARCHAR(50) a variable-length Unicode string type and does not allow NULL values.
- Specialty **NVARCHAR(50)**: This defines a column for storing the specialty of the doctor. It is of variable-length Unicode string type and allows NULL values.

Department Table:

Stores information about different departments in the hospital.

```sql
-- Creating Department Table
CREATE TABLE Department (
    DepartmentID INT PRIMARY KEY IDENTITY(1,1),
    DepartmentName NVARCHAR(50) NOT NULL
);
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-03-26T04:31:32.5665554+00:00
```

- **DepartmentID**: This column serves as the unique identifier for each department. It is of type **INT** and is set to auto-increment starting from 1 (**IDENTITY(1,1)**).
- **DepartmentName**: This column stores the name of the department. It is of type **NVARCHAR(50)** and is marked as **NOT NULL**.

Appointment Table:

Manages appointments between patients and doctors, including appointment date, time, and status.

```sql
-- Creating Appointments Table

CREATE TABLE Appointments (
    AppointmentID INT PRIMARY KEY IDENTITY(1, 1),
    PatientID INT,
    DoctorID INT,
    AppointmentDate DATE NOT NULL,
    AppointmentTime TIME NOT NULL,
    Status VARCHAR(10) DEFAULT 'Pending' CHECK (Status IN ('Pending', 'Cancelled', 'Completed')),
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID),
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID)
);
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-12T18:56:47.3498746+01:00
```

- **AppointmentID**: This column serves as the unique identifier for each appointment. It is of type **INT** and is set to auto-increment starting from 1 (**IDENTITY(1,1)**).
- **PatientID**: Shows the patient associated with the appointment. It is a foreign key referencing the **Patient** table's **PatientID** column.

- DoctorID: Represents the doctor associated with the appointment. It is a foreign key referencing the **Doctor** table's **DoctorID** column.
- AppointmentDate DATE NOT NULL: For storing the date of the appointment. It is of date type and does not allow NULL values.
- AppointmentTime TIME NOT NULL: Storing the time of the appointment. It is of time type and does not allow NULL values.
- Status: Stores the status of the appointment, which can be 'Pending', 'Cancelled', or 'Completed'. It is of type **NVARCHAR(10)** and defaults to 'Pending'.
- Constraints ensure that the appointment date is not in the past and that the appointment time is within a valid range.

Medical Record Table:

- Stores medical records including allergies and DateOfVisit.

```sql
-- Creating MedicalRecords Table

CREATE TABLE MedicalRecords (
    RecordID INT PRIMARY KEY IDENTITY(1, 1),
    PatientID INT,
    Allergies NVARCHAR(100),
    DateOfVisit DATE NOT NULL,
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID)
);

```

- **RecordID**: This column serves as the unique identifier for each medical record. It is of type **INT** and is set to auto-increment starting from 1 (**IDENTITY(1,1)**).
- **PatientID INT**: This defines a column for storing the ID of the patient associated with the medical record. It is a foreign key referencing the **Patients** table's **PatientID** column. It is an integer(INT) type column.
- **Allergies**: This column stores information about allergies of the patient. It is of type **NVARCHAR(100)** and allows NULL values.
- **DateOfVisit DATE NOT NULL**: It defines a column for storing the date of the medical visit. It is of date type and does not allow NULL values.

Updates Table:

The Updates table records updates made by doctors on patients' medical records.

```sql
-- Creating Updates Table

CREATE TABLE Updates (
    UpdateID INT PRIMARY KEY IDENTITY(1, 1),
    RecordID INT,
    DoctorID INT,
    Diagnoses NVARCHAR(500) NOT NULL,
    Medicines NVARCHAR(500) NOT NULL,
    MedicinePrescribedDate DATETIME DEFAULT GETDATE() NOT NULL,
    FOREIGN KEY (RecordID) REFERENCES MedicalRecords(RecordID),
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID)
);
```

- **Updates Table**:
    - **UpdateID INT PRIMARY KEY IDENTITY(1, 1)**: This line defines the primary key column for uniquely identifying updates. It is an integer column that automatically increments with each new entry.
    - **RecordID INT**: This field is set for storing the ID of the medical record associated with the update. It is a foreign key referencing the MedicalRecords table's **RecordID** column. It is an integer column.
    - **DoctorID INT**: It's a column for storing the ID of the doctor associated with the update. It is an integer column.
    - **Diagnoses NVARCHAR(500) NOT NULL**: This defines a column for storing diagnoses made during the medical visit. It is of variable-length Unicode string type and does not allow NULL values.
    - **Medicines NVARCHAR(500) NOT NULL**: This field is for storing prescribed medicines. It is of variable-length Unicode string type and does not allow NULL values.
    - **MedicinePrescribedDate DATETIME DEFAULT GETDATE() NOT NULL**: It's a column for storing the date when the medicines were prescribed. It is of datetime type with a default value of the current date and time, and it does not allow NULL values.

Availability Table:

The Availability table tracks the availability of doctors for appointments.

```sql
-- Creating Availability Table

CREATE TABLE Availability (
    AvailabilityID INT PRIMARY KEY IDENTITY(1, 1),
    DoctorID INT,
    Date DATE NOT NULL,
    TimeSlot TIME NOT NULL,
    AvailabilityStatus VARCHAR(15) NOT NULL,
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID)
);
```

- AvailabilityID: Serves as the unique identifier for each availability entry. It is of type INT and is set to auto-increment starting from 1 (IDENTITY(1,1)).
- DoctorID: Shows the doctor associated with the availability entry. It is a foreign key referencing the Doctor table's DoctorID column.
- Date DATE NOT NULL: This stores the date of the availability slot. It is of date type and does not allow NULL values.
- TimeSlot TIME NOT NULL: Stores the time of the availability slot. It is of time type and does not allow NULL values.
- AvailabilityStatus VARCHAR(10) NOT NULL: For storing the status of the availability slot. It is a variable-length string type and does not allow NULL values.

Reviews Table:

Related to both Patients and Doctors table

```sql
-- Creating Reviews Table

CREATE TABLE Reviews (
    ReviewID INT PRIMARY KEY IDENTITY(1, 1),
    DoctorID INT,
    PatientID INT,
    Review NVARCHAR(500) NOT NULL,
    Rating INT CHECK (Rating >= 1 AND Rating <= 5) NOT NULL,
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID),
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID)
);
```

- **ReviewID INT PRIMARY KEY IDENTITY(1, 1):** It's the primary key column for uniquely identifying reviews. It is an integer column that automatically increments with each new entry.

- **DoctorID INT:** Stores the ID of the doctor associated with the review. It is a foreign key referencing the **Doctors** table's **DoctorID** column. It is an integer column.
- **PatientID INT:** This is an integer(INT) type field for storing the ID of the patient providing the review. It is a foreign key referencing the **Patients** table's **PatientID** column.
- **Review NVARCHAR(500) NOT NULL:** Stores the review provided by the patient. It is of variable-length Unicode string type and does not allow NULL values.
- **Rating INT CHECK (Rating >= 1 AND Rating <= 5) NOT NULL:** It's a field for storing the rating provided by the patient. It is an integer column constrained to be between 1 and 5, and it does not allow NULL values.

Feedbacks Table:

```sql
-- Creating Feedback Table

CREATE TABLE Feedbacks (
    FeedbackID INT PRIMARY KEY IDENTITY(1, 1),
    PatientID INT,
    Feedback NVARCHAR(500) NOT NULL,
    FeedbackDate DATETIME DEFAULT GETDATE() NOT NULL,
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID)
);
```

100 %

- FeedbackID: This column serves as the unique identifier for each feedback entry. It is of type INT and is set to auto-increment starting from 1 (IDENTITY(1,1)).
- PatientID: This column represents the patient who provided the feedback. It is a foreign key referencing the Patients table's PatientID column.
- Feedback: This column stores the text content of the feedback provided by the patient. It is of type NVARCHAR(500).
- **FeedbackDate**: This column stores the date when the feedback was provided. It is of type **DATETIME** and defaults to the current date and time (**GETDATE()**).

DATA INSERTION COMMANDS IN THE ABOVE CREATED TABLES.

We are going to insert data into the tables we have created. Insert is a T-SQL Statement which is used to insert new records into a database.

- Populating records in the Patients table we use the statement:

```sql
-- Populating Patients Table
INSERT INTO Patients (FirstName, LastName, AddressID, DateOfBirth, Insurance, Username, Password, DateLeft)
VALUES
('John', 'Doe', 1, '1990-05-15', 'ABC Insurance', 'johndoe123', 'password123', NULL),
('Jane', 'Smith', 2, '1985-09-20', 'XYZ Insurance', 'janesmith456', 'securepassword', NULL),
('Michael', 'Johnson', 3, '1978-11-10', '123 Insurance', 'michaelj', 'pass123', NULL),
('Emily', 'Williams', 4, '1995-03-25', 'Insurance Co.', 'emilyw', 'password1234', NULL),
('David', 'Brown', 5, '1982-07-08', 'Insurance Group', 'davidb', 'abc123', NULL),
('Sarah', 'Miller', 6, '1970-12-03', 'Health Insurance', 'sarahm', 'mypass', NULL),
('Ryan', 'Wilson', 7, '1988-06-18', 'Medical Insurance', 'ryanw', 'password321', NULL),
('Jessica', 'Taylor', 8, '1992-09-30', 'Insurance Corp.', 'jessicat', 'taylor123', NULL),
('Christopher', 'Anderson', 9, '1980-04-12', 'Insure Inc.', 'chrisa', 'pass1234', NULL),
('Amanda', 'Martinez', 10, '1975-01-22', 'Healthcare Insurance', 'amandam', 'securepass', NULL);
Go
SELECT * FROM Patients;
```

100 %

Results | Messages

	PatientID	FirstName	LastName	AddressID	DateOfBirth	Insurance	Username	Password	DateLeft
1	1	John	Doe	1	1990-05-15	ABC Insurance	johndoe123	password123	NULL
2	2	Jane	Smith	2	1985-09-20	XYZ Insurance	janesmith456	securepassword	NULL
3	3	Michael	Johnson	3	1978-11-10	123 Insurance	michaelj	pass123	NULL
4	4	Emily	Williams	4	1995-03-25	Insurance Co.	emilyw	password1234	NULL
5	5	David	Brown	5	1982-07-08	Insurance Group	davidb	abc123	NULL
6	6	Sarah	Miller	6	1970-12-03	Health Insurance	sarahm	mypass	NULL
7	7	Ryan	Wilson	7	1988-06-18	Medical Insurance	ryanw	password321	NULL
8	8	Jessica	Taylor	8	1992-09-30	Insurance Corp.	jessicat	taylor123	NULL
9	9	Christopher	Anderson	9	1980-04-12	Insure Inc.	chrisa	pass1234	NULL
10	10	Amanda	Martinez	10	1975-01-22	Healthcare Insurance	amandam	securepass	NULL
11	11	John	Doe	1	1990-05-15	ABC Insurance	johndoe123	password123	NULL

Query executed successfully. | DESKTOP-MA56CTA (16.0 RTM) | DESKTOP-MA56CTA\Chiche... | Ho

- Populating records in the Addresses table we use the statement:

```sql
-- Populating Addresses Table
INSERT INTO Addresses (PatientID, Address1, City, Postcode)
VALUES
(1, '123 Main St', 'Cityville', '12345'),
(2, '456 Elm St', 'Townsville', '54321'),
(3, '789 Oak St', 'Villagetown', '67890'),
(4, '111 Pine St', 'Hamlet', '13579'),
(5, '222 Maple St', 'Ruraltown', '97531'),
(6, '333 Cedar St', 'Suburbia', '24680'),
(7, '444 Birch St', 'Metropolis', '86420'),
(8, '555 Walnut St', 'Smalltown', '64208'),
(9, '666 Spruce St', 'Citytown', '37589'),
(10, '777 Ash St', 'Hometown', '90876');
Go
SELECT * FROM Addresses
```

100 %

Results | Messages

	AddressID	PatientID	Address1	Address2	City	Postcode
1	1	1	123 Main St	NULL	Cityville	12345
2	2	2	456 Elm St	NULL	Townsville	54321
3	3	3	789 Oak St	NULL	Villagetown	67890
4	4	4	111 Pine St	NULL	Hamlet	13579
5	5	5	222 Maple St	NULL	Ruraltown	97531
6	6	6	333 Cedar St	NULL	Suburbia	24680
7	7	7	444 Birch St	NULL	Metropolis	86420
8	8	8	555 Walnut St	NULL	Smalltown	64208
9	9	9	666 Spruce St	NULL	Citytown	37589
10	10	10	777 Ash St	NULL	Hometown	90876
11	11	1	123 Main St	NULL	Cityville	12345

Query executed successfully. DESK

- Populating records in the Departments table we use the statement:

```sql
-- Populating Departments Table
INSERT INTO Departments (DepartmentName)
VALUES
('Cardiology'),
('Neurology'),
('Orthopedics'),
('Pediatrics'),
('Oncology'),
('Dermatology'),
('ENT'),
('Gastroenterology'),
('Urology'),
('Ophthalmology');
Go
SELECT * FROM Departments
```

100 %

Results | Messages

	DepartmentID	DepartmentName
1	1	Cardiology
2	2	Neurology
3	3	Orthopedics
4	4	Pediatrics
5	5	Oncology
6	6	Dermatology
7	7	ENT
8	8	Gastroenterology
9	9	Urology
10	10	Ophthalmology
11	11	Cardiology

Query executed successfully.

