Task 1 Report

To have the capability to store critical information and data related to patients, medical records, appointments, doctors, and departments, a thorough platform is required to be created to achieve a hospital's database structure. To be able to create a semantic model, guiding the entity relational diagram, which supports all functions, the creation of a database design is required which will store the data. To fulfil the database needs of the hospital, a hybrid database design approach will be used, where the top-down part will help identify the overall structure of the system, tables and relationships needed to achieve the acceptance criteria. Moreover, the bottom up will help split and refine the tables where needed into more tables, making sure that they all follow the three normal forms of data normalisation to make sure that the design achieved is free from anomalies and redundancies. In addition, the hybrid approach offers a certain flexibility throughout the design process that allows the switch between the two approaches and allows the incorporation of new requirement changes.

Part 1

Database tables:

Patients Table:

The patient's table includes all the required and essential information of the patients who are staying at the hospital and using its services. The table contains PID as the unique primary key. In addition, the patient's table contains the following fields: first name, middle name, last name, date of birth, and phone number. These personally identifiable information can help identify each patient. Moreover, billing is heavily reliant on the insurance number, which is why it is collected. The email address is optional which can function as a backup contact detail to the patient. And the address foreign key helps map the address to the patient. The left date field is included to track the date that the patient has left the hospital, fulfilling the hospital's requirement to keep information of previous patients according to GDPR. And a constraint is added to ensure the quality of the data, such as the email address has a standard pattern of X@X.com

```sql
CREATE TABLE Dim_Patients (
    PID INT PRIMARY KEY,
    PAID INT NOT NULL,
    First_Name VARCHAR(50) NOT NULL,
    Middle_Name VARCHAR(50),
    Last_Name VARCHAR(50) NOT NULL,
    DoB DATE NOT NULL,
    Insurance_number INT NOT NULL,
    Email VARCHAR(255),
    Telephone_number VARCHAR(16),
    Left_Date DATE,
    FOREIGN KEY (PAID) REFERENCES [dbo].Dim_Patient_Address(PAID)
);

ALTER TABLE Dim_Patients
ADD CONSTRAINT Check_Email_DP
CHECK (Email LIKE '%@%.com')

INSERT INTO Dim_Patients (PID, PAID, First_Name, Middle_Name, Last_Name, DoB, Insurance_number, Email, Telephone_number, Left_Date)
VALUES
(1, 101001, 'Alan', 'Michael', 'Rice', '1982-02-15', 10059435, 'Alan.Rice@outlook.com', '07700 123456', NULL),
(2, 101002, 'Chris', NULL, 'Frost', '1984-07-27', 10024850, 'Chris.Frost@gmail.com', '07700 123457', NULL),
(3, 101003, 'Kishan', 'Alan', 'Chauhan', '1986-11-03', 10056179, 'Kishan.Chauhan@hotmail.com', '07700 123458', NULL),
(4, 101004, 'George', 'Roy', 'Mokhtar', '1988-04-22', 10082816, 'George.Mokhtar@outlook.com', '07700 123459', NULL),
(5, 101005, 'Ade', NULL, 'Awonaike', '1990-09-09', 10014924, 'Ade.Awonaike@outlook.com', '07700 123460', NULL),
(6, 101006, 'Zac', 'Nassib', 'Hosn', '1992-12-14', 10094318, 'Zac.Hosn@gmail.com', '07700 123461', '2023-09-12'),
(7, 101007, 'Dan', NULL, 'Waterman', '1981-03-18', 10023487, 'Dan.Waterman@hotmail.com', '07700 123462', '2022-03-28'),
(8, 101008, 'Aquilla', NULL, 'Matafwali', '1983-06-25', 10031615, 'Aquilla.Matafwali@outlook.com', '07700 123463', '2023-11-19'),
(9, 101009, 'Tamsin', NULL, 'Anderson', '1985-10-30', 10025893, 'Tamsin.Anderson@gmail.com', '07700 123464', '2022-07-02'),
(10, 101006, 'Claire', NULL, 'Hosn', '1995-05-21', 10027209, 'Claire.Mokhtar@gmail.com', '07750 992731', NULL);
```

Patient Address Table:

Address is a particularly important attribute for multiple reasons, such as sending prescriptions, sending test results, billing, and financials. Thus, the table should include granular address information and uniquely identified by the PAID (Patient Address ID). It is separate from the Patients' table as multiple household occupants could have the same address, especially if it is a complex shared accommodation. A constraint was added which controls the data entry for the Postcodes when the patients are filling out their addresses.

```sql
CREATE TABLE Dim_Patient_Address (
    PAID INT PRIMARY KEY,
    House_Number VARCHAR(5) NOT NULL,
    Street_Number INT NOT NULL,
    Street_Name VARCHAR(100) NOT NULL,
    Post_Code VARCHAR(10) NOT NULL
);

ALTER TABLE Dim_Patient_Address
ADD CONSTRAINT Check_Postcode
CHECK (Post_Code LIKE '[A-Z]%[A-Z]')

INSERT INTO Dim_Patient_Address (PAID, House_Number, Street_Number, Street_Name, Post_Code)
VALUES
(101001, '67', '123', 'King Street', 'M1 2AD'),
(101002, '118A', '456' ,'Queen Road', 'M2 3BC'),
(101003, '12', '789', 'Prince Avenue', 'M3 4DE'),
(101004, '59', '321', 'Princess Drive', 'M4 5EF'),
(101005, '34B', '654', 'Duke Lane', 'M5 6GH'),
(101006, '23', '987', 'Duchess Way', 'M6 7IJ'),
(101007, '101', '231', 'Earl Boulevard', 'M7 8JK'),
(101008, '81', '564', 'Countess Park', 'M8 9LM'),
(101009, '76', '897', 'Viscount Place', 'M9 0NO');
```

Doctors Table:

Doctors are key entities in a hospital. The Doctors table includes a unique DoctorID for identification, FullName for personal identification, and Specialization to detail their expertise. It includes the years of experience as well, and the usual room number that he receives patients in for ease of access. The DepartmentID as a foreign key

connects each doctor to their respective department, reflecting the hospital's organizational structure. Two constraints were added; one would make sure that the doctor enters his hospital username in addition to the hospital' domain, which together would make up the email address, and the second would make sure that the doctor is allocated to an actual real room in the hospital, specified by all the available rooms.

```
CREATE TABLE Dim_Doctors (
    DID INT PRIMARY KEY,
    First_Name VARCHAR(50) NOT NULL,
    Last_Name VARCHAR(50) NOT NULL,
    Specialty VARCHAR(50) NOT NULL,
    DepID INT NOT NULL,
    Email VARCHAR(255) NOT NULL,
    Room_no VARCHAR(5) NOT NULL,
    Yrs_of_exp INT NOT NULL,
    FOREIGN KEY (DepID) REFERENCES [dbo].[Dim_Departments](DepID)
);

ALTER TABLE Dim_Doctors
ADD CONSTRAINT Check_Email_DD
CHECK (Email LIKE '%@george-hosp.com')

ALTER TABLE Dim_Doctors
ADD CONSTRAINT Check_Room_DD
CHECK (Room_no LIKE '1[0-4][0, 2, 4, 6, 8][A-H]')

INSERT INTO Dim_Doctors (DID, First_Name, Last_Name, Specialty, DepID, Email, Room_no, Yrs_of_exp)
VALUES
(101, 'Jared', 'Night', 'Cardiologist', 1001, 'Jared.N@george-hosp.com', '114A', 7),
(102, 'Phil', 'Greenwood', 'Dermatologist', 1002, 'Phil.G@george-hosp.com', '116B', 11),
(103, 'Claire', 'Dunphy', 'Neurologist', 1001, 'Claire.D@george-hosp.com', '118A', 6),
(104, 'Ben', 'Wood', 'Gastroenterologists', 1002, 'Ben.W@george-hosp.com', '120C', 15),
(105, 'Dylan', 'Carlson', 'Pediatrician', 1003, 'Dylan.C@george-hosp.com', '126F', 9),
(106, 'Dan', 'Barlow', 'Gastroenterologists', 1005, 'Dan.B@george-hosp.com', '124D', 12),
(107, 'Prashant', 'Patel', 'Pediatrician', 1004, 'Prashant.P@george-hosp.com', '120F', 9);
```

Departments Table:

Departments hold the practitioners in a certain taxonomy and hierarchy that deems necessary for the organization of the hospital. The Departments table with a unique DepartmentID, holds the department name, location, and email address for the admin team, allows for clear identification and organization of different hospital areas, facilitating department-specific patient care and administrative management. Two constraint control couple of the columns; making sure that the department email address is the department name followed by the domain name of the hospital, and the second ensures that the department name specified is uniquely identical, except for the last letter which recognizes the different departments.

```sql
CREATE TABLE Dim_Departments (
    DepID INT PRIMARY KEY,
    Dept_Name VARCHAR(12) NOT NULL,
    Location VARCHAR(8) NOT NULL,
    Dept_Email VARCHAR(30) NOT NULL
);

ALTER TABLE Dim_Departments
ADD CONSTRAINT Check_Email
CHECK (Dept_Email LIKE '%@george-hosp.com')

ALTER TABLE Dim_Departments
ADD CONSTRAINT Check_Dept_Name
CHECK (Dept_Name LIKE 'Department[A-Z]')

INSERT INTO Dim_Departments (DepID, Dept_Name, Location, Dept_Email)
VALUES
(1001, 'DepartmentA', 'Wing_1', 'DepartmentA@george-hosp.com'),
(1002, 'DepartmentB', 'Wing_2', 'DepartmentB@george-hosp.com'),
(1003, 'DepartmentC', 'Wing_3', 'DepartmentC@george-hosp.com'),
(1004, 'DepartmentD', 'Wing_4', 'DepartmentD@george-hosp.com'),
(1005, 'DepartmentE', 'Wing_5', 'DepartmentE@george-hosp.com'),
(1006, 'DepartmentF', 'Wing_6', 'DepartmentF@george-hosp.com'),
(1007, 'DepartmentG', 'Wing_7', 'DepartmentG@george-hosp.com');
```

Appointments Tables:

There are two appointments tables created. The Appointments table is crucial for managing the scheduling aspects and tracking the hospital's activities. It includes Patients and Doctors ids through foreign keys (PID, DID), and records the appointment's date, time, and status. The DepartmentID foreign key indicates where the appointment is scheduled, and the Review field allows storing feedback for completed appointments, addressing the requirement for post-appointment feedback. The Rebooking flag which will be provided by the doctor, will indicate 1 if the patient needs another visit, or vice versa. A constraint was added to ensure unity in the statuses provided, which forces the admin team to enter only 1 of 4 set statuses: Completed, Cancelled, Pending, or Available. The above constraint was added for both tables, previous and future. A final constraint was added for the Appointments future tables which checks if the date is small or equal to the current date of the system

```sql
CREATE TABLE Fact_Appointments (
    PID INT NOT NULL,
    DID INT NOT NULL,
    Date DATE NOT NULL,
    Time TIME NOT NULL,
    DepID INT NOT NULL,
    Status VARCHAR(10) NOT NULL,
    Rebooking_Flag TINYINT NOT NULL,
    Review VARCHAR(255),
    FOREIGN KEY (PID) REFERENCES [dbo].[Dim_Patients](PID),
    FOREIGN KEY (DID) REFERENCES [dbo].[Dim_Doctors](DID),
    FOREIGN KEY (DepID) REFERENCES [dbo].[Dim_Departments](DepID)
);

ALTER TABLE Fact_Appointments
ADD CONSTRAINT Check_Status
CHECK (Status = 'Completed' OR Status = 'Cancelled' OR Status = 'Pending' OR Status = 'Available')
```

```sql
INSERT INTO Fact_Appointments (PID, DID, Date, Time, DepID, Status, Rebooking_Flag, Review)
VALUES
(6, 105, '2024-01-09', '13:15:00', 1005, 'Completed', 0, NULL),
(7, 103, '2024-01-12', '10:45:00', 1001, 'Completed', 0, 'I waited for such a while!'),
(3, 106, '2024-01-15', '11:15:00', 1004, 'Completed', 0, 'Thank you'),
(9, 105, '2024-01-27', '14:45:00', 1002, 'Cancelled', 1, NULL),
(9, 102, '2024-01-29', '12:45:00', 1004, 'Completed', 0, 'Friendly staff'),
(9, 104, '2024-01-30', '14:30:00', 1002, 'Completed', 0, 'Great experience, friendly Dr.'),
(1, 101, '2024-02-11', '11:00:00', 1001, 'Completed', 0, 'Thank you Dr.'),
(4, 102, '2024-02-12', '13:00:00', 1001, 'Completed', 0, NULL),
(1, 101, '2024-02-15', '11:30:00', 1001, 'Cancelled', 1, NULL),
(5, 107, '2024-04-29', '16:15:00', 1003, 'Cancelled', 1, NULL),
(6, 101, '2024-06-26', '13:30:00', 1003, 'Cancelled', 1, NULL);

CREATE TABLE Fact_Appointments_Future (
    PID INT NOT NULL,
    DID INT NOT NULL,
    Date DATE NOT NULL,
    Time TIME NOT NULL,
    DepID INT NOT NULL,
    Status VARCHAR(10) NOT NULL,
    Rebooking_Flag TINYINT NOT NULL,
    Review VARCHAR(255),
    FOREIGN KEY (PID) REFERENCES [dbo].[Dim_Patients](PID),
    FOREIGN KEY (DID) REFERENCES [dbo].[Dim_Doctors](DID),
    FOREIGN KEY (DepID) REFERENCES [dbo].[Dim_Departments](DepID)
);

ALTER TABLE Fact_Appointments_Future
ADD CONSTRAINT Check_Status_Future
CHECK (Status = 'Cancelled' OR Status = 'Pending' OR Status = 'Available')

ALTER TABLE Fact_Appointments_Future
ADD CONSTRAINT Check_Date_Future
CHECK (Date >= GETDATE())

INSERT INTO Fact_Appointments (PID, DID, Date, Time, DepID, Status, Rebooking_Flag, Review)
VALUES
(7, 103, '2024-04-29', '09:00:00', 1001, 'Pending', 0, NULL),
(5, 102, '2024-05-16', '15:45:00', 1003, 'Pending', 0, NULL),
(2, 104, '2024-05-08', '15:45:00', 1002, 'Pending', 0, NULL),
(4, 102, '2024-07-02', '13:15:00', 1001, 'Pending', 0, NULL),
(2, 104, '2024-06-30', '14:00:00', 1002, 'Pending', 0, NULL);
```

Medical Records Table:

This table is vital for tracking patient health over time. Each record is linked to a patient (PID) and a Doctor (DID) to maintain a detailed history. It includes a foreign key for medicines, which could be null in case none were prescribed, the date it was prescribed, diagnoses, and any noted allergies, ensuring comprehensive medical history tracking. A constraint is added to ensure that once a medicine is prescribed and in turn recorded in the database, the doctor will have to submit the date when that medicine was prescribed for tracking and logging purposes.

```sql
CREATE TABLE Fact_Medical_Records (
    PID INT NOT NULL,
    DID INT NOT NULL,
    MID VARCHAR(5),
    Diagnosis VARCHAR(255) NOT NULL,
    Dose INT NOT NULL,
    Prescription_Date DATE,
    Allergies VARCHAR(255),
    FOREIGN KEY (PID) REFERENCES [dbo].[Dim_Patients](PID),
    FOREIGN KEY (DID) REFERENCES [dbo].[Dim_Doctors](DID),
    FOREIGN KEY (MID) REFERENCES [dbo].[Dim_Medicines](MID)
);

ALTER TABLE Fact_Medical_Records
ADD CONSTRAINT prescription_date_for_medicine
CHECK
    (MID IS NOT NULL and Prescription_Date IS NOT NULL);

INSERT INTO Fact_Medical_Records (PID, DID, MID, Diagnosis, Dose, Prescription_Date, Allergies)
VALUES
(1, 105, 'AAABA ', 'Hypertension', 100, '2022-03-14', 'Pollen'),
(1, 104, 'AAABB', 'Cancer', 250, '2023-11-02', NULL),
(2, 106, 'AAABC', 'Asthma', 100, '2022-05-21', 'Dust'),
(2, 103, 'AAABD', 'Chronic Obstructive Pulmonary Disease (COPD)', 500, '2023-04-03', NULL),
(3, 107, 'AAABE', 'Rheumatoid Arthritis', 20, '2023-08-30', NULL),
(4, 103, 'AAABF', 'Osteoporosis', 5, '2023-02-09', 'Dog'),
(5, 102, 'AAABA', 'Hyperthyroidism', 50, '2023-05-17', 'Peanuts'),
(5, 101, 'AAABB', 'Hypothyroidism', 100, '2022-09-11', NULL),
(6, 103, 'AAABC', 'Gastroesophageal Reflux Disease (GERD)', 250, '2022-10-13', 'Kiwi'),
(7, 107, 'AAABD', 'Peptic Ulcer Disease', 300, '2023-04-11', NULL),
(8, 101, 'AAABE', 'Migraine', 250, '2023-06-21', NULL),
(9, 104, 'AAABF', 'Psoriasis', 10, '2022-12-07', NULL),
(9, 106, 'AAABA', 'Eczema', 50, '2022-07-19', NULL);
```

Medicines:

The medicines table is a central database for all medicines that were and will be prescribed by the doctors in the hospital. The medicine information includes the name, the base formula, and could include more details further down the road. To limit data redundancy, a dimensional table was created to store this information once in a unique manner.

```sql
---------- Medicine Table Creation ----------
CREATE TABLE Dim_Medicines (
    MID VARCHAR(5) PRIMARY KEY,
    Medicine_Name VARCHAR(50) NOT NULL,
    Base_formula VARCHAR(50) NOT NULL
);

INSERT INTO Dim_Medicines (MID, Medicine_Name, Base_formula)
VALUES
('AAABA', 'Tranquillia-XR', 'Corticosteroids '),
('AAABB', 'RespiroSol', 'Omeprazole'),
('AAABC', 'AlliumMend', 'Amlodipine'),
('AAABD', 'Lumizol', 'Ramipril'),
('AAABE', 'CardioNova', 'Lansoprazole'),
('AAABF', 'DermaRegen', 'Colecalciferol');
```

User Authentication:

This is an important table, which is why it was kept separate from other sensitive information such as the PII (Personal Identifiable Information) of the patients. It includes the Patient ID as the foreign key, though set to unique to ensure the relationship integrity between the tables, the username of the patient, and the password to access the portal. A constraint was created and applied on the password column, which ensures the following:

- Password includes at least one capital letter.
  - PATINDEX('%[A-Z]%', Password) > 0
- Password includes at least one number.
  - PATINDEX('%[0-9]%', Password) > 0
- Password includes at least one small letter.
  - PATINDEX('%[a-z]%', Password) > 0
- Password is at least six characters in length.
  - LEN(Password) >= 6);

```
242 CREATE TABLE User_Auth_tbl (
243     PID INT UNIQUE,
244     Username VARCHAR(100) NOT NULL,
245     Password VARCHAR(50) NOT NULL,
246     FOREIGN KEY (PID) REFERENCES [dbo].[Dim_Patients](PID)
247 );
248
249 INSERT INTO User_Auth_tbl (PID, Username, Password)
250 VALUES
251 (1, 'AlanRice', 'Admin1'),
252 (2, 'ChrisFrost', 'Admin2'),
253 (3, 'KishanChauhan', 'Admin3'),
254 (4, 'GeorgeMokhtar', 'Admin4'),
255 (5, 'AdeAwonaike', 'Admin5'),
256 (6, 'ZacHosn', 'Admin6'),
257 (7, 'DanWaterman', 'Admin7'),
258 (8, 'AquillaMatafwali', 'Admin8'),
259 (9, 'TamsinAnderson', 'Admin9'),
260 (10, 'ClaireHosn', 'Admin10');
261
262 ALTER TABLE User_Auth_tbl
263 ADD CONSTRAINT Check_Password
264 CHECK (PATINDEX('%[A-Z]%', Password) > 0
265     AND PATINDEX('%[0-9]%', Password) > 0
266     AND PATINDEX('%[a-z]%', Password) > 0
267     AND LEN(Password) >= 6);
```

Billing table:

The billing table holds the financial information for each of the patients. Once they get a treatment, or a service from the hospital, their account will be charged with the respective amount. The table includes information around the amount charged, amount due, payment status, date charged, when the record was last updated, and it is identifiable based on the three foreign keys included for Patient ID, Doctor ID, and Department ID. The constraint added ensures the status consistency by making sure the team can only add 1 of the 3 available statuses in the database.

```sql
---------- Fact_Billing Dimension Creation ----------
CREATE TABLE Fact_Billing (
    PID INT NOT NULL,
    DID INT NOT NULL,
    DepID INT NOT NULL,
    Date_Charged DATE NOT NULL,
    Amount_Charged DECIMAL NOT NULL,
    Amount_Due DECIMAL NOT NULL,
    Payment_Status VARCHAR(10),
    Date_updated DATE,
    FOREIGN KEY (PID) REFERENCES [dbo] [Dim_Patients](PID),
    FOREIGN KEY (DID) REFERENCES [dbo] [Dim_Doctors](DID),
    FOREIGN KEY (DepID) REFERENCES [dbo] [Dim_Departments](DepID)
);

ALTER TABLE Fact_Billing
ADD CONSTRAINT Check_Status_Payment
CHECK (Payment_Status = 'Paid' OR Payment_Status = 'Unpaid' OR Payment_Status = 'Partial')

INSERT INTO Fact_Billing (PID, DID, DepID, Date_Charged, Amount_Charged, Amount_Due, Payment_Status, Date_updated)
VALUES
(5, 105, 1001, '2023-03-12', 1598, 984.56, 'Partial', '2023-05-27'),
(2, 101, 1001, '2022-06-29', 1197.36, 1197.36, 'Unpaid', NULL),
(3, 102, 1001, '2022-09-02', 2729.41, 1675.25, 'Partial', NULL),
(4, 103, 1001, '2023-09-12', 1913.67, 0, 'Paid', NULL),
(1, 106, 1001, '2023-04-18', 5243.46, 5243.46, 'Unpaid', '2023-11-18'),
(7, 104, 1001, '2022-11-05', 942.50, 268.67, 'Partial', NULL),
(6, 107, 1001, '2023-10-20', 2264, 0, 'Paid', NULL);
```

Data Types Explanation:

```sql
SELECT
    TABLE_NAME,
    COLUMN_NAME,
    CASE
        WHEN DATA_TYPE = 'varchar'
        THEN CONCAT(DATA_TYPE, '(', CHARACTER_MAXIMUM_LENGTH, ')')
        ELSE DATA_TYPE
        END AS DATA_TYPE,
    IS_NULLABLE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_CATALOG = 'George_Hospital'
    and TABLE_NAME not in ('all_appointments', 'sysdiagrams')
ORDER BY TABLE_NAME, DATA_TYPE Asc
```

- INT
    - As shown below, this data type was used for most IDs and numbers in the tables. Although in this specific example, “tinyint” would have been a correct data type, but if scalability is kept in mind for further down the road, “int” is the most suitable data type in this case.
- TINYINT
    - It was specifically used for the rebooking flag field, as the value within it would be either 1 or 0. Thus using “tinyint” would save more space than using “int”.
- DATE & TIME
    - It was used for all dates and time columns, and it facilitates the use of datetime functions in all future queries required.
- DECIMAL
    - It was used in the billing table, specifically in recording the amounts and finance related fields. It will allow the value to have decimal places which is extremely useful in finance.

- VARCHAR
    - It was used throughout the tables as it is the most commonly used data type. The benefit of using a variable character, is the fact that even if its length is 255 and the user only inputs 10, only 10 characters will be recorded in storage. In the below table, as per the screenshot, some lengths were specified for the sake of data quality, for example:
        - MID's length is 5 as it is the maximum number of characters allowed.
        - Status's length is 10 as it is the maximum characters lengths between the 3 allowed statuses.
        - Telephone number's length is 16 as it is the maximum number of characters in a phone number including the country code. Etc…

	TABLE_NAME	COLUMN_NAME	DATA_TYPE	IS_NULLABLE
1	Dim_Departments	DepID	int	NO
2	Dim_Departments	Dept_Name	varchar(12)	NO
3	Dim_Departments	Dept_Email	varchar(30)	NO
4	Dim_Departments	Location	varchar(8)	NO
5	Dim_Doctors	DID	int	NO
6	Dim_Doctors	DepID	int	NO
7	Dim_Doctors	Yrs_of_exp	int	NO
8	Dim_Doctors	Email	varchar(255)	NO
9	Dim_Doctors	Room_no	varchar(5)	NO
10	Dim_Doctors	First_Name	varchar(50)	NO
11	Dim_Doctors	Last_Name	varchar(50)	NO
12	Dim_Doctors	Specialty	varchar(50)	NO
13	Dim_Medicines	MID	varchar(5)	NO
14	Dim_Medicines	Medicine_Name	varchar(50)	NO
15	Dim_Medicines	Base_formula	varchar(50)	NO
16	Dim_Patient_Address	PAID	int	NO
17	Dim_Patient_Address	Street_Number	int	NO
18	Dim_Patient_Address	Post_Code	varchar(10)	NO
19	Dim_Patient_Address	Street_Name	varchar(100)	NO
20	Dim_Patient_Address	House_Number	varchar(5)	NO
21	Dim_Patients	DoB	date	NO
22	Dim_Patients	Left_Date	date	YES
23	Dim_Patients	Insurance_number	int	NO
24	Dim_Patients	PID	int	NO
25	Dim_Patients	PAID	int	NO
26	Dim_Patients	Telephone_number	varchar(16)	NO
27	Dim_Patients	Email	varchar(255)	YES
28	Dim_Patients	First_Name	varchar(50)	NO
29	Dim_Patients	Middle_Name	varchar(50)	YES
30	Dim_Patients	Last_Name	varchar(50)	NO
31	Fact_Appointments	Date	date	NO
32	Fact_Appointments	DepID	int	NO
33	Fact_Appointments	PID	int	NO
34	Fact_Appointments	DID	int	NO
35	Fact_Appointments	Time	time	NO
36	Fact_Appointments	Rebooking_Flag	tinyint	NO
37	Fact_Appointments	Status	varchar(10)	NO
38	Fact_Appointments	Review	varchar(255)	YES
39	Fact_Billing	Date_Charged	date	NO
40	Fact_Billing	Date_updated	date	YES
41	Fact_Billing	Amount_Charged	decimal	NO
42	Fact_Billing	Amount_Due	decimal	NO
43	Fact_Billing	PID	int	NO
44	Fact_Billing	DID	int	NO
45	Fact_Billing	DepID	int	NO
46	Fact_Billing	Payment_Status	varchar(10)	YES
47	Fact_Medical_Rec...	Prescription_Date	date	YES
48	Fact_Medical_Rec...	Dose	int	NO
49	Fact_Medical_Rec...	PID	int	NO
50	Fact_Medical_Rec...	DID	int	NO
51	Fact_Medical_Rec...	Diagnosis	varchar(255)	NO
52	Fact_Medical_Rec...	Allergies	varchar(255)	YES
53	Fact_Medical_Rec...	MID	varchar(5)	YES
54	User_Auth_tbl	PID	int	YES
55	User_Auth_tbl	Username	varchar(100)	NO
56	User_Auth_tbl	Password	varchar(50)	NO