- Populating records in the Doctors table we use the statement:

```sql
-- Populating Doctors Table
INSERT INTO Doctors (FirstName, LastName, DepartmentID, EmailAddress, TelephoneNumber, Username, Password, Specialty)
VALUES
('Michael', 'Smith', 1, 'michaelsmith@example.com', '123-456-7890', 'michaels', 'pass4321', 'Cardiologist'),
('Emily', 'Johnson', 2, 'emilyjohnson@example.com', '987-654-3210', 'emilyj', 'pass9876', 'Neurologist'),
('David', 'Brown', 3, 'davidbrown@example.com', '456-789-0123', 'davidb', 'pass6543', 'Orthopedic Surgeon'),
('Sarah', 'Wilson', 4, 'sarahwilson@example.com', '789-012-3456', 'sarahw', 'pass2109', 'Pediatrician'),
('Ryan', 'Taylor', 5, 'ryantaylor@example.com', '321-654-9870', 'ryant', 'pass7654', 'Oncologist'),
('Jessica', 'Martinez', 6, 'jessicamartinez@example.com', '654-987-0123', 'jessicam', 'pass0987', 'Dermatologist'),
('Christopher', 'Garcia', 7, 'christophergarcia@example.com', '987-012-3456', 'christopherg', 'pass5432', 'ENT Specialist'),
('Amanda', 'Anderson', 8, 'amandaanderson@example.com', '210-543-8765', 'amandaa', 'pass3210', 'Gastroenterologist'),
('Taylor', 'Hernandez', 9, 'taylorhernandez@example.com', '543-876-2109', 'taylorh', 'pass8765', 'Urologist'),
('Andrew', 'Lopez', 10, 'andrewlopez@example.com', '876-210-5432', 'andrewl', 'pass21098', 'Ophthalmologist');
GO
SELECT * FROM Doctors
```

100 %

Results | Messages

	DoctorID	FirstName	LastName	DepartmentID	EmailAddress	TelephoneNumber	Username	Password	Specialty
1	1	Michael	Smith	1	michaelsmith@example.com	123-456-7890	michaels	pass4321	Cardiologist
2	2	Emily	Johnson	2	emilyjohnson@example.com	987-654-3210	emilyj	pass9876	Neurologist
3	3	David	Brown	3	davidbrown@example.com	456-789-0123	davidb	pass6543	Orthopedic Surgeon
4	4	Sarah	Wilson	4	sarahwilson@example.com	789-012-3456	sarahw	pass2109	Pediatrician
5	5	Ryan	Taylor	5	ryantaylor@example.com	321-654-9870	ryant	pass7654	Oncologist
6	6	Jessica	Martinez	6	jessicamartinez@example.com	654-987-0123	jessicam	pass0987	Dermatologist
7	7	Christopher	Garcia	7	christophergarcia@example.com	987-012-3456	christopherg	pass5432	ENT Specialist
8	8	Amanda	Anderson	8	amandaanderson@example.com	210-543-8765	amandaa	pass3210	Gastroenterologist
9	9	Taylor	Hernandez	9	taylorhernandez@example.com	543-876-2109	taylorh	pass8765	Urologist
10	10	Andrew	Lopez	10	andrewlopez@example.com	876-210-5432	andrewl	pass21098	Ophthalmologist
11	11	Michael	Smith	1	michaelsmith@example.com	123-456-7890	michaels	pass4321	Cardiologist

Query executed successfully. | DESKTOP-MA56CTA (16.0 RTM) | DESKTOP-MA56CTA\Chiche... | HospitalManagementSys

- Populating records in the Appointments table we use the statement:

```sql
-- Populating Appointments Table
INSERT INTO Appointments (PatientID, DoctorID, AppointmentDate, AppointmentTime, Status)
VALUES
(1, 1, '2024-04-15', '09:00:00', 'Pending'),
(2, 2, '2024-04-16', '10:00:00', 'Pending'),
(3, 3, '2024-04-17', '11:00:00', 'Pending'),
(4, 4, '2024-04-18', '12:00:00', 'Pending'),
(5, 5, '2024-04-19', '13:00:00', 'Pending'),
(6, 6, '2024-04-20', '14:00:00', 'Pending'),
(7, 7, '2024-04-21', '15:00:00', 'Pending'),
(8, 8, '2024-04-22', '16:00:00', 'Pending'),
(9, 9, '2024-04-23', '17:00:00', 'Pending'),
(10, 10, '2024-04-24', '18:00:00', 'Pending');
GO
SELECT * FROM Appointments
```

100 %

Results | Messages

	AppointmentID	PatientID	DoctorID	AppointmentDate	AppointmentTime	Status
1	1	1	1	2024-04-15	09:00:00.0000000	Pending
2	2	2	2	2024-04-16	10:00:00.0000000	Pending
3	3	3	3	2024-04-17	11:00:00.0000000	Pending
4	4	4	4	2024-04-18	12:00:00.0000000	Pending
5	5	5	5	2024-04-19	13:00:00.0000000	Pending
6	6	6	6	2024-04-20	14:00:00.0000000	Pending
7	7	7	7	2024-04-21	15:00:00.0000000	Pending
8	8	8	8	2024-04-22	16:00:00.0000000	Pending
9	9	9	9	2024-04-23	17:00:00.0000000	Pending
10	10	10	10	2024-04-24	18:00:00.0000000	Pending
11	11	1	1	2024-04-15	09:00:00.0000000	Pending

Query executed successfully. | DESKTOP-MA56CTA (16.0 RTM) | DESKTOP-M

- Populating records in the MedicalRecords table we use the statement:

```sql
-- Populating MedicalRecords Table
INSERT INTO MedicalRecords (PatientID, Allergies, DateOfVisit)
VALUES
(1, 'Peanuts', '2024-04-10'),
(2, 'Penicillin', '2024-04-11'),
(3, 'Shellfish', '2024-04-12'),
(4, 'Dust', '2024-04-13'),
(5, 'Eggs', '2024-04-14'),
(6, 'Mold', '2024-04-15'),
(7, 'Pollen', '2024-04-16'),
(8, 'Cats', '2024-04-17'),
(9, 'Grass', '2024-04-18'),
(10, 'Insect stings', '2024-04-19');
GO
SELECT * FROM MedicalRecords
```

100 %

Results | Messages

	RecordID	PatientID	DateOfVisit	Allergies
1	1	1	2024-04-10	Peanuts
2	2	2	2024-04-11	Penicillin
3	3	3	2024-04-12	Shellfish
4	4	4	2024-04-13	Dust
5	5	5	2024-04-14	Eggs
6	6	6	2024-04-15	Mold
7	7	7	2024-04-16	Pollen
8	8	8	2024-04-17	Cats
9	9	9	2024-04-18	Grass
10	10	10	2024-04-19	Insect stings
11	11	1	2024-04-10	Peanuts

Query executed successfully. DESKTOP-M

- Populating records in the Updates table we use the statement:

```sql
-- Populating Updates Table
INSERT INTO Updates (RecordID, DoctorID, Diagnoses, Medicines, MedicinePrescribedDate)
VALUES
(1, 1, 'Hypertension', 'Lisinopril', '2024-04-15 09:30:00'),
(2, 2, 'Migraine', 'Sumatriptan', '2024-04-16 10:30:00'),
(3, 3, 'Fractured leg', 'Cast', '2024-04-17 11:30:00'),
(4, 4, 'Strep throat', 'Amoxicillin', '2024-04-18 12:30:00'),
(5, 5, 'Lung cancer', 'Chemotherapy', '2024-04-19 13:30:00'),
(6, 6, 'Eczema', 'Hydrocortisone', '2024-04-20 14:30:00'),
(7, 7, 'Sinusitis', 'Antibiotics', '2024-04-21 15:30:00'),
(8, 8, 'Gastritis', 'Proton pump inhibitors', '2024-04-22 16:30:00'),
(9, 9, 'Kidney stones', 'Pain relievers', '2024-04-23 17:30:00'),
(10, 10, 'Cataracts', 'Surgery', '2024-04-24 18:30:00');
GO
SELECT * FROM Updates
```

100 %

Results | Messages

	UpdateID	RecordID	DoctorID	Diagnoses	Medicines	MedicinePrescribedDate
1	1	1	1	Hypertension	Lisinopril	2024-04-15 09:30:00.000
2	2	2	2	Migraine	Sumatriptan	2024-04-16 10:30:00.000
3	3	3	3	Fractured leg	Cast	2024-04-17 11:30:00.000
4	4	4	4	Strep throat	Amoxicillin	2024-04-18 12:30:00.000
5	5	5	5	Lung cancer	Chemotherapy	2024-04-19 13:30:00.000
6	6	6	6	Eczema	Hydrocortisone	2024-04-20 14:30:00.000
7	7	7	7	Sinusitis	Antibiotics	2024-04-21 15:30:00.000
8	8	8	8	Gastritis	Proton pump inhibitors	2024-04-22 16:30:00.000
9	9	9	9	Kidney stones	Pain relievers	2024-04-23 17:30:00.000
10	10	10	10	Cataracts	Surgery	2024-04-24 18:30:00.000
11	11	1	1	Hypertension	Lisinopril	2024-04-15 09:30:00.000

Query executed successfully. DESKTOP-MA56CTA (16.0 RTM) DESKTOP-

- Populating records in the Availability table we use the statement:

```sql
-- Populating Availability Table
INSERT INTO Availability (DoctorID, Date, TimeSlot, AvailabilityStatus)
VALUES
(1, '2024-04-15', '09:00:00', 'Available'),
(2, '2024-04-16', '10:00:00', 'Available'),
(3, '2024-04-17', '11:00:00', 'Available'),
(4, '2024-04-18', '12:00:00', 'Available'),
(5, '2024-04-19', '13:00:00', 'Available'),
(6, '2024-04-20', '14:00:00', 'Available'),
(7, '2024-04-21', '15:00:00', 'Available'),
(8, '2024-04-22', '16:00:00', 'Available'),
(9, '2024-04-23', '17:00:00', 'Available'),
(10, '2024-04-24', '18:00:00', 'Available');
GO
SELECT * FROM Availability
```

100 %

Results | Messages

	AvailabilityID	DoctorID	Date	TimeSlot	AvailabilityStatus
1	1	1	2024-04-15	09:00:00.0000000	Available
2	2	2	2024-04-16	10:00:00.0000000	Available
3	3	3	2024-04-17	11:00:00.0000000	Available
4	4	4	2024-04-18	12:00:00.0000000	Available
5	5	5	2024-04-19	13:00:00.0000000	Available
6	6	6	2024-04-20	14:00:00.0000000	Available
7	7	7	2024-04-21	15:00:00.0000000	Available
8	8	8	2024-04-22	16:00:00.0000000	Available
9	9	9	2024-04-23	17:00:00.0000000	Available
10	10	10	2024-04-24	18:00:00.0000000	Available
11	11	1	2024-04-15	09:00:00.0000000	Available

Query executed successfully. DESKTOP-MA56CTA

- Populating records in the Reviews table we use the statement:

```sql
-- Populating Reviews Table
INSERT INTO Reviews (DoctorID, PatientID, Review, Rating)
VALUES
(1, 1, 'Great doctor, very informative', 5),
(2, 2, 'Helped me manage my migraines', 4),
(3, 3, 'Excellent care for my fractured leg', 5),
(4, 4, 'Quick recovery from strep throat', 4),
(5, 5, 'Compassionate oncologist', 5),
(6, 6, 'Effective treatment for my eczema', 4),
(7, 7, 'Relieved my sinus infection', 4),
(8, 8, 'Professional and caring gastroenterologist', 5),
(9, 9, 'Knowledgeable urologist', 4),
(10, 10, 'Vision restored after cataract surgery', 5);
GO
SELECT * FROM Reviews
```

100 %

Results | Messages

	ReviewID	DoctorID	PatientID	Review	Rating
1	1	1	1	Great doctor, very informative	5
2	2	2	2	Helped me manage my migraines	4
3	3	3	3	Excellent care for my fractured leg	5
4	4	4	4	Quick recovery from strep throat	4
5	5	5	5	Compassionate oncologist	5
6	6	6	6	Effective treatment for my eczema	4
7	7	7	7	Relieved my sinus infection	4
8	8	8	8	Professional and caring gastroenterologist	5
9	9	9	9	Knowledgeable urologist	4
10	10	10	10	Vision restored after cataract surgery	5
11	11	1	1	Great doctor, very informative	5

Query executed successfully. DESKTO

- Populating records in the Feedbacks table we use the statement:

```sql
-- Populating Feedbacks Table
INSERT INTO Feedbacks (PatientID, Feedback, FeedbackDate)
VALUES
(1, 'Overall, I had a good experience', '2024-04-16 10:30:00'),
(2, 'Satisfied with the service provided', '2024-04-17 11:30:00'),
(3, 'Thank you for the excellent care', '2024-04-18 12:30:00'),
(4, 'Friendly staff and efficient service', '2024-04-19 13:30:00'),
(5, 'Highly recommend this hospital', '2024-04-20 14:30:00'),
(6, 'Improved my quality of life', '2024-04-21 15:30:00'),
(7, 'Grateful for the treatment received', '2024-04-22 16:30:00'),
(8, 'Very pleased with the outcome', '2024-04-23 17:30:00'),
(9, 'Professional and caring staff', '2024-04-24 18:30:00'),
(10, 'Excellent surgical team', '2024-04-25 19:30:00');
GO
SELECT * FROM Feedbacks
```

100 %

Results | Messages

	FeedbackID	PatientID	Feedback	FeedbackDate
1	1	1	Overall, I had a good experience	2024-04-16 10:30:00.000
2	2	2	Satisfied with the service provided	2024-04-17 11:30:00.000
3	3	3	Thank you for the excellent care	2024-04-18 12:30:00.000
4	4	4	Friendly staff and efficient service	2024-04-19 13:30:00.000
5	5	5	Highly recommend this hospital	2024-04-20 14:30:00.000
6	6	6	Improved my quality of life	2024-04-21 15:30:00.000
7	7	7	Grateful for the treatment received	2024-04-22 16:30:00.000
8	8	8	Very pleased with the outcome	2024-04-23 17:30:00.000
9	9	9	Professional and caring staff	2024-04-24 18:30:00.000
10	10	10	Excellent surgical team	2024-04-25 19:30:00.000
11	11	1	Overall, I had a good experience	2024-04-16 10:30:00.000

Query executed successfully. DESKTOP-MA56C

Part 2:

Q2. Adding constraint to ensure appointment date is not in the past

Adding a constraint to check that the appointment date is not in the past, we apply CHECK constraint in the Appointments table.

```sql
-- Q2. Adding constraint to ensure appointment date is not in the past

-- Altering Appointments Table to Add CHECK Constraint for AppointmentDate
ALTER TABLE Appointments
ADD CONSTRAINT CHK_AppointmentDate_NotInPast
CHECK (AppointmentDate >= CAST(GETDATE() AS DATE));
GO
SELECT * FROM Appointments
```

100 %

Results | Messages

	AppointmentID	PatientID	DoctorID	AppointmentDate	AppointmentTime	Status
1	1	1	1	2024-04-15	09:00:00.0000000	Pending
2	2	2	2	2024-04-16	10:00:00.0000000	Pending
3	3	3	3	2024-04-17	11:00:00.0000000	Pending
4	4	4	4	2024-04-18	12:00:00.0000000	Pending
5	5	5	5	2024-04-19	13:00:00.0000000	Pending
6	6	6	6	2024-04-20	14:00:00.0000000	Pending
7	7	7	7	2024-04-21	15:00:00.0000000	Pending
8	8	8	8	2024-04-22	16:00:00.0000000	Pending
9	9	9	9	2024-04-23	17:00:00.0000000	Pending
10	10	10	10	2024-04-24	18:00:00.0000000	Pending
11	11	1	1	2024-04-15	09:00:00.0000000	Pending

Query executed successfully. DESKTOP-MA56CTA (16.0

This constraint ensures that the AppointmentDate cannot be in the past, thus preventing users from scheduling appointments for past dates.

Q3. Patients older than 40 who have a diagnosis of cancer

To list all the patients older than 40 who have been diagnosed with cancer, join the Patients table with the MedicalRecords table and the Updates table to retrieve the

relevant information.

```sql
-- Q3. Patients older than 40 who have a diagnosis of cancer

SELECT DISTINCT P.*
FROM Patients P
JOIN MedicalRecords MR ON P.PatientID = MR.PatientID
JOIN Updates U ON MR.RecordID = U.RecordID
WHERE DATEDIFF(YEAR, P.DateOfBirth, GETDATE()) > 40
AND U.Diagnoses LIKE '%Cancer%';
```

100 %

Results | Messages

	PatientID	FirstName	LastName	AddressID	DateOfBirth	Insurance	Username	Password	DateLeft
1	5	David	Brown	5	1982-07-08	Insurance Group	davidb	abc123	NULL

Query executed successfully. DESKTOP-MA56CTA (16.0 RTM)

In this query:

- We're using the **DATEDIFF** function to calculate the age of each patient by finding the difference in years between the current date (**GETDATE()**) and the patient's date of birth (**P.DateOfBirth**).
- The **JOIN** clauses link the Patients table with the MedicalRecords table and then the MedicalRecords table with the Updates table using their respective IDs.

The **WHERE** clause filters the results to include only patients older than 40 (**DATEDIFF(YEAR, P.DateOfBirth, GETDATE()) > 40**) and those with a diagnosis containing the word "Cancer" (**U.Diagnoses LIKE '%Cancer%'**).

Q4. The Hospital stored procedures and user-defined functions for the specified tasks:

a) Search the database for matching character strings by name of medicine:

```sql
-- a) Search the database for matching character strings by name of medicine:

CREATE PROCEDURE SearchMedicineByName
    @MedicineName NVARCHAR(100)
AS
BEGIN
    SELECT P.FirstName, P.LastName, U.Medicines, U.MedicinePrescribedDate
    FROM Patients P
    JOIN MedicalRecords MR ON P.PatientID = MR.PatientID
    JOIN Updates U ON MR.RecordID = U.RecordID
    WHERE U.Medicines LIKE '%' + @MedicineName + '%'
    ORDER BY U.MedicinePrescribedDate DESC;
END;
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-14T19:17:23.6234201+01:00
```

This query defines a stored procedure named SearchMedicineByName that searches the database for matching character strings within the name of a medicine.

I. **Procedure Definition**:

   CREATE PROCEDURE SearchMedicineByName @MedicineName NVARCHAR(100) declares a stored procedure named **SearchMedicineByName** with a single input parameter **@MedicineName** of type **NVARCHAR(100)**. It specifies the medicine name to search for.

II. **Procedure Body**: **AS BEGIN ... END** encloses the body of the stored procedure.

III. **SELECT Statement**: The **SELECT** statement retrieves data from the database. It selects the **FirstName** and **LastName** columns from the **Patients** table, along with the **Medicines** and **MedicinePrescribedDate** columns from the **Updates** table.

IV. **Joins**: The **FROM** clause specifies the tables involved in the query and the join conditions:

   - **Patients P**: Aliases the **Patients** table as **P**.
   - **MedicalRecords MR**: Aliases the **MedicalRecords** table as **MR**.
   - **Updates U**: Aliases the **Updates** table as **U**.
   - **JOIN** conditions connect the tables based on their primary and foreign key relationships (**P.PatientID = MR.PatientID** and **MR.RecordID = U.RecordID**).

V. **Filtering**: The **WHERE** clause filters the results to include only records where the medicine name (stored in the **Medicines** column of the **Updates** table) contains the specified character string. The **LIKE** operator with **%** wildcards allows for partial matches before and after the search string.

VI. **Ordering**: The **ORDER BY** clause sorts the results based on the **MedicinePrescribedDate** column in descending order, meaning the most recent medicines prescribed will appear first in the result set.

b) Returning a full list of diagnoses and allergies for a specific patient who has an appointment today:

```sql
-- b) Return a full list of diagnoses and allergies for a specific patient who has an appointment today:

CREATE PROCEDURE GetPatientDiagnosisAndAllergiesForToday
    @PatientID INT
AS
BEGIN
    DECLARE @Today DATE = CAST(GETDATE() AS DATE);

    SELECT MR.Allergies, U.Diagnoses
    FROM MedicalRecords MR
    JOIN Updates U ON MR.RecordID = U.RecordID
    JOIN Appointments A ON MR.PatientID = A.PatientID
    WHERE A.AppointmentDate = @Today
    AND MR.PatientID = @PatientID;
END;
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-14T19:47:29.5315621+01:00
```

This stored procedure named **GetPatientDiagnosisAndAllergiesForToday** retrieves the diagnosis and allergies for a specific patient who has an appointment on the current day (the day when the procedure is executed).

Procedure Definition: **CREATE PROCEDURE**

GetPatientDiagnosisAndAllergiesForToday @PatientID INT declares a stored procedure named **GetPatientDiagnosisAndAllergiesForToday** with a single input parameter **@PatientID** of type **INT**. This parameter represents the ID of the patient for whom the diagnosis and allergies are to be retrieved.

i. **Procedure Body**: **AS BEGIN ... END** encloses the body of the stored procedure.

ii. **Variable Declaration**: **DECLARE @Today DATE = CAST(GETDATE() AS DATE);** declares a local variable **@Today** and assigns it the current date obtained using the **GETDATE()** function. The **CAST** function is used to convert the datetime value returned by **GETDATE()** to a date value.

iii. **SELECT Statement**: The **SELECT** statement retrieves data from the database. It selects the **Allergies** column from the **MedicalRecords** table and the **Diagnoses** column from the **Updates** table.

iv. **Joins**: The **FROM** clause specifies the tables involved in the query and the join conditions:

- **MedicalRecords MR**: Aliases the **MedicalRecords** table as **MR**.
- **Updates U**: Aliases the **Updates** table as **U**.
- **Appointments A**: Aliases the **Appointments** table as **A**.
- **JOIN** conditions connect the tables based on their primary and foreign key relationships (**MR.RecordID = U.RecordID** and **MR.PatientID = A.PatientID**).

v. **Filtering**: The **WHERE** clause filters the results to include only records where the appointment date in the **Appointments** table matches the current date (**@Today**) and the patient ID matches the input parameter **@PatientID**.

c) Updating the details for an existing doctor:

```sql
-- c) Update the details for an existing doctor:

CREATE PROCEDURE UpdateDoctorDetails
    @DoctorID INT,
    @NewEmailAddress NVARCHAR(50),
    @NewTelephoneNumber NVARCHAR(15),
    @NewSpecialty NVARCHAR(50)
AS
BEGIN
    UPDATE Doctors
    SET EmailAddress = @NewEmailAddress,
        TelephoneNumber = @NewTelephoneNumber,
        Specialty = @NewSpecialty
    WHERE DoctorID = @DoctorID;
END;
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-14T19:59:14.1230223+01:00
```

This stored procedure named **UpdateDoctorDetails** allows updating the details (email address, telephone number, and specialty) for an existing doctor based on the provided **DoctorID**. Here's a breakdown of the procedure:

i. **Procedure Definition**: **CREATE PROCEDURE UpdateDoctorDetails @DoctorID INT, @NewEmailAddress NVARCHAR(50), @NewTelephoneNumber NVARCHAR(15), @NewSpecialty NVARCHAR(50)** declares a stored procedure named **UpdateDoctorDetails** with four input parameters:

   - **@DoctorID**: The ID of the doctor whose details are to be updated.
   - **@NewEmailAddress**: The new email address to be assigned to the doctor.
   - **@NewTelephoneNumber**: The new telephone number to be assigned to the doctor.
   - **@NewSpecialty**: The new specialty to be assigned to the doctor.

ii. **Procedure Body**: **AS BEGIN ... END** encloses the body of the stored procedure.

iii. **UPDATE Statement**: The **UPDATE** statement modifies data in the **Doctors** table. It sets the **EmailAddress**, **TelephoneNumber**, and **Specialty** columns to the new values provided as input parameters.

iv. **Filtering**: The **WHERE** clause specifies the condition for updating records. In this case, it updates the details only for the doctor whose **DoctorID** matches the input parameter **@DoctorID**.

d) Delete the appointment whose status is already completed:

```sql
-- d) Delete the appointment whose status is already completed:

CREATE PROCEDURE DeleteCompletedAppointments
AS
BEGIN
    DELETE FROM Appointments
    WHERE Status = 'Completed';
END;
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-14T20:15:45.9452628+01:00
```

This stored procedure named **DeleteCompletedAppointments** allows deleting appointments that have already been completed.

i. **Procedure Definition**: **CREATE PROCEDURE DeleteCompletedAppointments** declares a stored procedure named **DeleteCompletedAppointments** without any input parameters.

ii. **Procedure Body**: **AS BEGIN ... END** encloses the body of the stored procedure.

iii. **DELETE Statement**: The **DELETE** statement removes records from the **Appointments** table. It deletes appointments where the value of the **Status** column is **'Completed'**.

iv. **Filtering**: The **WHERE** clause specifies the condition for deleting records. In this case, it deletes appointments only if their **Status** is **'Completed'**.

Q5. Creating a view that displays appointment details along with information about the doctor's department, specialty, and any associated reviews/feedback.

```sql
-- Q5. Creating a view that displays appointment details along with information about the doctor's department, specialt
-- the following SQL query can be used:

CREATE VIEW AllAppointmentsDetails AS
SELECT A.AppointmentID,
       A.AppointmentDate,
       A.AppointmentTime,
       D.FirstName + ' ' + D.LastName AS DoctorName,
       DP.DepartmentName AS Department,
       D.Specialty AS DoctorSpecialty,
       R.Review AS DoctorReview,
       R.Rating AS DoctorRating
FROM Appointments A
JOIN Doctors D ON A.DoctorID = D.DoctorID
JOIN Departments DP ON D.DepartmentID = DP.DepartmentID
LEFT JOIN Reviews R ON D.DoctorID = R.DoctorID;
SELECT * FROM AllAppointmentsDetails
```

100 %

Results | Messages

	AppointmentID	AppointmentDate	AppointmentTime	DoctorName	Department	DoctorSpecialty	DoctorReview	DoctorRating
1	1	2024-04-15	09:00:00.0000000	Michael Smith	Cardiology	Cardiologist	Great doctor, very informative	5
2	1	2024-04-15	09:00:00.0000000	Michael Smith	Cardiology	Cardiologist	Great doctor, very informative	5
3	2	2024-04-16	10:00:00.0000000	Emily Johnson	Neurology	Neurologist	Helped me manage my migraines	4
4	2	2024-04-16	10:00:00.0000000	Emily Johnson	Neurology	Neurologist	Helped me manage my migraines	4
5	3	2024-04-17	11:00:00.0000000	David Brown	Orthopedics	Orthopedic Surgeon	Excellent care for my fractured leg	5
6	3	2024-04-17	11:00:00.0000000	David Brown	Orthopedics	Orthopedic Surgeon	Excellent care for my fractured leg	5
7	4	2024-04-18	12:00:00.0000000	Sarah Wilson	Pediatrics	Pediatrician	Quick recovery from strep throat	4
8	4	2024-04-18	12:00:00.0000000	Sarah Wilson	Pediatrics	Pediatrician	Quick recovery from strep throat	4
9	5	2024-04-19	13:00:00.0000000	Ryan Taylor	Oncology	Oncologist	Compassionate oncologist	5
10	5	2024-04-19	13:00:00.0000000	Ryan Taylor	Oncology	Oncologist	Compassionate oncologist	5
11	6	2024-04-20	14:00:00.0000000	Jessica Martinez	Dermatology	Dermatologist	Effective treatment for my eczema	4

Query executed successfully. | DESKTOP-MA56CTA (16.0 RTM) | DESKTOP-MA56CTA\Chiche... | HospitalManage

In this view:

- Select columns from the Appointments table (**AppointmentID**, **AppointmentDate**, **AppointmentTime**) along with the doctor's name (**FirstName** and **LastName** concatenated), the department name, the doctor's specialty, and any associated review and rating.
- The view joins the Appointments table with the Doctors table to get doctor details and the Departments table to get department details.

- Use a LEFT JOIN with the Reviews table to include any reviews associated with the doctor, allowing for appointments without reviews.

This view will give a comprehensive list of appointment details for all doctors, including department, specialty, and any associated reviews or feedback.