Relationships explanation:

- Departments to Doctors:
  - Doctors are associated with specific departments based on their specialization. The One-to-Many relationship allows multiple doctors to be associated with a single department, reflecting real-world hospital structures.
- Departments to Appointments:
  - Appointments are made in specific departments based on the medical service required. The One-to-Many relationship allows for the organization of appointments by department, facilitating resource and space management.
- Departments to Billing:
  - Bills are related and fall under a specific department which provided the services for the patient. The One-to-Many relationship allows for the organization of the finances for every department, facilitating financial auditing.
- Doctors to Appointments:
  - A doctor can see multiple patients, hence multiple appointments. The One-to-Many relationship here enables tracking of all appointments per doctor, aiding in schedule management and workload analysis.
- Doctors to Medical Records:
  - A doctor can submit multiple medical records for multiple patients. The One-to-Many relationship in this case enables the doctor to track his patients and their medical history enabling better medical care.
- Doctors to Billing:
  - A doctor can treat multiple patients and in turn be responsible for multiple bills. The One-to-Many relationships allows the doctors to check what they have billed and track their finances in case they get paid per patient or a percentage of the bill.
- Medicines to Medical Records:
  - One medicine and all of its information could be prescribed and recorded in multiple medical records. The One-to-Many relationship in that case can allow the hospital to track medicine inventory needed based on prescriptions.
- Patients to Appointments:
  - Patients can have multiple appointments over time, necessitating a One-to-Many relationship. This relationship allows the system to track all appointments a patient has, past and future, in a structured manner.
- Patients to Medical Records:
  - Each patient can have multiple medical records from various appointments, necessitating a One-to-Many relationship. This setup ensures a comprehensive and accessible medical history for each patient.
- Patients to Billing:

    - A patient could have used multiple services around the hospital, thus be billed multiple times. The One-to-Many relationship, allows the patient to track their billing on the patient portal.
- Patient Address to Patients:
    - Multiple patients could have the same address, and if patients move addresses and it is already in the system, it will not require the addition of the same address multiple times. Thus, One-to-Many from Patient Address to Patients.
- User Authentication to Patients:
    - This is a One-to-One relationship that is specifically built for login. One patient can only have one user login information, which allows the patient to login to their portal.

Data Normalization:

The designed database mentioned above, adheres to the three principles of data normalization: 1NF, 2NF and 3NF. Adopting this technique will ensure data integrity, minimize redundancy where fields are not reliant on each other to apply any function such as insert or delete data, and ensure effective data management. Data normalization is typically applied in 3 stages corresponding to 3 normal forms and taking into consideration the functional dependency and Armstrong's Inferential Rules. Explanation below:

The first normal form applied to a table, ensures data atomicity, which means that all values in each column contains inseparable values. Moreover, the full row of the table's metadata has no repeating groups where each column should hold a single value. In the database design submitted, all tables hold columns that possess atomic values.

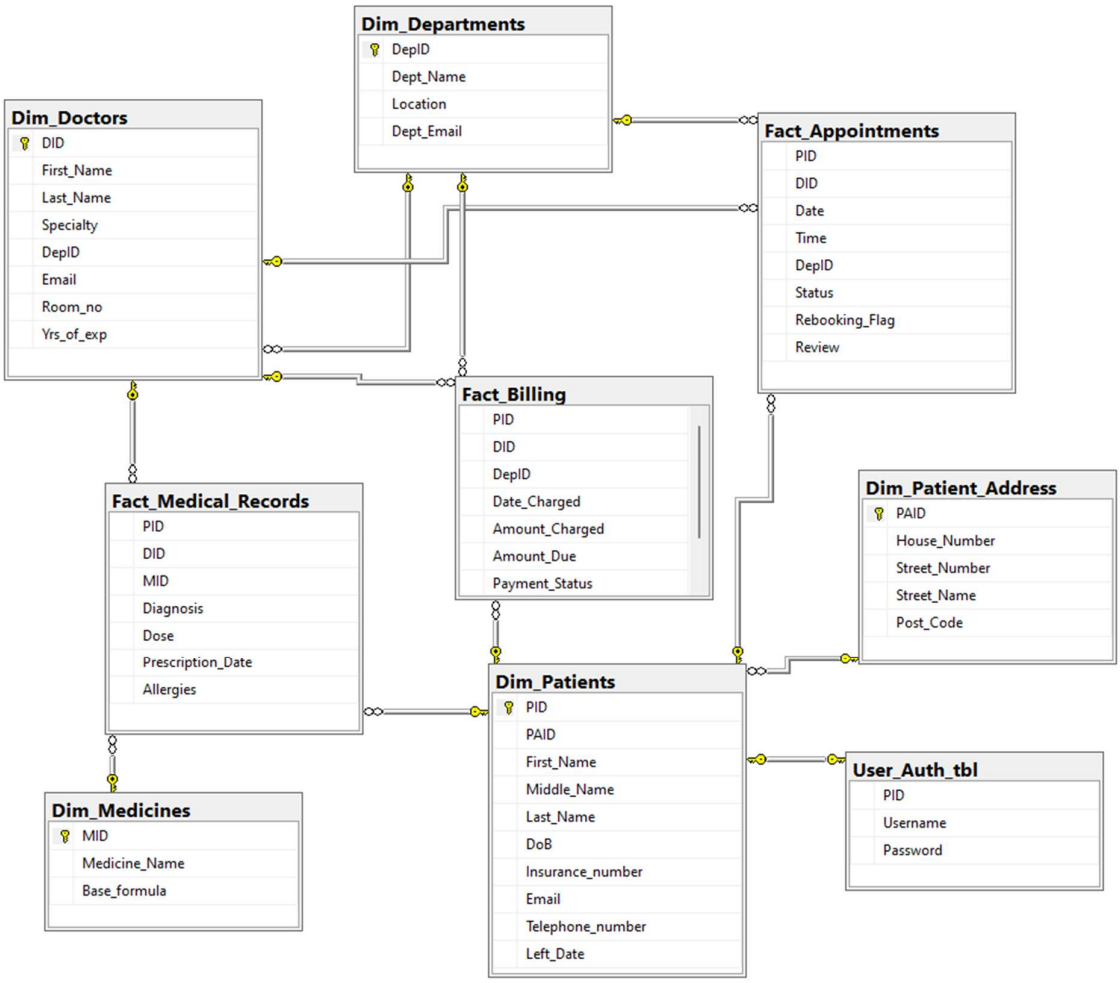
In the case of medical records, if a patient has two diagnoses with two different dates in one record only, they will be separated and identified as separate records, which in turn ensures the first normal form by achieving atomic values in each column. This prevents the "Insert Anomaly", and in turn would prevent "Delete Anomaly" and "Update Anomaly". Below the explanations is the screenshot for the tables created which shows that the first normal is achieved.

A table is in 2NF if it has already achieved and is in the first normal form, as well as it has achieved full functional dependency. This means that the dependant key, even if it is a composite one, is used as one full entity to identify other non-key attributes. In the case of the database design submitted, all tables were stripped down and divided where all non-key attributes are fully functionally dependant on the primary key of the table. In the appointment table, to be able to identify a specific appointment, the record is fully functionally dependent on the 3 foreign keys: Patient ID, Doctor ID, and Department ID: {PID, DID, DepID} → Appointment Record.

And a table finally achieves 3NF when there are no transitive dependencies based on Armstrong's Inferential Rule (IR3), which states and explains that non-key attributes must not depend on other non-key attributes. In another meaning, if a primary key of

a table can identify an entity, and in turn that entity can find a final entity, thus the primary key can identify the final entity and does not need the “middleman”.

The design provided separates different entities into different tables to eliminate transitive dependencies. For example, in the original medical records table, the patient id and doctor id can identify the medicine name, which can be identified at the same time by the medicine id: {PID, DID} → {MID} → {Medicine_name}. And to be able to avoid such thing, a medicine table was created which separated and broke down the transitive rule into the following: {PID, DID} → {MID} and {MID} → {Medicine_name}.