Q6. Creating a trigger to automatically change the current state of an appointment to "Available" when it is canceled

The below SQL code can be used:

```sql
-- Q6. Creating a trigger to automatically change the current state of an appointment to "Available" when it is canceled.

CREATE TRIGGER UpdateAppointmentStatus
ON Appointments
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Status) -- Check if the Status column was updated
    BEGIN
        UPDATE Appointments
        SET Status = 'Available'
        FROM inserted
        WHERE Appointments.AppointmentID = inserted.AppointmentID
        AND inserted.Status = 'Cancelled';
    END
END;
```

i. **CREATE TRIGGER UpdateAppointmentStatus ON Appointments AFTER UPDATE**: This creates a trigger named **UpdateAppointmentStatus** on the **Appointments** table. It specifies that the trigger should fire after an update operation is performed on the table.

ii. **IF UPDATE(Status)**: This checks if the **Status** column was updated during the update operation.

iii. **BEGIN ... END**: This block of code is executed if the **Status** column was updated.

iv. **UPDATE Appointments SET Status = 'Available' FROM inserted**: This line updates the **Status** column of the **Appointments** table to 'Available'. It selects rows from the **inserted** pseudo-table, which contains the new values that were updated.

v. **WHERE Appointments.AppointmentID = inserted.AppointmentID AND inserted.Status = 'Cancelled'**: Conditions for updating the **Status** column is specified. It updates the status to 'Available' only for appointments that were cancelled.

With this trigger in place, whenever an appointment's status is updated to 'Cancelled', the trigger will automatically change the status to 'Available'.

Q7. Identifying the number of completed appointments with the specialty of doctors as 'Gastroenterologists',

```sql
-- Q7. identifying the number of completed appointments with the specialty of doctors as 'Gastroenterologists',
-- The following SQL query is used:

SELECT COUNT(*) AS CompletedAppointments
FROM Appointments A
JOIN Doctors D ON A.DoctorID = D.DoctorID
JOIN Departments DP ON D.DepartmentID = DP.DepartmentID
WHERE A.Status = 'Completed'
AND D.Specialty = 'Gastroenterologists';
```

100 %

Results | Messages

	CompletedAppointments
1	0

Here's a breakdown of the query:

i. **FROM Clause**: The query starts by selecting data from the **Appointments** table (**FROM Appointments A**).

ii. **JOIN Clauses**: Joins the **Doctors** table (**JOIN Doctors D ON A.DoctorID = D.DoctorID**) to get doctor details and the **Departments** table (**JOIN Departments DP ON D.DepartmentID = DP.DepartmentID**) to get department details.

iii. **WHERE Clause**: The **WHERE** clause filters the appointments to include only those that are completed (**A.Status = 'Completed'**) and where the doctor's specialty is 'Gastroenterologists' (**D.Specialty = 'Gastroenterologists'**).

iv. **SELECT Clause**: The **SELECT** clause calculates the count of completed appointments (**COUNT(*) AS CompletedAppointments**).

This query will return the total number of completed appointments with doctors specializing in 'Gastroenterologists'.

Additional Recommendations

i. **Data Integrity and Concurrency**:

- The task solution includes the implementation of constraints such as primary keys, foreign keys, and check constraints to enforce data integrity rules at the database level. For example, foreign key constraints are used to maintain referential integrity between related tables like Patients, Addresses, Doctors, etc.
- Recommendations on concurrency control are indirectly addressed through the use of transactions in stored procedures and triggers. For example, when updating the status of an appointment in the

UpdateAppointmentStatus trigger, concurrency issues are mitigated by ensuring that the status is changed atomically.

ii. **Database Security**:

- The solution employs basic user authentication with usernames and passwords for patients and doctors. However, enhancing security with multi-factor authentication would be advantageous.

- Encryption should be implemented to safeguard sensitive data, particularly patient information and medical records.

- Also, incorporating database auditing for tracking user activities and implementing fine-grained access controls based on roles and privileges would enhance security and accountability.

iii. **Database Backup and Recovery**:

Database backup and recovery mechanisms are enclosed as part of the zip file submitted.

Conclusions:

The Hospital Management System project delivered a database solution to meet hospital needs efficiently. It covers patient information, medical records, appointments, doctors, and departments, integrating key functions like appointment scheduling and patient feedback. The design covers normalization principles, ensuring data integrity. However, there's room for enhancement in implementing encryption, audit trails, and access controls for improved security. A backup and recovery strategy is essential for data resilience, which should be implemented to safeguard patient data and ensure business continuity.

TASK 2

Introduction

The objective is to create a database for a food service company based on provided CSV files containing information about restaurants, consumers, ratings, and restaurant cuisines.

The dataset consists of four related tables: Restaurants, Consumers, Ratings, and Restaurant_Cuisines.

To tackle this task, perform the following steps:

1. Create a database named FoodserviceDB.
2. Import the four CSV files into separate tables in the database.
3. Ensure the appropriate primary and foreign key constraints are added to maintain data integrity.
4. Provide a database diagram illustrating the relationships between the tables.

Creating a database:

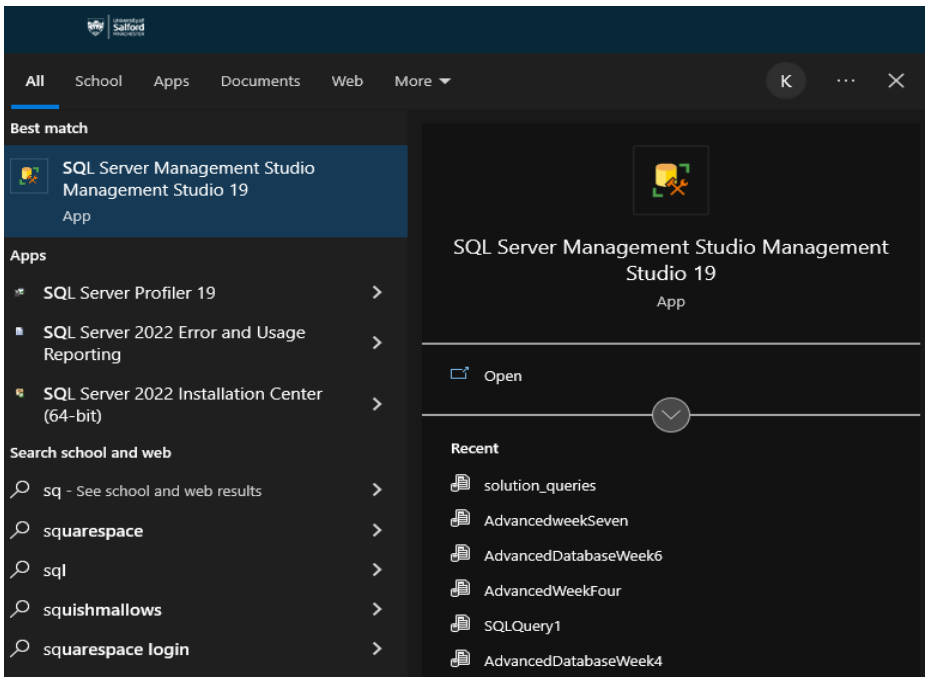


Fig 1.1.1

Searching for “SQL Server Managenment Studio”

Launch SSMS by typing the word "SQL Server Management Studio" into the Windows search box.

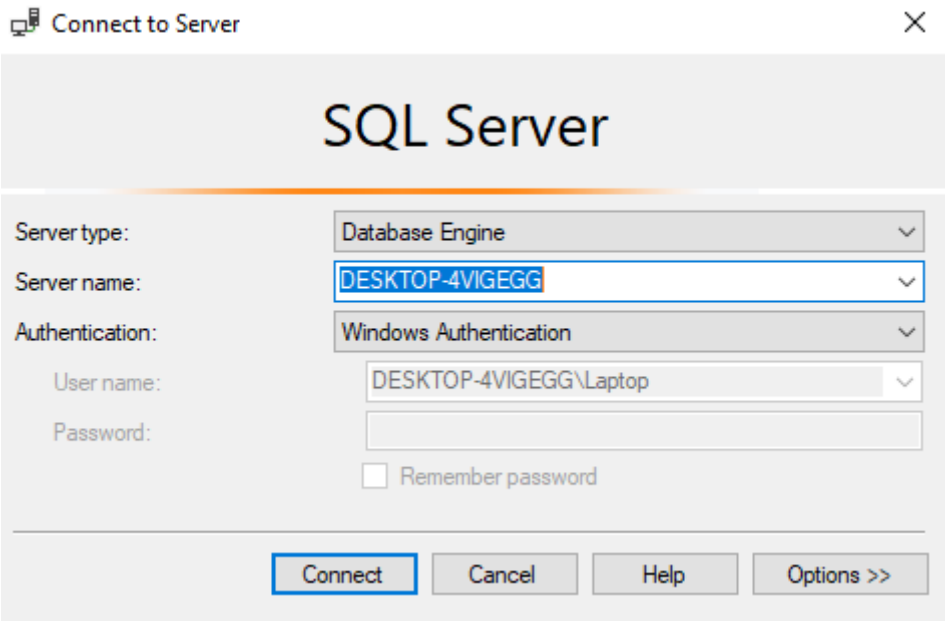


Fig 1.1.2

Connecting to the server

Connect to the SQL server by clicking on the connect button.

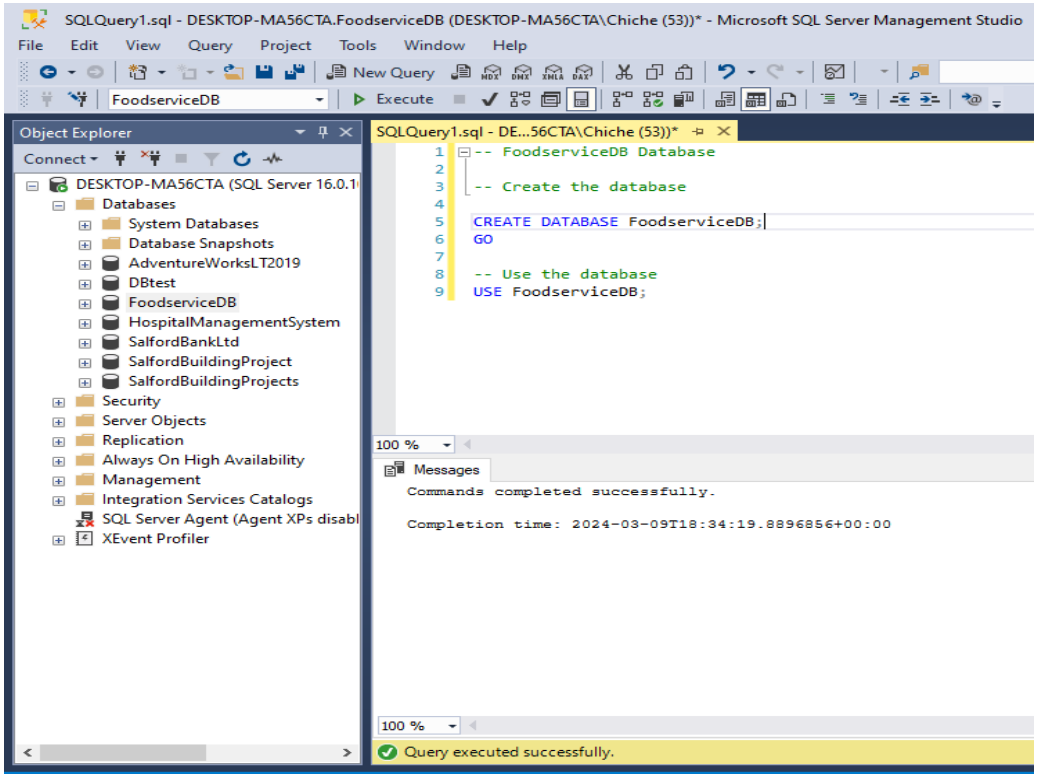
Creating the database using the CREATE DATABASE statement below:

-- Create the database

CREATE DATABASE FoodserviceDB;

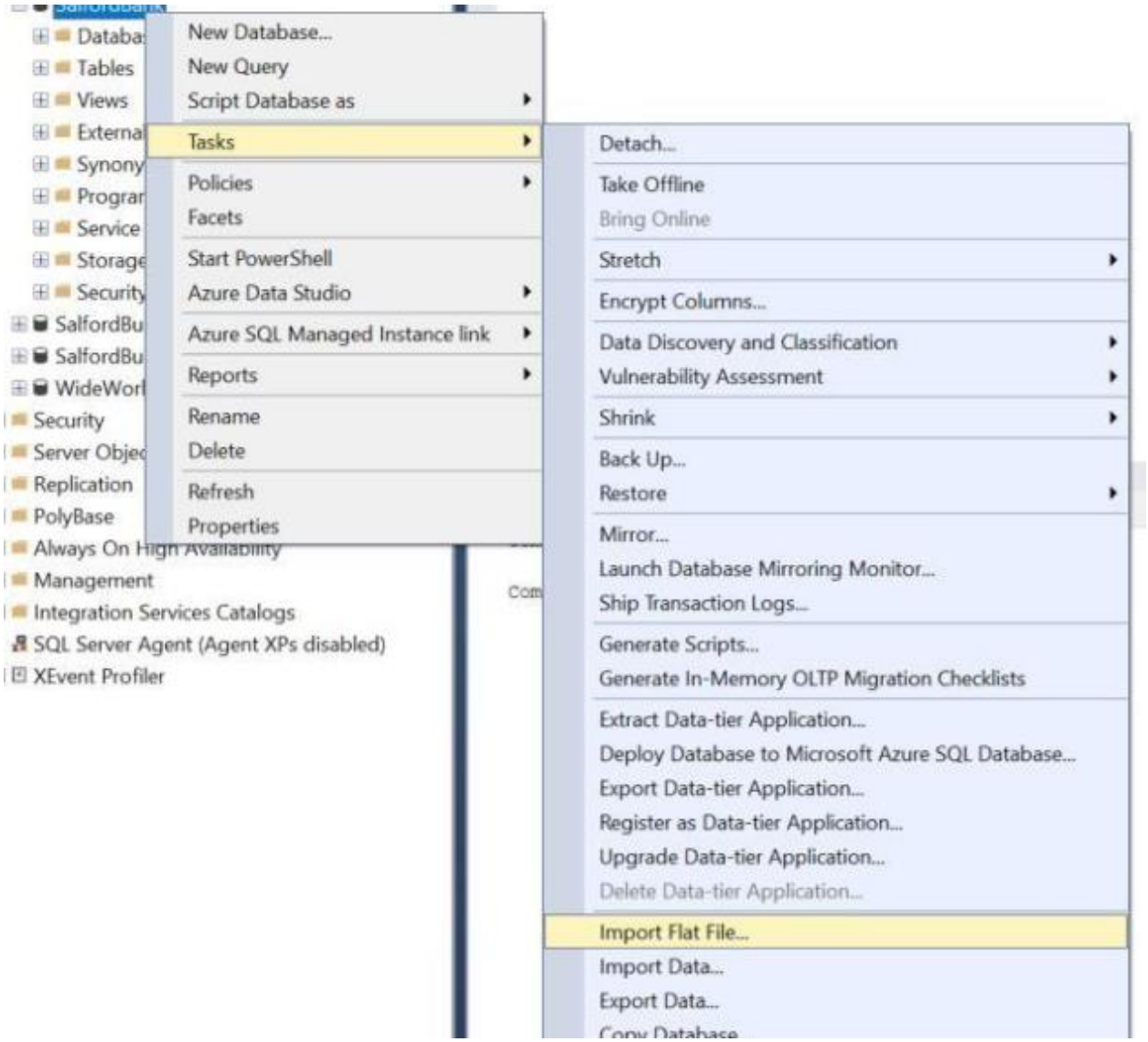
The below command is use to set the newly created FoodserviceDB database:

USE FoodserviceDB; GO

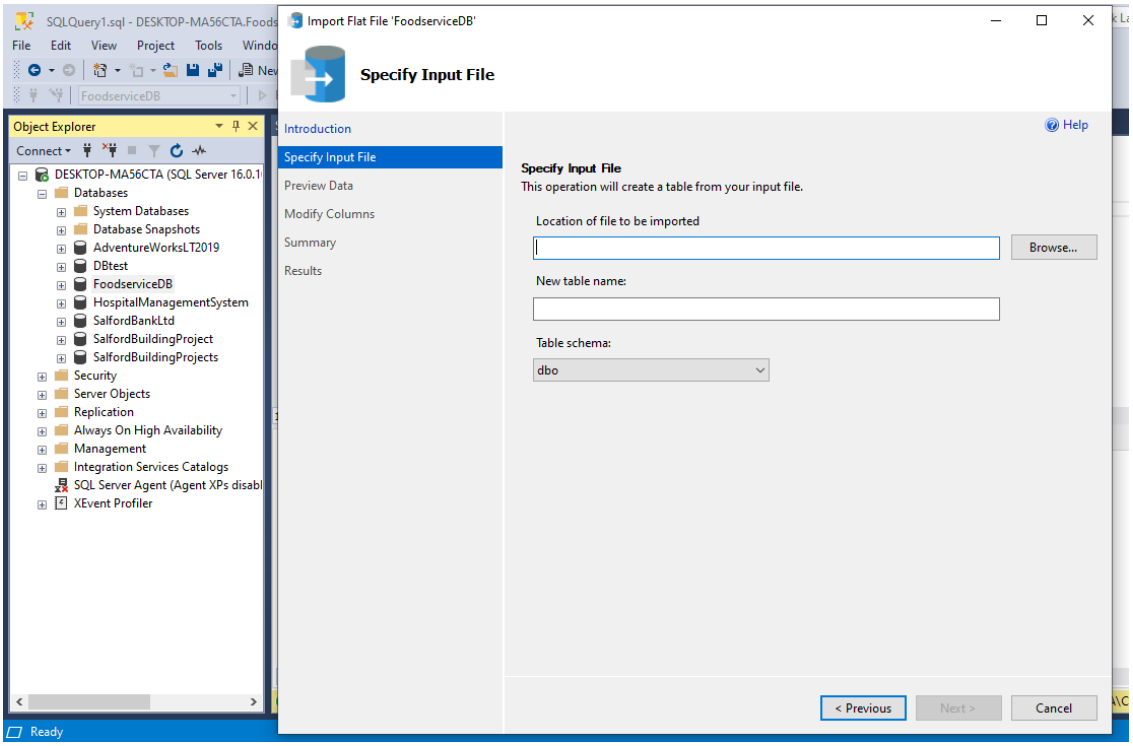


1. Importing Flat Files:

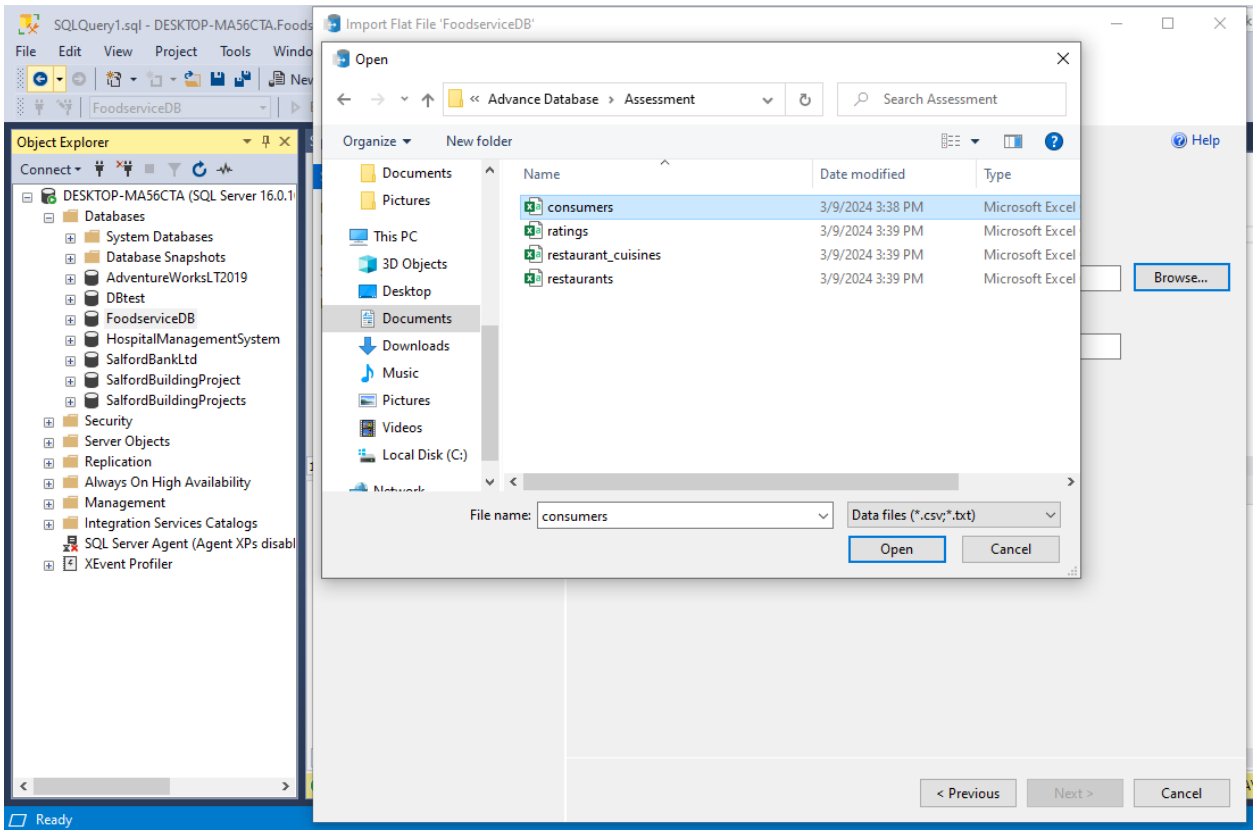
I achieved this by right clicking on the database, selecting Tasks from the menu and then select Import Flat File.

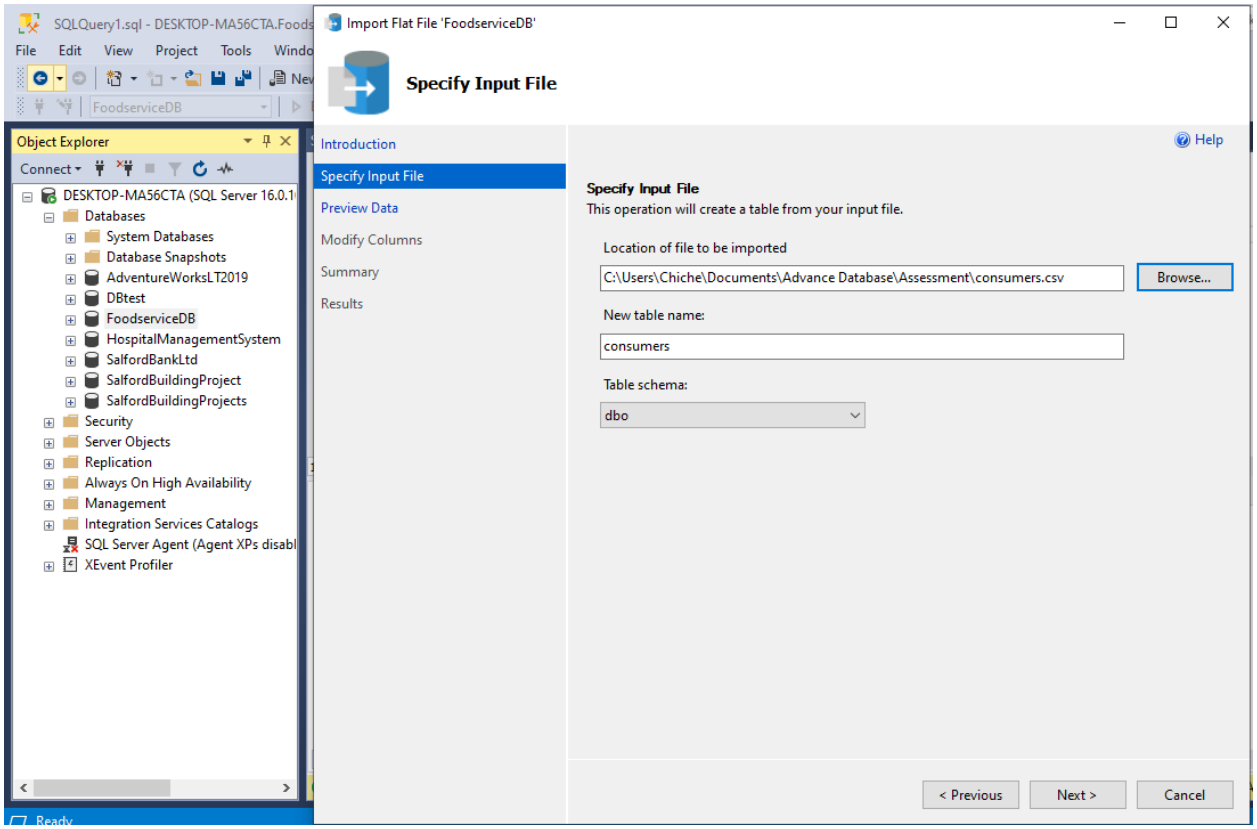


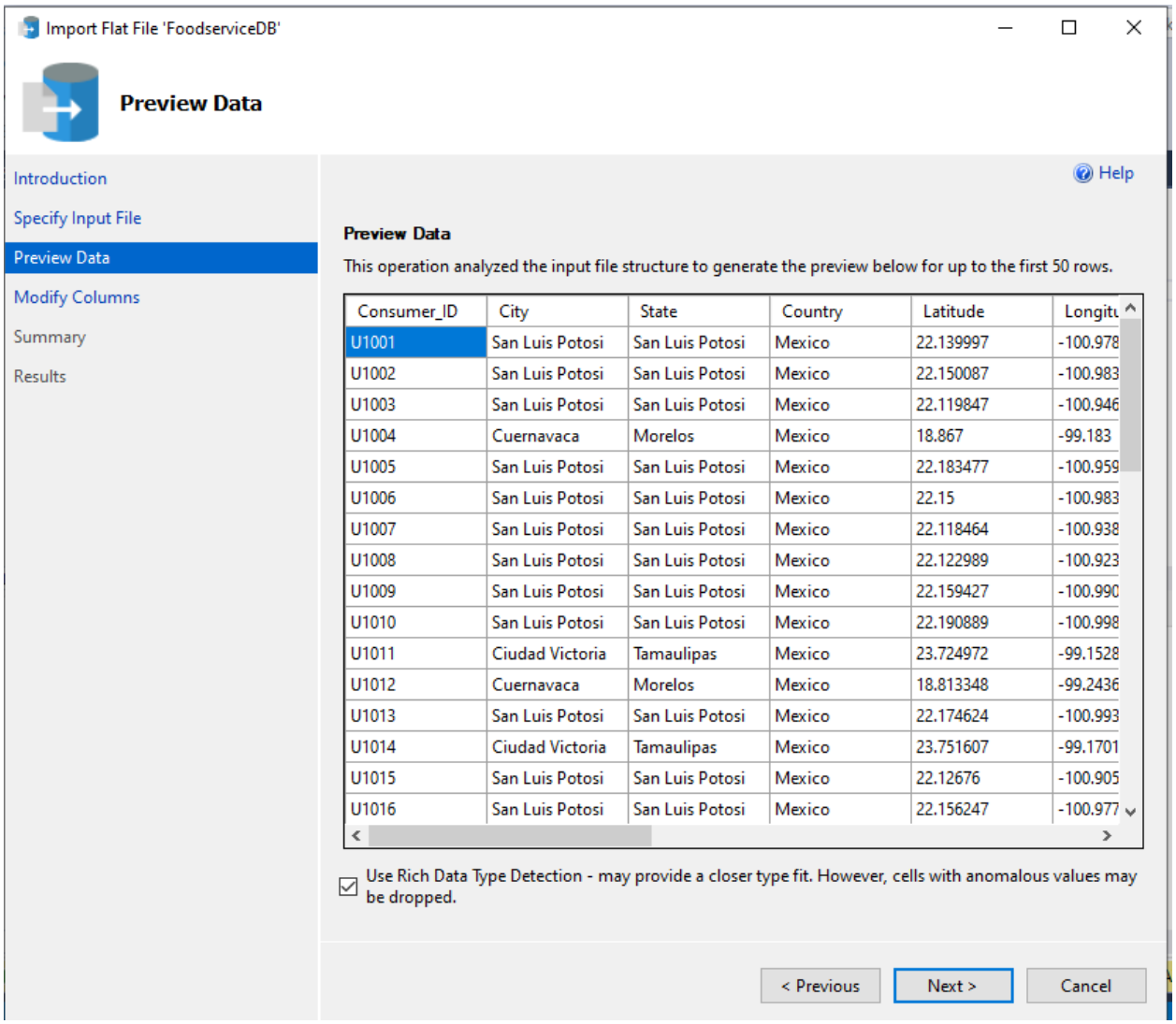
Click on Browse and then navigate to where I have saved the csv file downloaded from Blackboard.