Due to the build and relationships between the tables, the data model that was used is the snowflake schema

Part 2

Task solutions:

2. Add the constraint to check that the appointment date is not in the past:

The above ask was already achieved in the table creation stage. A constraint was added on the table that will check that any inputted value in the date column is bigger or equal to the date and time of running the session, thus the use of GETDATE() function.

```
167 ALTER TABLE Fact_Appointments_Future
168 ADD CONSTRAINT Check_Date_Future
169 CHECK (Date >= GETDATE())
```

	PID	DID	Date	Time	DepID	Status	Rebooking_Flag	Review
1	7	103	2024-04-29	09:00:00.0000000	1001	Pending	0	NULL
2	5	102	2024-05-16	15:45:00.0000000	1003	Pending	0	NULL
3	2	104	2024-05-08	15:45:00.0000000	1002	Pending	0	NULL
4	4	102	2024-07-02	13:15:00.0000000	1001	Pending	0	NULL
5	2	104	2024-06-30	14:00:00.0000000	1002	Pending	0	NULL

3. List all the patients with older than 40 and have Cancer in diagnosis.

To be able to select the above information, first of all, a join was used to be able to access the diagnosis of the patients. An inner join specifically was used as the required only ask to see the records of the patients that have medical records. Then, two conditions have been added, the first to check that the patient's age is 40 and above, and the second is to check that the diagnosis is "cancer".

```
278 SELECT
279     dp.PID
280     ,dp.First_Name
281     ,dp.Last_Name
282     ,DATEDIFF(YEAR,dp.DoB,GETDATE()) AS Age
283     ,fmr.Diagnosis
284 FROM [dbo].[Dim_Patients] dp
285 INNER JOIN Fact_Medical_Records fmr
286     ON dp.PID = fmr.PID
287 WHERE DATEDIFF(YEAR,dp.DoB,GETDATE()) >= 40
288     and fmr.Diagnosis like '%cancer%'
```

	PID	First_Name	Last_Name	Age	Diagnosis
1	1	Alan	Rice	42	Cancer

4. A) Search the database of the hospital for matching character strings by name of medicine. Results should be sorted with most recent medicine prescribed date first.

A user function called "Seach_By_Medicine" was created to be able to fulfil the above requirement. To be able to achieve the acceptance criteria, an inner join was created between the medical records table and the medicines table. Once the variable for the user input was defined, it was then used as the condition for the

medicine name search to be able to get the output. Due to some technical issues in using only the “ORDER BY” function to be able to list the records based on the prescription date in a descending order, a “TOP 100 PERCENT” was added with the select statement to be able to execute the query. The example below will look at medical records with the medicine name of “Lumizol”.

```
291 CREATE FUNCTION Search_By_Medicine (@med_char_str NVARCHAR(255))
292 RETURNS TABLE
293 AS
294 RETURN (
295     SELECT TOP 100 PERCENT
296         fmr.PID,
297         fmr.Diagnosis,
298         mdc.Medicine_Name,
299         fmr.Dose,
300         fmr.Prescription_Date,
301         fmr.Allergies
302     FROM
303         [dbo].[Fact_Medical_Records] fmr
304     INNER JOIN Dim_Medicines mdc
305         ON mdc.MID = fmr.MID
306     WHERE
307         mdc.Medicine_Name LIKE '%' + @med_char_str + '%'
308     ORDER BY
309         fmr.Prescription_Date DESC
310 );
311
312 SELECT * FROM Search_By_Medicine('Lumizol')
```

Result:

	PID	Diagnosis	Medicine_Name	Dose	Prescription_Date	Allergies
1	2	Chronic Obstructive Pulmonary Disease (COPD)	Lumizol	500	2023-04-03	NULL
2	7	Peptic Ulcer Disease	Lumizol	300	2023-04-11	NULL

4. B) Return a full list of diagnosis and allergies for a specific patient who has an appointment today (i.e., the system date when the query is run)

In this example, a user-defined function named Patient_Current_Info is created to retrieve the current diagnosis and allergies for a specific patient, based on their first and last name. The function accepts two parameters: @patient_name and @patient_last_name, which represent the patient's first and last names, respectively. The main query is based on the medical records table to pull the diagnosis and the allergies of a selected patient. Thus, a condition was created on the patient ID where a subquery was created to specify the value of said condition. Moreover, the subquery is looking for the Patient ID where the first name and last name matches the @patient_name and @patient_last_name variable respectively, and due to the right join on the appointment table with the patient table, a final condition was added to check if the patient has an appointment date on the day the query is run.

```sql
CREATE FUNCTION Patient_Current_Info (@patient_name NVARCHAR(255), @patient_last_name NVARCHAR(255))
RETURNS TABLE
AS
RETURN (
  SELECT
    fmr.Diagnosis,
    fmr.Allergies
  FROM
    [dbo].[Fact_Medical_Records] fmr
  WHERE
    fmr.PID in (
      SELECT pt.PID
      FROM [dbo].[Dim_Patients] pt
      RIGHT JOIN [dbo].[Fact_Appointments] fa
        ON pt.PID = fa.PID
      WHERE First_Name like '%' + @patient_name + '%'
        and Last_Name like '%' + @patient_last_name + '%'
        and fa.Date = CAST(GETDATE() AS DATE)
    )
);

SELECT * FROM Patient_Current_Info('Chris', 'Frost')
```

Result:

	Diagnosis	Allergies
1	Asthma	Dust
2	Chronic Obstructive Pulmonary Disease (COPD)	NULL

4. C) Update the details for an existing doctor

A stored procedure named "Update_Doctor_Details" was developed to address the specific need for updating doctor information in the database system designed. To meet the acceptance criteria, a conditional logic was implemented to verify the existence of a doctor's record based on their unique identifier (Doctor ID), otherwise, the following error will pop up: “No doctor found with the provided Doctor ID”. Once the parameters for the doctor's details, such as first name, last name, specialty, department ID, email, room number, and years of experience were defined, they were used to update the doctor's information, with a special check to retain existing values if the input parameter is left blank or, in the case of years of experience, set to a negative value to indicate no change. This method allows for selective updating of doctor details, ensuring data integrity and flexibility in data management. In the below example, for the doctor id 107, the specialty is changing to gastroenterologists, the room number to 120D and the years of experience to 3. As the other fields were not specified, it will pick up the original data.

```sql
CREATE PROCEDURE Update_Doctor_Details
    @DID INT,
    @FirstName NVARCHAR(255),
    @LastName NVARCHAR(255),
    @Specialty NVARCHAR(255),
    @DepID INT,
    @Email NVARCHAR(255),
    @Room_no NVARCHAR(255),
    @Yrs_of_exp INT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM [dbo].[Dim_Doctors] WHERE DID = @DID) -- or = 1
    BEGIN
        UPDATE [dbo].[Dim_Doctors]
        SET
            First_Name = IIF(@FirstName = '', (SELECT First_Name FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @FirstName),
            Last_Name = IIF(@LastName = '', (SELECT Last_Name FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @LastName),
            Specialty = IIF(@Specialty = '', (SELECT Specialty FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @Specialty),
            DepID = IIF(@DepID = '', (SELECT DepID FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @DepID),
            Email = IIF(@Email = '', (SELECT Email FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @Email),
            Room_no = IIF(@Room_no = '', (SELECT Room_no FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @Room_no),
            Yrs_of_exp = IIF(@Yrs_of_exp = -1, (SELECT Yrs_of_exp FROM [dbo].[Dim_Doctors] WHERE DID = @DID), @Yrs_of_exp)
        WHERE
            DID = @DID;
    END
    ELSE
    BEGIN
        RAISERROR ('No doctor found with the provided Doctor ID.', 16, 1);
    END
END
GO
```

```sql
EXEC Update_Doctor_Details
    @DID = 107,
    @FirstName = '',
    @LastName = '',
    @Specialty = 'Gastroenterologists',
    @DepID = '',
    @Email = '',
    @Room_no = '120D',
    @Yrs_of_exp = 3;
```

Before:

	DID	First_Name	Last_Name	Specialty	DepID	Email	Room_no	Yrs_of_exp
1	107	Prashant	Patel	Pediatrician	1004	Prashant.P@george-hosp.com	120F	13

After:

	DID	First_Name	Last_Name	Specialty	DepID	Email	Room_no	Yrs_of_exp
1	107	Prashant	Patel	Gastroenterologists	1004	Prashant.P@george-hosp.com	120D	3

4. D) Delete the appointment who status is already completed.

A stored procedure titled “Delete_Completed_Appointments” was created and stored in the database. A delete command was used to erase records from the appointments table in the database. The main part of this operation is a condition that only targets records that have a status of 'Completed'.

```
382 CREATE PROCEDURE Delete_Completed_Appointments AS
383 BEGIN
384     DELETE FROM [dbo].[Fact_Appointments]
385     WHERE [Status] = 'Completed'
386 END
387 GO
388
389 EXEC Delete_Completed_Appointments
```

Before:

	PID	DID	Date	Time	DepID	Status	Rebooking_Flag	Review
1	6	105	2024-01-09	13:15:00.0000000	1005	Completed	0	NULL
2	7	103	2024-01-12	10:45:00.0000000	1001	Completed	0	I waited for such a while!
3	3	106	2024-01-15	11:15:00.0000000	1004	Completed	0	Thank you
4	9	105	2024-01-27	14:45:00.0000000	1002	Cancelled	1	NULL
5	9	102	2024-01-29	12:45:00.0000000	1004	Completed	0	Friendly staff
6	9	104	2024-01-30	14:30:00.0000000	1002	Completed	0	Great experience, frien...
7	1	101	2024-02-11	11:00:00.0000000	1001	Completed	0	Thank you Dr.
8	4	102	2024-02-12	13:00:00.0000000	1001	Completed	0	NULL
9	1	101	2024-02-15	11:30:00.0000000	1001	Cancelled	1	NULL
10	7	103	2024-04-19	09:00:00.0000000	1001	Pending	0	NULL
11	5	102	2024-04-22	15:45:00.0000000	1003	Pending	0	NULL
12	5	104	2024-04-29	16:15:00.0000000	1003	Cancelled	1	NULL
13	2	104	2024-03-17	15:45:00.0000000	1002	Pending	0	NULL
14	4	102	2024-05-11	13:15:00.0000000	1001	Pending	0	NULL
15	6	101	2024-06-26	13:30:00.0000000	1003	Cancelled	1	NULL
16	2	104	2024-06-30	14:00:00.0000000	1002	Pending	0	NULL

After:

	PID	DID	Date	Time	DepID	Status	Rebooking_Flag	Review
1	9	105	2024-01-27	14:45:00.0000000	1002	Cancelled	1	NULL
2	1	101	2024-02-15	11:30:00.0000000	1001	Cancelled	1	NULL
3	7	103	2024-04-19	09:00:00.0000000	1001	Pending	0	NULL
4	5	102	2024-04-22	15:45:00.0000000	1003	Pending	0	NULL
5	5	104	2024-04-29	16:15:00.0000000	1003	Cancelled	1	NULL
6	2	104	2024-03-17	15:45:00.0000000	1002	Pending	0	NULL
7	4	102	2024-05-11	13:15:00.0000000	1001	Pending	0	NULL
8	6	101	2024-06-26	13:30:00.0000000	1003	Cancelled	1	NULL
9	2	104	2024-06-30	14:00:00.0000000	1002	Pending	0	NULL

5. The hospital wants to view the appointment date and time, showing all previous and current appointments for all doctors, and including details of the department (the doctor is associated with), doctor's specialty and any associate review/feedback given for a doctor. You should create a view containing all the required information.

A create view function was created to achieve the required. It starts with fields from the appointment table, and then an INNER JOIN with the doctor's table to make sure the records between the two tables match, and finally a LEFT JOIN with the department table, which will pull the information from the other tables, and only the

matching record from the department table. In terms of fields, to be able to display the full name of the doctor, the abbreviation Dr was added, with a space, then the doctor's first name, another space and then the last name. The square brackets were used to be able to use the space character in the column name. Finally, a condition was added to the query which looks only for appointments older or same day of the day the query was run.

```
393 CREATE VIEW all_appointments AS
394 SELECT
395     fa.Date AS [Appointment Date],
396     fa.Time AS [Appointment Time],
397     'Dr. ' + dd.First_Name + ' ' + dd.Last_Name AS [Doctor Name],
398     ddep.Dept_Name AS [Department Name],
399     ddep.Location AS [Hospital Wing],
400     dd.Room_no AS [Examination Room],
401     dd.Specialty AS [Dr's Specialty],
402     fa.Review AS Feedback
403 FROM [dbo].[Fact_Appointments] fa
404 INNER JOIN [dbo].[Dim_Doctors] dd
405     ON fa.DID = dd.DID
406 LEFT JOIN [dbo].[Dim_Departments] ddep
407     ON ddep.DepID = dd.DepID
408 WHERE fa.Date <= GetDate()
409
410 SELECT *
411 FROM [dbo].[all_appointments]
```

Result:

	Appointment Date	Appointment Time	Doctor Name	Department Name	Hospital Wing	Examination Room	Dr's Specialty	Feedback
1	2024-01-27	14:45:00.0000000	Dr. Dylan Carlson	DepartmentC	Wing_3	126F	Pediatrician	NULL
2	2024-02-15	11:30:00.0000000	Dr. Jared Night	DepartmentA	Wing_1	114A	Cardiologist	NULL
3	2024-03-17	15:45:00.0000000	Dr. Ben Wood	DepartmentB	Wing_2	120C	Gastroenterologists	NULL

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

A create trigger function "Status_Change" was used to achieve the acceptance criteria. The trigger of said function is the "Update" statement, where when u ser apply an update function on the table, the trigger will get actioned and then it will itself apply an update function where it will change the status of the appointment from "Available" to "Cancelled". A standard update function was applied to force trigger the "Status_Change" as shown below.

```
414 CREATE TRIGGER Status_Change ON [dbo].[Fact_Appointments]
415 AFTER UPDATE AS
416 BEGIN
417     SET NOCOUNT ON;
418     IF UPDATE(Status)
419     BEGIN
420         UPDATE [dbo].[Fact_Appointments]
421         SET Status = 'Available'
422         FROM [dbo].[Fact_Appointments] fa
423         WHERE fa.Status = 'Cancelled';
424     END
425 END;

428 UPDATE [dbo].[Fact_Appointments]
429 SET Status = 'Cancelled'
430 WHERE PID = 9
431     and DID = 105
432     and DepID = 1002
```

Before:

	PID	DID	Date	Time	DepID	Status	Rebooking_Flag	Review
1	9	105	2024-01-27	14:45:00.0000000	1002	Cancelled	1	NULL
2	1	101	2024-02-15	11:30:00.0000000	1001	Cancelled	1	NULL
3	7	103	2024-04-19	09:00:00.0000000	1001	Pending	0	NULL
4	5	102	2024-04-22	15:45:00.0000000	1003	Pending	0	NULL
5	5	104	2024-04-29	16:15:00.0000000	1003	Cancelled	1	NULL
6	2	104	2024-03-17	15:45:00.0000000	1002	Pending	0	NULL
7	4	102	2024-05-11	13:15:00.0000000	1001	Pending	0	NULL
8	6	101	2024-06-26	13:30:00.0000000	1003	Cancelled	1	NULL
9	2	104	2024-06-30	14:00:00.0000000	1002	Pending	0	NULL

After:

	PID	DID	Date	Time	DepID	Status	Rebooking_Flag	Review
1	9	105	2024-01-27	14:45:00.0000000	1002	Available	1	NULL
2	1	101	2024-02-15	11:30:00.0000000	1001	Available	1	NULL
3	7	103	2024-04-19	09:00:00.0000000	1001	Pending	0	NULL
4	5	102	2024-04-22	15:45:00.0000000	1003	Pending	0	NULL
5	5	104	2024-04-29	16:15:00.0000000	1003	Available	1	NULL
6	2	104	2024-03-17	15:45:00.0000000	1002	Pending	0	NULL
7	4	102	2024-05-11	13:15:00.0000000	1001	Pending	0	NULL
8	6	101	2024-06-26	13:30:00.0000000	1003	Available	1	NULL
9	2	104	2024-06-30	14:00:00.0000000	1002	Pending	0	NULL

7. Write a select query which allows the hospital to identify the number of completed appointments with the specialty of doctors as ‘Gastroenterologists’.