Select Next. Check the preview of the file imported. Leave ‘Use Rich Data Type Detection’ ticked and click Next again.

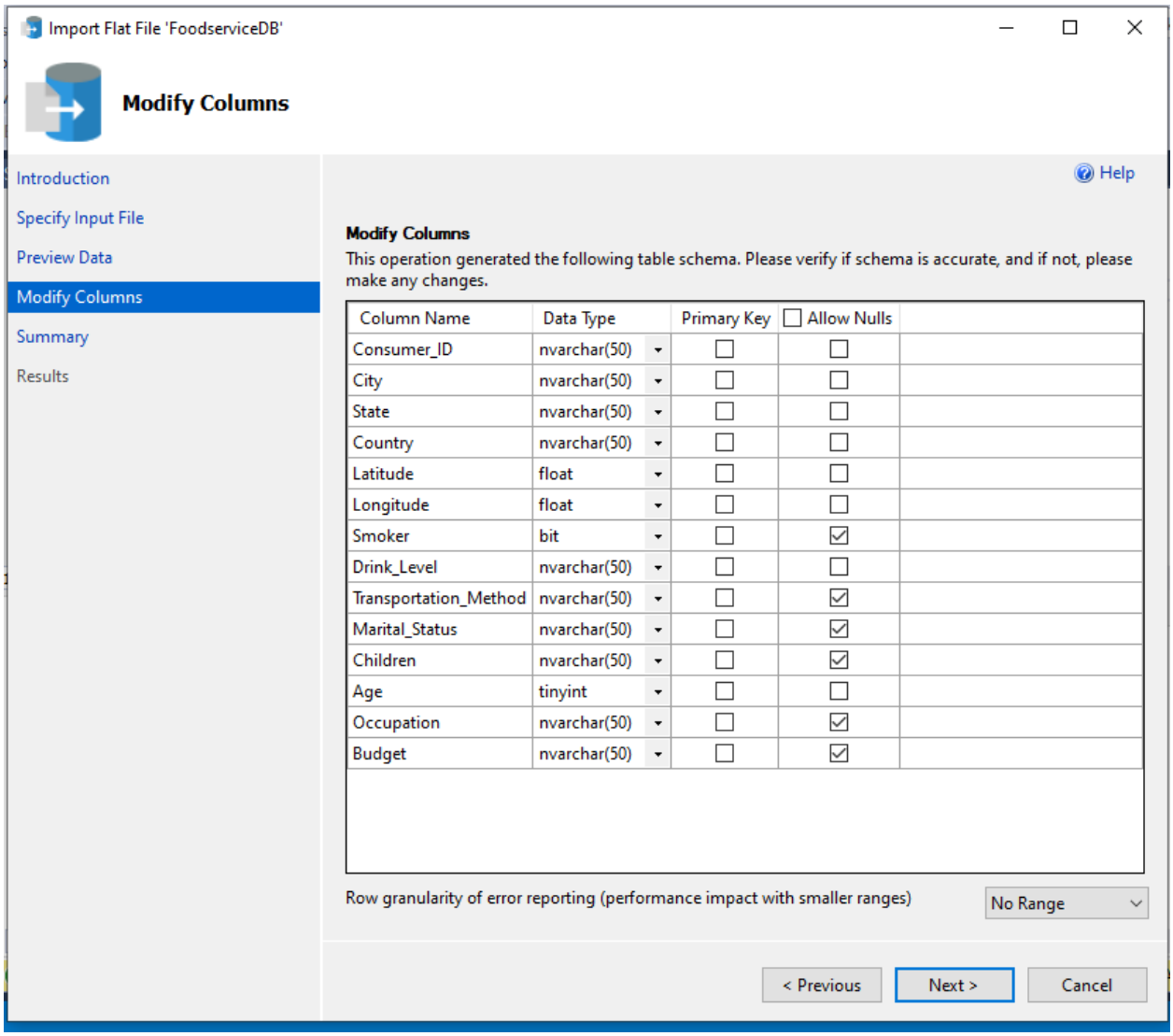

SQLQuery1.sql - DESKTOP-MA56CTA.Foods
File Edit View Project Tools Windo
FoodserviceDB
Object Explorer
Connect
DESKTOP-MA56CTA (SQL Server 16.0.1
Databases
System Databases
Database Snapshots
AdventureWorksLT2019
DBtest
FoodserviceDB
HospitalManagementSystem
SalfordBankLtd
SalfordBuildingProject
SalfordBuildingProjects
Security
Server Objects
Replication
Always On High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs disabl
XEvent Profiler
Ready
Import Flat File 'FoodserviceDB'
Open
Advance Database › Assessment
Search Assessment
Organize
New folder
Documents
Pictures
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
Name
Date modified
Type
consumers 3/9/2024 3:38 PM Microsoft Excel
ratings 3/9/2024 3:39 PM Microsoft Excel
restaurant_cuisines 3/9/2024 3:39 PM Microsoft Excel
restaurants 3/9/2024 3:39 PM Microsoft Excel
File name: consumers
Data files (*.csv;*.txt)
Open
Cancel
Help
Browse...
< Previous
Next >
Cancel

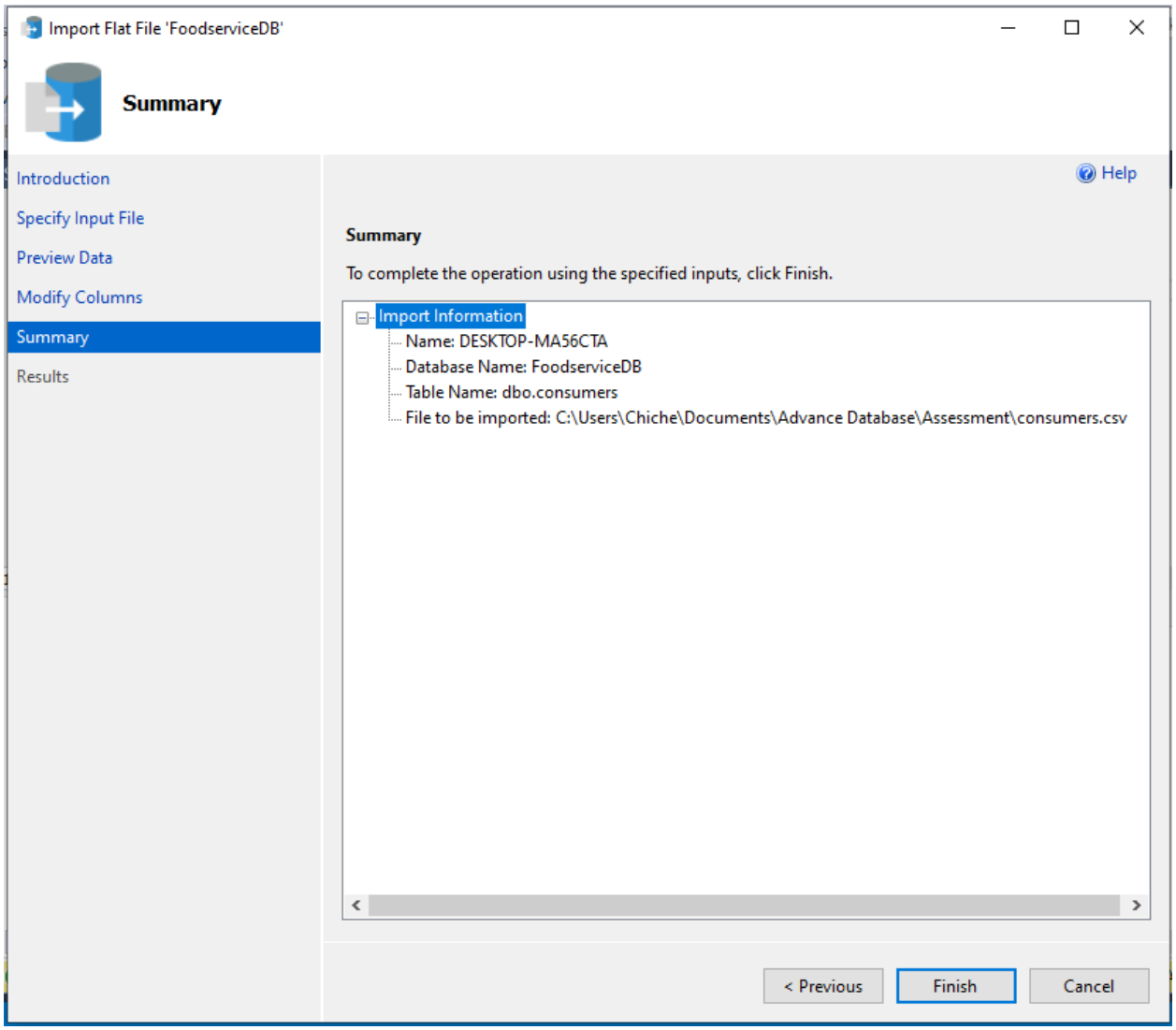

SQLQuery1.sql - DESKTOP-MA56CTA.Foods
File Edit View Project Tools Windo
FoodserviceDB
Object Explorer
Connect
DESKTOP-MA56CTA (SQL Server 16.0.1
Databases
System Databases
Database Snapshots
AdventureWorksLT2019
DBtest
FoodserviceDB
HospitalManagementSystem
SalfordBankLtd
SalfordBuildingProject
SalfordBuildingProjects
Security
Server Objects
Replication
Always On High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs disabl
XEvent Profiler
Ready
Import Flat File 'FoodserviceDB'
Specify Input File
Introduction
Specify Input File
Preview Data
Modify Columns
Summary
Results
Help
Specify Input File
This operation will create a table from your input file.
Location of file to be imported
C:\Users\Chiche\Documents\Advance Database\Assessment\consumers.csv
Browse...
New table name:
consumers
Table schema:
dbo
< Previous
Next >
Cancel



Consumer_ID	City	State	Country	Latitude	Longitu
U1001	San Luis Potosi	San Luis Potosi	Mexico	22.139997	-100.978
U1002	San Luis Potosi	San Luis Potosi	Mexico	22.150087	-100.983
U1003	San Luis Potosi	San Luis Potosi	Mexico	22.119847	-100.946
U1004	Cuernavaca	Morelos	Mexico	18.867	-99.183
U1005	San Luis Potosi	San Luis Potosi	Mexico	22.183477	-100.959
U1006	San Luis Potosi	San Luis Potosi	Mexico	22.15	-100.983
U1007	San Luis Potosi	San Luis Potosi	Mexico	22.118464	-100.938
U1008	San Luis Potosi	San Luis Potosi	Mexico	22.122989	-100.923
U1009	San Luis Potosi	San Luis Potosi	Mexico	22.159427	-100.990
U1010	San Luis Potosi	San Luis Potosi	Mexico	22.190889	-100.998
U1011	Ciudad Victoria	Tamaulipas	Mexico	23.724972	-99.1528
U1012	Cuernavaca	Morelos	Mexico	18.813348	-99.2436
U1013	San Luis Potosi	San Luis Potosi	Mexico	22.174624	-100.993
U1014	Ciudad Victoria	Tamaulipas	Mexico	23.751607	-99.1701
U1015	San Luis Potosi	San Luis Potosi	Mexico	22.12676	-100.905
U1016	San Luis Potosi	San Luis Potosi	Mexico	22.156247	-100.977

Check the data types it has identified for each of the columns and allow all.


Import Flat File 'FoodserviceDB'
Modify Columns
Introduction
Specify Input File
Preview Data
Modify Columns
Summary
Results
Help
Modify Columns
This operation generated the following table schema. Please verify if schema is accurate, and if not, please make any changes.
Column Name | Data Type | Primary Key | Allow Nulls
Consumer_ID | nvarchar(50)
City | nvarchar(50)
State | nvarchar(50)
Country | nvarchar(50)
Latitude | float
Longitude | float
Smoker | bit
Drink_Level | nvarchar(50)
Transportation_Method | nvarchar(50)
Marital_Status | nvarchar(50)
Children | nvarchar(50)
Age | tinyint
Occupation | nvarchar(50)
Budget | nvarchar(50)
Row granularity of error reporting (performance impact with smaller ranges)
No Range
< Previous
Next >
Cancel



1. After the import has been completed, the execution status indicates that the import has been successful then click on the close button

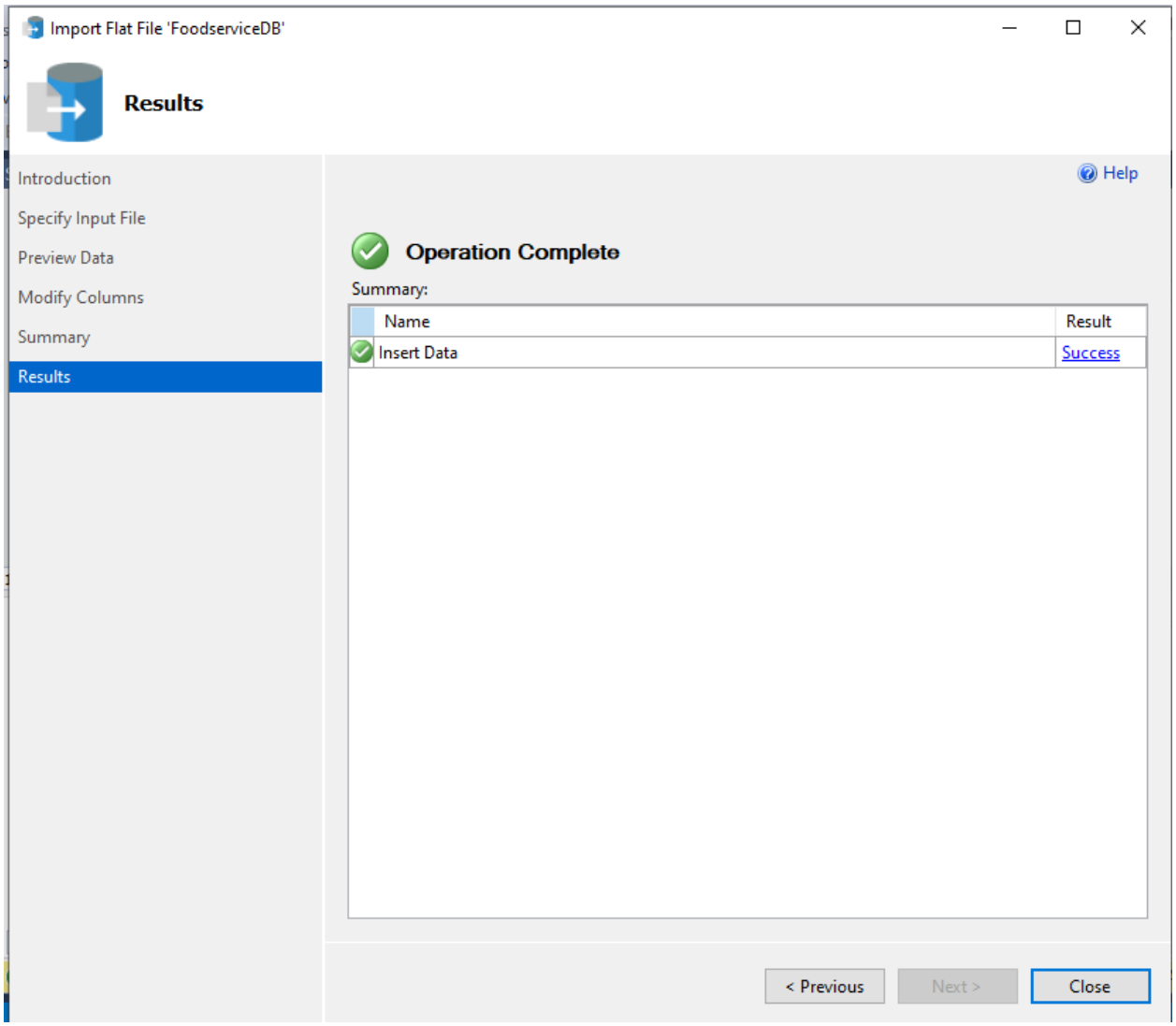

Import Flat File 'FoodserviceDB'
Results
Introduction
Specify Input File
Preview Data
Modify Columns
Summary
Results
Help
Operation Complete
Summary:
Name
Result
Insert Data
Success
< Previous
Next >
Close

Fig 1.1.11

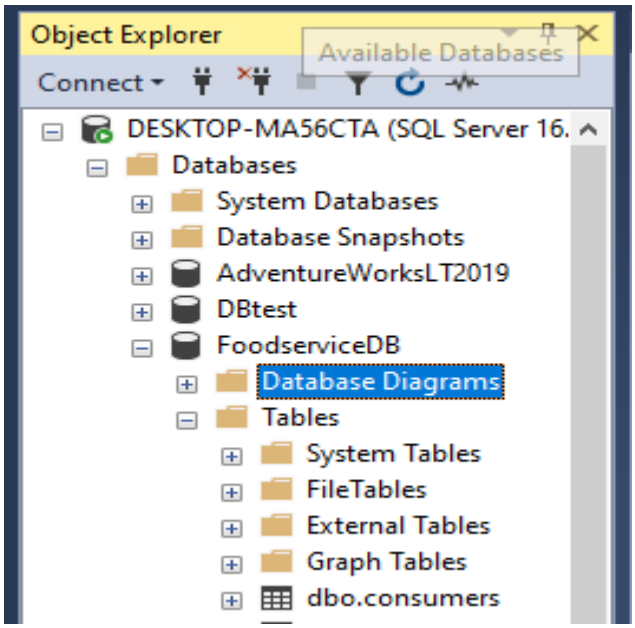


Fig 1.1.12

As a result, the table has been imported as dbo.customers.

Following the same procedure, import the three other tables (restaurants, ratings and restaurant_cuisines) in the same manner.

Primary and Foreign key Constraints:

```sql
-- Add primary key constraint to Restaurant table
ALTER TABLE restaurants
ADD PRIMARY KEY (Restaurant_ID);

-- Add primary key constraint to Consumers table
ALTER TABLE consumers
ADD PRIMARY KEY (Consumer_ID);

-- Add primary key constraint to Ratings table
ALTER TABLE ratings
ADD CONSTRAINT PK_ratings PRIMARY KEY (Consumer_ID, Restaurant_ID);

-- Add foreign key constraint to Ratings table referencing Consumers table
ALTER TABLE ratings
ADD CONSTRAINT FK_Consumer_ID FOREIGN KEY (Consumer_ID) REFERENCES consumers(Consumer_ID);

-- Add foreign key constraint to Ratings table referencing Restaurant table
ALTER TABLE ratings
ADD CONSTRAINT FK_Restaurant_ID FOREIGN KEY (Restaurant_ID) REFERENCES restaurants(Restaurant_ID);

-- Add primary key constraint to Restaurant_Cuisines table
ALTER TABLE restaurant_cuisines
ADD CONSTRAINT PK_restaurant_cuisines PRIMARY KEY (Restaurant_ID, Cuisine);

-- Add foreign key constraint to Restaurant_Cuisines table referencing Restaurant table
ALTER TABLE restaurant_cuisines
ADD CONSTRAINT FK_Restaurant_ID_Cuisines FOREIGN KEY (Restaurant_ID) REFERENCES restaurants(Restaurant_ID);
```

Fig 1.2.1

Primary and Foreign key Constraints

Creating relationships among tables is an essential aspect of database design, and it involves defining the primary keys and foreign keys in the tables. A primary key is a unique identifier for a record in a table, while a foreign key is a field in one table that refers to the primary key in another table.

To create a relationship between two tables, specify the foreign key in the child table that refers to the primary key in the parent table. It's done using the ALTER TABLE statement in SQL. Once the relationship is established, it can be used to retrieve data from multiple tables using JOIN queries.

With these ALTER TABLE commands, primary key constraints are added to the Restaurants, Consumers, Ratings, and Restaurant_Cuisines tables. Foreign key constraints are also added to ensure referential integrity between the Ratings table and the Consumers and Restaurant tables, and between the Restaurant_Cuisines table and the Restaurant table.

Run the ALTER queries once to create the relationships among the tables in your database. Visualize the database schema under the database diagram in the SQL server GUI interface to ensure that the relationships are correctly established.

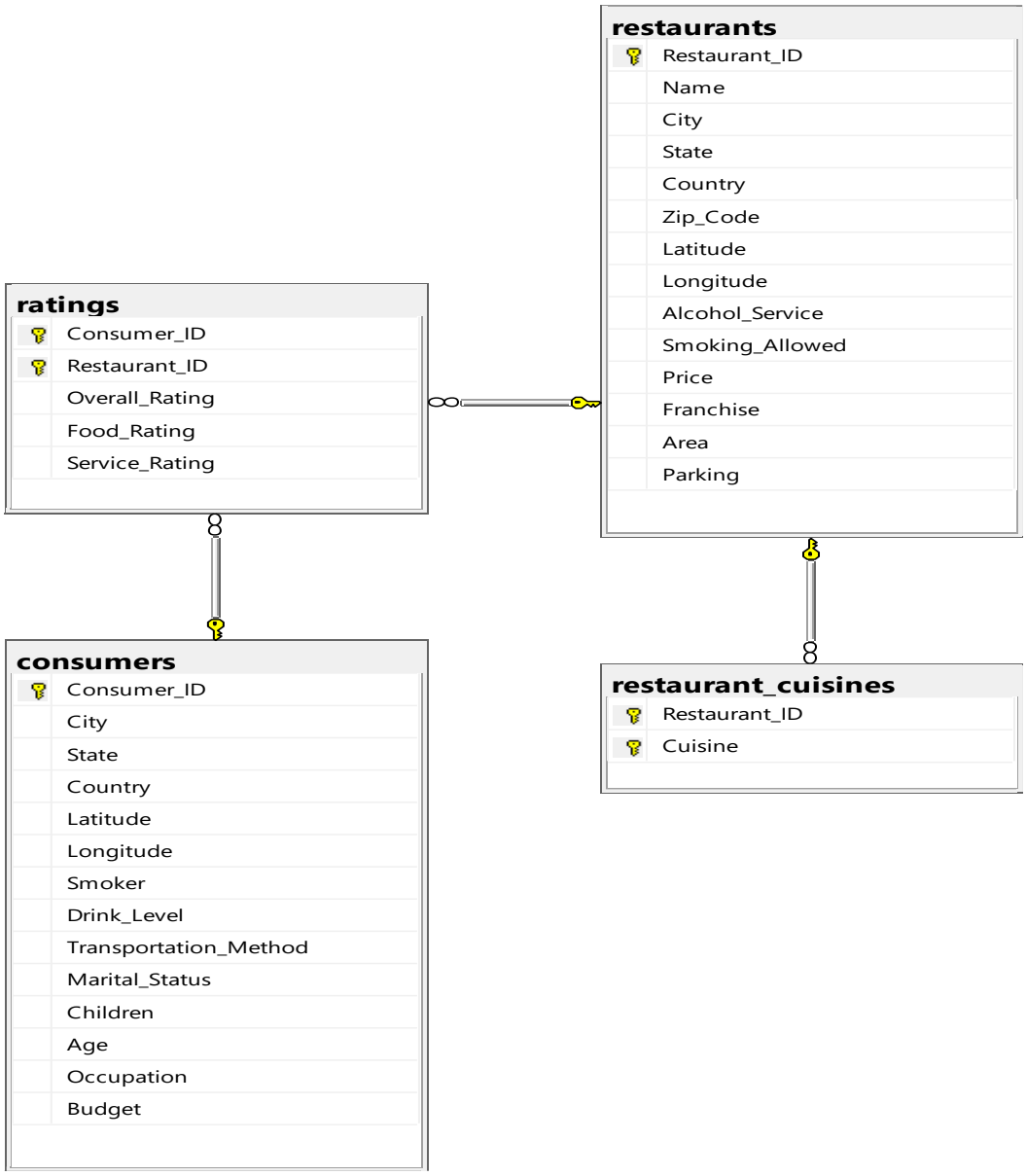


Fig 1.2.2

Database Diagram

In a relational database, tables are related to each other based on common fields or columns. The "Restaurants" table serves as the main repository for restaurant information, with the "Restaurant_ID" column acting as the primary key, ensuring each restaurant has a unique identifier within the table. This "Restaurant_ID" column is referenced in the "Ratings" table as a foreign key, establishing a relationship between restaurants and ratings. Additionally, the "Ratings" table also references the "Consumers" table through the "Consumer_ID" foreign key, indicating which consumers provided ratings for various restaurants.

Furthermore, the "Restaurant Cuisines" table is linked to the "Restaurants" table via the "Restaurant_ID" column. This relationship signifies the types of cuisines offered by each restaurant.

It's important to choose primary keys that have unique values and are not null to ensure the integrity and consistency of your data. By using primary and foreign keys correctly, you can maintain the relationships between tables in your database, allowing for accurate and reliable data management, enabling the retrieval of interconnected data using JOIN queries, facilitating comprehensive analysis and reporting across multiple tables within the database.

Part 2:

Q1. To retrieve all restaurants with a medium range price, open area, and serving Mexican food, use the following SQL query:

```sql
-- 1. retrieving all restaurants with a Medium range price, open area, and serving Mexican food

SELECT *
FROM Restaurants r
INNER JOIN Restaurant_Cuisines rc ON r.Restaurant_id = rc.Restaurant_id
WHERE r.Price = 'Medium'
AND r.Area = 'Open'
AND rc.Cuisine = 'Mexican';
```

100 %

Results | Messages

	Restaurant_ID	Name	City	State	Country	Zip_Code	Latitude	Longitude	Alcohol_Service	Smoking_Allowed	Price	Franchise
1	135018	El Oceano Dorado	Cuernavaca	Morelos	Mexico	NULL	18.8598022460938	-99.2221603393555	Full Bar	Yes	Medium	0
2	135106	El Rincón De San Francisco	San Luis Potosi	San Luis Potosi	Mexico	78000	22.1497097015381	-100.976089477539	Wine & Beer	Bar Only	Medium	0

Query Explanation:

- **SELECT * FROM Restaurants r INNER JOIN Restaurant_Cuisines rc ON r.Restaurant_id = rc.Restaurant_id**:
    - This part of the query selects all columns from the 'Restaurants' table and joins it with the 'Restaurant_Cuisines' table based on the 'Restaurant_id' column. This join ensures that we get restaurants serving Mexican cuisine.
- **WHERE r.Price = 'Medium' AND r.Area = 'Open' AND rc.Cuisine = 'Mexican'**:
    - This part of the query filters the results to include only restaurants with a medium range price, open area, and serving Mexican food.
    - **r.Price = 'Medium'** ensures that only restaurants with a Medium price range are included.

- **r.Area = 'Open'** filters the results to include only restaurants with an open area.
- **rc.Cuisine = 'Mexican'** ensures that only restaurants serving Mexican cuisine are included.

Result Analysis:

- The query successfully retrieves two restaurants (El Oceano Dorado and El Rincón De San Francisco) located in Mexico with a medium range price, open area, and serving Mexican food.
- Each restaurant has its unique alcohol service, smoking policy, and parking availability.
- This query helps identify suitable restaurants meeting specific criteria for potential customers looking for Mexican cuisine in an open setting within a medium price range.

Q2. To retrieve the total number of restaurants with an overall rating of 1 and serving Mexican food, and comparing it with restaurants of same ratings serving Italian food:

```sql
-- Q2. retrieving the total number of restaurants with an overall rating of 1 and serving Mexican food
SELECT COUNT(*) AS Total_Mexican_Restaurants_Rating_1
FROM Restaurants r
INNER JOIN Ratings ra ON r.Restaurant_id = ra.Restaurant_id
INNER JOIN Restaurant_Cuisines rc ON r.Restaurant_id = rc.Restaurant_id
WHERE ra.Overall_Rating = 1
AND rc.Cuisine = 'Mexican';

-- comparing the results with the total number of restaurants with an overall rating of 1 serving Italian food
SELECT COUNT(*) AS Total_Italian_Restaurants_Rating_1
FROM Restaurants r
INNER JOIN Ratings ra ON r.Restaurant_id = ra.Restaurant_id
INNER JOIN Restaurant_Cuisines rc ON r.Restaurant_id = rc.Restaurant_id
WHERE ra.Overall_Rating = 1
AND rc.Cuisine = 'Italian';
```

100 %

Results | Messages

	Total_Mexican_Restaurants_Rating_1
1	87

	Total_Italian_Restaurants_Rating_1
1	11

- Total number of restaurants with an overall rating of 1 serving Mexican food: 87
- Total number of restaurants with an overall rating of 1 serving Italian food: 11

This analysis indicates a significant difference in the number of restaurants with an overall rating of 1 between Mexican and Italian cuisines.

Explanations for this difference:

- **Preference**: It's possible that consumers have a higher expectation or preference for Mexican cuisine compared to Italian cuisine in this dataset. This could lead to a higher number of low-rated Mexican restaurants compared to Italian restaurants.
- **Quality Variation**: The quality and standards of Mexican and Italian restaurants in the dataset might vary significantly. It's possible that there are more low-quality Mexican restaurants compared to Italian restaurants, resulting in a higher number of low ratings.
- **Sample Size**: The dataset could have a larger number of Mexican restaurants compared to Italian restaurants, leading to more instances of low-rated Mexican restaurants.
- **Consumer Expectations**: Consumers might have different expectations or standards for Mexican and Italian cuisines, leading to different rating distributions.
- **Other Factors**: There could be other factors such as location, pricing, service quality, etc., that influences the ratings and contributes to the observed difference.

Q3. To calculate the average age of consumers who have given a 0 rating to the 'Service_rating' column, you can use the following SQL query:

```sql
-- Q3. calculating the average age of consumers who have given a 0 rating to the 'Service_rating'
SELECT ROUND(AVG(c.Age), 0) AS average_age

FROM consumers c

JOIN ratings r ON c.Consumer_id = r.Consumer_id

WHERE r.Service_Rating = 0;
```

100 %

Results | Messages

	average_age
1	26

- **SELECT ROUND(AVG(c.Age), 0) AS Average_Age**:
    - This part of the query calculates the average age of consumers who have given a 0 rating to the 'Service_rating' column and rounds off the result to the nearest integer.

- **AVG(c.Age)** calculates the average age of consumers by taking the average of the 'Age' column in the 'Consumers' table and in this case the result is 26.
- **ROUND(..., 0)** rounds off the calculated average to the nearest integer. The second argument '0' specifies that we want to round to 0 decimal places.
- **AS Average_Age** aliases the result column as 'Average_Age' for easier reference.

- **FROM Consumers c INNER JOIN Ratings ra ON c.Consumer_id = ra.Consumer_id**:

  This part of the query specifies the tables involved and how they are joined.

  - **Consumers c** aliases the 'Consumers' table as 'c'.
  - **Ratings ra** aliases the 'Ratings' table as 'ra'.
  - **ON c.Consumer_id = ra.Consumer_id** specifies the join condition, linking records in the 'Consumers' table with those in the 'Ratings' table based on the 'Consumer_id' column.

- **WHERE ra.Service_Rating = 0**:

  - This part of the query filters the rows to include only those where the 'Service_Rating' column in the 'Ratings' table is equal to 0.
  - It ensures that we're only considering ratings where the service was rated 0.

Q4. To retrieve the restaurants ranked by the youngest consumer along with the food rating given by that customer to the restaurant, sorted by food rating from high to low, the following SQL query can be use:

```sql
-- Q4. Retrieving the restaurants ranked by the youngest consumer along with the food rating
-- by that customer to the restaurant, sorted by food rating from high to low
SELECT
    r.Name AS Restaurant_Name,
    MIN(c.Age) AS Youngest_Consumer_Age,
    ra.Food_Rating
FROM
    restaurants r
JOIN
    ratings ra ON r.Restaurant_id = ra.Restaurant_id
JOIN
    consumers c ON ra.Consumer_id = c.Consumer_id
GROUP BY
    r.Restaurant_id, r.Name, ra.Food_Rating
ORDER BY
    ra.Food_Rating DESC;
```

100 %

Results | Messages

	Restaurant_Name	Youngest_Consumer_Age	Food_Rating
1	Puesto de Gorditas	23	2
2	Cafe Ambar	23	2
3	Church's	21	2
4	Cafe Chaires	22	2
5	McDonalds Centro	20	2
6	Gorditas Doña Tota	23	2
7	Tacos De Barbacoa Enfrente Del Tec	23	2
8	Hamburguesas La Perica	21	2
9	Pollo Frito Buenos Aires	23	2
10	Camitas Mata	23	2
11	La Perica Hamburguesa	23	2

Query executed successfully. | DESKTOP-MA56CTA (16.0 RTM) | DESKTO

- **SELECT r.Name AS Restaurant_Name, MIN(c.Age) AS Youngest_Consumer_Age, ra.Food_Rating**:
    - This part of the query selects three columns: the restaurant name (aliased as Restaurant_Name), the minimum age of consumers (to find the youngest consumer) for each restaurant (aliased as Youngest_Consumer_Age), and the food rating given by that consumer to the restaurant.
- **FROM restaurants r JOIN ratings ra ON r.Restaurant_id = ra.Restaurant_id JOIN consumers c ON ra.Consumer_id = c.Consumer_id**:
    - This part of the query specifies the tables involved in the query and how they are joined. It joins the 'restaurants' table with the 'ratings' table based on the 'Restaurant_id', and then joins the 'ratings' table with the 'consumers' table based on the 'Consumer_id'.
- **GROUP BY r.Restaurant_id, r.Name, ra.Food_Rating**:

- This part of the query groups the results by 'Restaurant_id', 'Name' (restaurant name), and 'Food_Rating'. It ensures that each group represents a unique combination of restaurant and food rating.

- **ORDER BY ra.Food_Rating DESC**:
  - This part of the query orders the results by food rating in descending order, meaning restaurants with higher food ratings appear first.

Results Analysis:

- The results show each restaurant's name along with the age of the youngest consumer who rated the restaurant and the food rating given by that consumer.
- The food ratings range from 0 to 2, with 2 being the highest rating and 0 being the lowest.
- The restaurants are listed in descending order of food rating, meaning those with higher ratings appear first. If two restaurants have the same food rating, they are ordered by the age of the youngest consumer, with younger consumers listed first.

Q5. Writing a stored procedure for the query given as:

Update the Service_rating of all restaurants to '2' if they have parking available

```sql
-- Q5. Writing a stored procedure for the query given as:
-- Update the Service_rating of all restaurants to '2' if they have parking available

CREATE PROCEDURE UpdateServiceRatingWithParking
AS
BEGIN
    SET NOCOUNT ON;
    -- Updating Service_Rating for restaurants with parking available
    UPDATE ratings
    SET Service_Rating = '2'
    WHERE Restaurant_id IN (
        SELECT r.Restaurant_id
        FROM restaurants r
        WHERE r.Parking IN ('yes', 'public')
    );
END;
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2024-04-22T14:29:02.9637021+01:00
```

This stored procedure, named "UpdateServiceRatingWithParking," updates the Service_Rating of all restaurants to '2' if they have parking available. Breaking down the procedure:

- **CREATE PROCEDURE UpdateServiceRatingWithParking**:
    - This initiates the creation of a stored procedure named "UpdateServiceRatingWithParking."
- **AS BEGIN**:
    - Begins the definition of the stored procedure.
- **SET NOCOUNT ON**:
    - This is used to suppress the message indicating the number of rows affected by the SQL statements inside the stored procedure. It's often used to reduce network traffic when the number of rows affected is not needed.
- **UPDATE ratings SET Service_Rating = '2' WHERE Restaurant_id IN (SELECT r.Restaurant_id FROM restaurants r WHERE r.Parking IN ('yes', 'public'))**:
    - This is the main logic of the stored procedure.
    - It updates the Service_Rating column in the "ratings" table to '2' for all restaurants that have parking available.
    - The WHERE clause filters the restaurants based on their parking availability. It checks if the Parking column in the "restaurants" table contains values 'yes' or 'public'. If so, it retrieves the corresponding Restaurant_id.
- **END**:
    - Marks the end of the stored procedure definition.

Q6. The Four Queries

Query 1: The average overall rating for restaurants that serve Mexican cuisine and have a price level of 'Medium'.

```sql
-- Query 1: Find the average overall rating for restaurants that serve Mexican cuisine
-- and have a price level of 'Medium'.

SELECT AVG(ra.Overall_Rating) AS Avg_Overall_Rating
FROM Ratings ra
WHERE ra.Restaurant_id IN (
    SELECT r.Restaurant_id
    FROM Restaurants r
    WHERE r.Price = 'Medium'
    AND r.Restaurant_id IN (
        SELECT rc.Restaurant_id
        FROM Restaurant_Cuisines rc
        WHERE rc.Cuisine = 'Mexican'
    )
);
```

100 %

Results | Messages

	Avg_Overall_Rating
1	1

- This query calculates the average overall rating for restaurants that serve Mexican cuisine and have a price level of 'Medium'.
- It uses nested queries with the IN operator to filter restaurants based on the specified criteria.
- The outer query calculates the average overall rating for the filtered restaurants.

Query 2: List restaurants with the highest food rating.

```sql
-- Query 2: List restaurants with the highest food rating.
SELECT Top 5 r.Name AS Restaurant_Name, MAX(ra.Food_Rating) AS Max_Food_Rating
FROM Restaurants r
JOIN Ratings ra ON r.Restaurant_id = ra.Restaurant_id
GROUP BY r.Name
ORDER BY Max_Food_Rating DESC
;
```

100 %

Results | Messages

	Restaurant_Name	Max_Food_Rating
1	Cafe Punta Del Cielo	2
2	Cafe Chaires	2
3	Cafe Ambar	2
4	Cabana Huasteca	2
5	Arrachela Grill	2

- This query retrieves the top 5 restaurants with the highest food rating.

- It uses GROUP BY to group the results by restaurant name and calculates the maximum food rating for each restaurant.
- The results are sorted in descending order of food rating, and the top 5 restaurants are selected.

Query 3: Finding the number of consumers who have rated a restaurant's service higher than its food.

```sql
-- Query 3: Find the number of consumers who have rated a restaurant's service higher than its food.
SELECT COUNT(*) AS Num_Consumers
FROM (
    SELECT ra.Consumer_id
    FROM Ratings ra
    WHERE ra.Service_Rating > ra.Food_Rating
    GROUP BY ra.Consumer_id
) AS Subquery;
```

100 %

Results | Messages

	Num_Consumers
1	71

- This query counts the number of consumers who have rated a restaurant's service higher than its food.
- It uses a nested query with EXISTS to filter ratings where the service rating is higher than the food rating.
- The outer query counts the distinct consumers from the filtered ratings.

System functions

```sql
-- System functions
SELECT Name, City, State, LEN(Name) AS Name_Length
FROM restaurants
SELECT Name, City, State
FROM restaurants
WHERE Restaurant_id IN (
    SELECT Restaurant_id
    FROM ratings
    GROUP BY Restaurant_id
    HAVING AVG(Overall_Rating) < 3
);
```

100 %

Results | Messages

	Name	City	State	Name_Length
1	Puesto de Gorditas	Ciudad Victoria	Tamaulipas	18
2	Cafe Ambar	Ciudad Victoria	Tamaulipas	10
3	Church's	Ciudad Victoria	Tamaulipas	8
4	Cafe Chaires	San Luis Potosi	San Luis Potosi	12
5	McDonalds Centro	Cuernavaca	Morelos	16
6	Gorditas Doña Tota	Ciudad Victoria	Tamaulipas	18
7	Tacos De Barbacoa Enfrente Del Tec	Ciudad Victoria	Tamaulipas	34
8	Hamburguesas La Perica	Ciudad Victoria	Tamaulipas	22

	Name	City	State
1	Puesto de Gorditas	Ciudad Victoria	Tamaulipas
2	Cafe Ambar	Ciudad Victoria	Tamaulipas

Query executed successfully.

The main purpose of this query is to select the names, cities, and states of restaurants whose average overall rating is less than 3. It calculates the length of the restaurant names. The subquery is used to filter restaurants based on their average ratings, and the outer query fetches the desired columns for those filtered restaurants.

Use of GROUP BY, HAVING and ORDER BY clauses

This query utilizes the GROUP BY, HAVING, and ORDER BY clauses to retrieve information about restaurant prices where the count of restaurants with each price is greater than 10.

```sql
-- Use of GROUP BY, HAVING and ORDER BY clauses

SELECT COUNT(*), Price

FROM restaurants

GROUP BY Price

HAVING COUNT(*) > 10;
```

100 %

Results | Messages

	(No column name)	Price
1	25	High
2	45	Low
3	60	Medium

This query retrieves the count of restaurants for each price category where the count is greater than 10, and it orders the result set by price. It's useful for identifying price categories with a significant number of restaurants.

Conclusion:

The database design solution includes four tables: Restaurants, Consumers, Ratings, and Restaurant_Cuisines. Each table captures specific information related to restaurants, consumers, their ratings, and the cuisines served by restaurants. Primary and foreign key constraints ensure data integrity and establish relationships between tables.

Key functionality provided by the database includes:

1. **Storing Restaurant Information**: Stores details such as name, location, pricing, services, and parking availability of restaurants.
2. **Capturing Consumer Data**: Records information about consumers, including their demographics, preferences, and ratings given to restaurants.
3. **Recording Ratings**: Links consumers to restaurants and stores their ratings for overall experience.
4. **Tracking Cuisines**: Associates restaurants with the cuisines they serve, allowing for easy retrieval of restaurants based on cuisine type.
5. **Querying and Analysis**: Supports various types of queries, including filtering restaurants by specific criteria, analyzing consumer ratings, and calculating aggregate statistics.