The above ask was completed in two diverse ways; using joins and a subquery. The first used the join between appointments table and doctors, where it will count the doctor’s ID, specifically from the appointments table, which means how many times they had an appointment assigned to them, with two main conditions: the status of the

appointment is complete, and the doctor's specialty is Gastro. The Subquery option yields a similar result, but instead of the join and the Gastro filter, there is a filter on the Doctor ID where it will check if it is within a list created by a subquery that pulls the doctors IDs from the doctor table which have Gastro as the specialty.

```
434  ----- Option 1 ----- JOIN
435  SELECT
436      'Count of Appointments where Dr specialty is Gastroenterologists: ' AS Detail
437      ,COUNT(fa.DID) AS [Count]
438  FROM [dbo].[Fact_Appointments] fa
439  INNER JOIN [dbo].[Dim_Doctors] dd
440      ON fa.DID = dd.DID
441  WHERE fa.Status = 'Completed'
442      and dd.Specialty like '%Gastroenterologists%'
443
444  ----- Option 2 ----- SUB-QUERY
445  SELECT
446      'Count of Appointments where Dr specialty is Gastroenterologists: ' AS Detail
447      ,COUNT(DID) AS [Count]
448  FROM [dbo].[Fact_Appointments]
449  WHERE Status = 'Completed'
450      and DID in (
451      SELECT DID
452      FROM [dbo].[Dim_Doctors]
453      WHERE Specialty like '%Gastroenterologists%'
454      )
```

Result:

	Detail	Count
1	Count of Appointments where Dr specialty is Gastroenterologists:	2

Advice and guidance:

Data integrity and concurrency

- Data validation rules: It includes constraints, as well as checks for data integrity for the data sent to the tables in the database designed. In addition to setting the corresponding columns as Primary Keys, Foreign Keys, specifying data types and if it is a nullable filed or not. For example, most of the mentioned validations have been used in creating the tables which ensure consistent information submission and information management. All figures from the previous section “tables explanation” shows the mentioned.
- Transaction management: It ensure the synchronization of the database and the transactions being applied, as well as the data inputted into the database. The end-to-end transaction life should follow ACID properties: atomicity, consistency, isolation, durability. Thus, a rollback strategy would ensure that. For example, if the following flow of actions exists; update status of appointment to complete, create medical record, prescribe a medicine, add the prescription date. If an error occurs in the described series of actions, everything will revert back to the initial stage.
- Database roles: Creating database roles and assigning them specific access is a remarkably effective way of ensuring database concurrency. Then the database administrator can assign specific employees to corresponding groups. The roles can have anything from “SELECT” access to “DELETE” access and allows different user to apply the allowed functions on specific tables specified. For example, the HR department in the hospital can be assigned “DELETE” access on the doctor’s table. In addition, the administration team can as well have “DELETE” access on the patient’s table. On the other hand, patients can only have “UPDATE” access to the patients table same as the doctors that can have “UPDATE” access on the doctors’ table.

Database security

- Authentication and MFA: setting up the correct authentication and access for all relevant personnel who will access the database. Some of the considerations would be creating a username and password, specifying which database, schema, tables the DBA can access, and ready/write access on each specified access. An additional security layer is the MFA which stands for multi-factor authentication which means that the DBA will need to verify that it is actually him accessing his account even though his password is correct.

```
471 CREATE LOGIN Chadi_Ghosn
472 WITH PASSWORD = 'Salford2024!'
473
474 CREATE USER Chadi_Ghosn
475 FOR LOGIN Chadi_Ghosn
476
477 CREATE ROLE AdminCenter
478 GRANT SELECT, UPDATE, DELETE, INSERT ON [dbo].[Dim_Departments] TO AdminCenter
479
480 ALTER ROLE AdminCenter ADD MEMBER Chadi_Ghosn
```

- Encryption and decryption: This is a two-way street that ensures the security of the database, as when encrypted, it transforms the database information into an unreadable format, and once decrypted with the correct key, it goes back to the original, readable state. There are specific algorithms used to be able to achieve the encryption and decryption. In the case of the hospital database, once the DBA creates a backup of the database, it should be encrypted in case of an unauthorised access to the backup.
- Auditing and logging: These methods ensure recording and tracking activities that take place on the database, i.e. errors, queries, or transactions. Doing so can help the DBA of the hospital with monitoring, compliance and debugging. For example, the DBA can track any unintentional changes that happens on the tables created, including but not limited to billing, patients, doctors, and then revert back the information.
- SQL injection considerations: One of the main concerns for database security, is to have a secure login portal. Thus, the worries from the vulnerability called “SQL Injection”. This vulnerability allows the hacker to be able to access the account of a user through bypassing the password field using a SQL injection trick. To mitigate this, an advised method is to use parameterized statements, which means that instead of using the input to directly build the statement, the code will use a JDBC connector to SQL which will as well store the password input in a variable to build the connection string.

Database backup and recovery

- Frequency and schedule of backups: based on the size of the data, volatility and worth, a backup strategy is established. In our case, the database should be backed up daily due to the importance of the data. Though, an incremental backup will be setup for an efficient back up strategy. Business continuity and disaster recovery (BCDR) is a cloud strategy that is available to take advantage of, which provides a backup and restoration in case of failure.
- The backup destination and format: backups should be stored in a secure way and ideally on the cloud with more than one region redundancy in case of problems. And the backup should be of course encrypted in case of security breach.
- The recovery strategy and testing: As a hospital database holds extremely important data, there should be a mirror database RAID in standby mode which in case of issue, will take a limited and short amount of time before the hospital is back online.

Extra queries:

Query 1:

The following query will help update billing information for a specific patient.

```
CREATE PROCEDURE Update_Billing
    @PID INT,
    @DID INT,
    @DepID INT,
    @Date_Charged DATE,
    @Amount_Charged DECIMAL,
    @Amount_Due DECIMAL,
    @Payment_Status VARCHAR(10),
    @Date_updated DATE
AS
BEGIN
    IF EXISTS (SELECT 1 FROM [dbo].[Fact_Billing] WHERE PID = @PID )
    BEGIN
        UPDATE [dbo].[Fact_Billing]
        SET
            DID = IIF(@DID = '', (SELECT DID FROM [dbo].[Fact_Billing] WHERE PID = @PID), @DID),
            DepID = IIF(@DepID = '', (SELECT DepID FROM [dbo].[Fact_Billing] WHERE PID = @PID), @DepID),
            Date_Charged = IIF(@Date_Charged = '', (SELECT Date_Charged FROM [dbo].[Fact_Billing] WHERE PID = @PID), @Date_Charged),
            Amount_Charged = IIF(@Amount_Charged = -1, (SELECT Amount_Charged FROM [dbo].[Fact_Billing] WHERE PID = @PID), @Amount_Charged),
            Amount_Due = IIF(@Amount_Due = -1, (SELECT Amount_Due FROM [dbo].[Fact_Billing] WHERE PID = @PID), @Amount_Due),
            Payment_Status = IIF(@Payment_Status = '', (SELECT Payment_Status FROM [dbo].[Fact_Billing] WHERE PID = @PID), @Payment_Status),
            Date_updated = IIF(@Date_updated = '', (SELECT Date_updated FROM [dbo].[Fact_Billing] WHERE PID = @PID), @Date_updated)
        WHERE
            PID = @PID;
    END
    ELSE
    BEGIN
        RAISERROR ('No patient found with the provided patient ID', 16, 1);
    END
END

EXEC Update_Billing
    @PID = 3,
    @DID = 102,
    @DepID = '',
    @Date_Charged = '',
    @Amount_Charged = 2729,
    @Amount_Due = 634,
    @Payment_Status = '',
    @Date_updated = '2024-04-01'

SELECT * FROM [dbo].[Fact Billing]
```

Before:

	PID	DID	DepID	Date_Charged	Amount_Charged	Amount_Due	Payment_Status	Date_updated
1	5	105	1001	2023-03-12	1598	985	Partial	2023-05-27
2	2	101	1001	2022-06-29	1197	1197	Unpaid	NULL
3	3	102	1001	2022-09-02	2729	1639	Partial	2023-12-28
4	4	103	1001	2023-09-12	1914	0	Paid	NULL
5	1	106	1001	2023-04-18	5243	5243	Unpaid	2023-11-18
6	7	104	1001	2022-11-05	943	269	Partial	NULL
7	6	107	1001	2023-10-20	2264	0	Paid	NULL

After

	PID	DID	DepID	Date_Charged	Amount_Charged	Amount_Due	Payment_Status	Date_updated
1	5	105	1001	2023-03-12	1598	985	Partial	2023-05-27
2	2	101	1001	2022-06-29	1197	1197	Unpaid	NULL
3	3	102	1001	2022-09-02	2729	634	Partial	2024-04-01
4	4	103	1001	2023-09-12	1914	0	Paid	NULL
5	1	106	1001	2023-04-18	5243	5243	Unpaid	2023-11-18
6	7	104	1001	2022-11-05	943	269	Partial	NULL
7	6	107	1001	2023-10-20	2264	0	Paid	NULL

Query 2:

Creating a view for patient billing information

```
CREATE VIEW Patient_Billing AS
SELECT
    CONCAT(p.First_Name, ' ',p.Last_Name) AS [Patient Name],
    CONCAT('Dr. ', d.First_Name, ' ', d.Last_Name) AS [Doctor Name],
    d.Specialty AS [Doctor Speciality],
    b.Date_Charged AS [Date Charged],
    FORMAT(b.Amount_Charged, 'C', 'en-gb') AS [Amount Charged],
    FORMAT(b.Amount_Due, 'C', 'en-gb') AS [Amount Due],
    b.Payment_Status AS [Payment Status]
FROM [dbo].[Fact_Billing] b
LEFT JOIN [dbo].[Dim_Patients] p
    ON b.PID = p.PID
LEFT JOIN [dbo].[Dim_Doctors] d
    ON d.DID = b.DID

SELECT * FROM Patient_Billing
```

Result:

	Patient Name	Doctor Name	Doctor Speciality	Date Charged	Amount Charged	Amount Due	Payment Status
1	Ade Awonaike	Dr. Dylan Carlson	Pediatrician	2023-03-12	£1,598.00	£985.00	Partial
2	Chris Frost	Dr. Jared Night	Cardiologist	2022-06-29	£1,197.00	£1,197.00	Unpaid
3	Kishan Chauhan	Dr. Phil Greenwood	Dermatologist	2022-09-02	£2,729.00	£634.00	Partial
4	George Mokhtar	Dr. Claire Dunphy	Neurologist	2023-09-12	£1,914.00	£0.00	Paid
5	Alan Rice	Dr. Dan Barlow	Gastroenterologists	2023-04-18	£5,243.00	£5,243.00	Unpaid
6	Dan Waterman	Dr. Ben Wood	Gastroenterologists	2022-11-05	£943.00	£269.00	Partial
7	Zac Hosn	Dr. Prashant Patel	Gastroenterologists	2023-10-20	£2,264.00	£0.00	Paid

Query 3:

Return patient's prescriptions

```
CREATE FUNCTION Patient_Prescriptions (@patient_name NVARCHAR(255), @patient_last_name NVARCHAR(255))
RETURNS TABLE
AS
RETURN (
    SELECT
        fmr.Diagnosis,
        md.Medicine_Name,
        fmr.Dose,
        fmr.Prescription_Date
    FROM
        [dbo].[Fact_Medical_Records] fmr
    INNER JOIN [dbo].[Dim_Medicines] md
        ON fmr.MID = md.MID
    WHERE
        fmr.PID in (
            SELECT pt.PID
            FROM [dbo].[Dim_Patients] pt
            RIGHT JOIN [dbo].[Fact_Appointments] fa
                ON pt.PID = fa.PID
            WHERE First_Name like  '%' + @patient_name + '%'
                and Last_Name like '%' + @patient_last_name + '%'
        )
);

SELECT * FROM Patient_Prescriptions('Chris', 'Frost')
```

Result:

	Diagnosis	Medicine_Name	Dose	Prescription_Date
1	Asthma	AlliumMend	100	2022-05-21
2	Chronic Obstructive Pulmonary Disease (COPD)	Lumizol	500	2023-04-03

Query 4:

A TRIGGER on update, that will delete patient's information, if they have left the hospital for 3 years or more.

```
564 CREATE TRIGGER Remove_Expired_Patients ON [dbo].[Dim_Patients]
565 AFTER DELETE AS
566 BEGIN
567     DELETE FROM [dbo].[Dim_Patients]
568     WHERE ABS(DATEDIFF(year, Left_Date, GETDATE())) >= 3
569 END
```

Trigger:

```
571 UPDATE [dbo].[Dim_Patients]
572 SET Left_Date = '2021-01-26'
573 WHERE PID = 21
```

Before:

	PID	First_Name	Last_Name	Left_Date	Years_since_left
1	1	Alan	Rice	NULL	NULL
2	2	Chris	Frost	NULL	NULL
3	3	Kishan	Chauhan	NULL	NULL
4	4	George	Mokhtar	NULL	NULL
5	5	Ade	Awonaike	NULL	NULL
6	6	Zac	Hosn	2023-09-12	1
7	7	Dan	Waterman	2022-03-28	2
8	8	Aquilla	Matafwali	2023-11-19	1
9	9	Tamsin	Anderson	2022-07-02	2
10	10	Claire	Hosn	NULL	NULL
11	21	Chadi	Ghosn	2021-01-26	3

After:

	PID	First_Name	Last_Name	Left_Date	Years_since_left
1	1	Alan	Rice	NULL	NULL
2	2	Chris	Frost	NULL	NULL
3	3	Kishan	Chauhan	NULL	NULL
4	4	George	Mokhtar	NULL	NULL
5	5	Ade	Awonaike	NULL	NULL
6	6	Zac	Hosn	2023-09-12	1
7	7	Dan	Waterman	2022-03-28	2
8	8	Aquilla	Matafwali	2023-11-19	1
9	9	Tamsin	Anderson	2022-07-02	2
10	10	Claire	Hosn	NULL	NULL

Query 5:

Billing information of patients where the Doctor's specialty in Gastroenterology. The query uses JOIN and Subquery.

```
584 SELECT
585     p.First_Name,
586     p.Last_Name,
587     b.Date_Charged,
588     b.Amount_Due
589 FROM [dbo].[Fact_Billing] b
590 INNER JOIN [dbo].[Dim_Patients] p
591     ON p.PID = b.PID
592 WHERE b.DID in (
593     SELECT DID
594     FROM [dbo].[Dim_Doctors]
595     WHERE Specialty LIKE 'Gastro%'
596 )
```

Result:

	First_Name	Last_Name	Date_Charged	Amount_Due
1	Alan	Rice	2023-04-18	5243
2	Dan	Waterman	2022-11-05	269
3	Zac	Hosn	2023-10-20	0

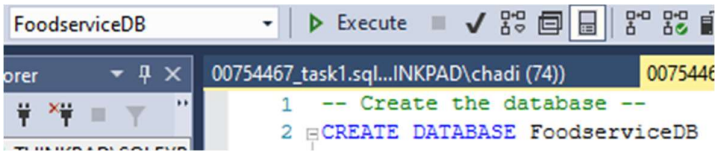
In conclusion, the database design, which was created, following the snowflake schema, will be able to achieve all of the hospital's requirements. Security measures, such as authentication, authorization and logging were advised on, as well as backup and recovery. Moreover, constraints were set up throughout table creation for quality assurance.

Task 2 Report

Information and data were provided to the database consultant, working in a food and service company, to be able to help and set up the database, and query it to achieve the necessary results. The tables, which were provided as comma separated value spreadsheets, were 4 in total, covering restaurant names, location, and details, as well as information and details about the consumers and a reference to which restaurants they have been in. In addition, one of the tables contains ratings that the consumers provided to the restaurants and the fourth table indicates the cuisines of the restaurants. The database will be created with primary and foreign keys in place to set up the necessary relationships between the tables.

Part 1

The first step is to create a new query and run the command which will create the database that is needed, and the name will be FoodserviceDB



Then, a dedicated schema was created for the database where all the tables will be assigned to. There are several advantages of using a schema, thus the use of it in this database, which will allow the admin to logically group the tables, assign permissions on schema level as well as revoking or restricting access.

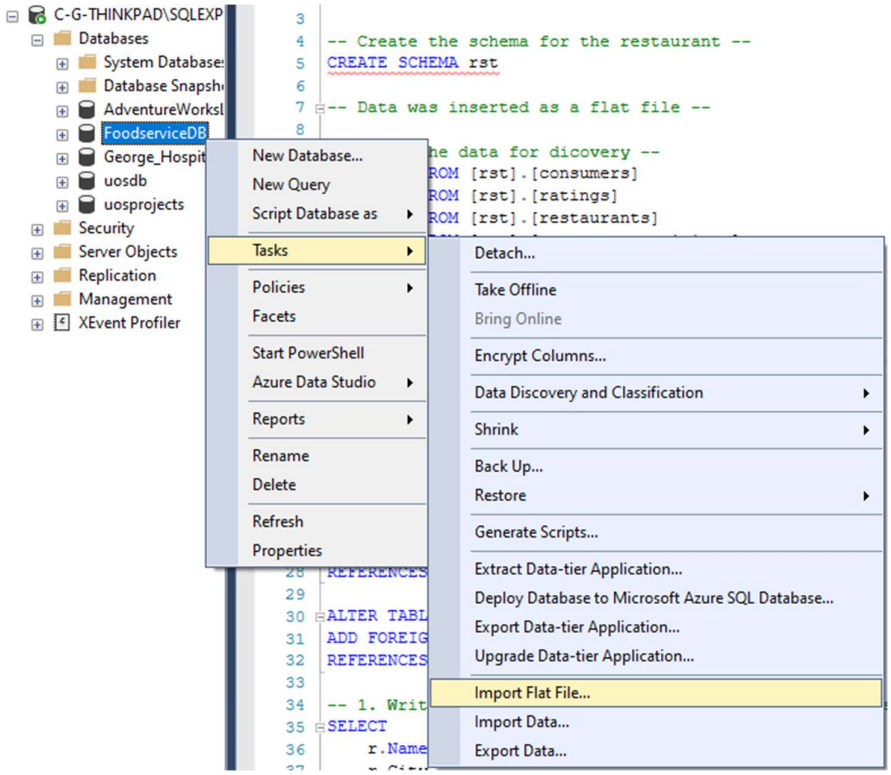
```
4  -- Create the schema for the restaurant --
5  CREATE SCHEMA rst
```

There are several ways to import data from a csv file into the database created, from using a function in SQL, to using a python library such as SQL Alchemy, to using the interface which was actually used in this case. By right clicking on the correct database name in the list of databases and choosing the option of “Tasks” and then clicking on “Import Flat File”, the admin can then follow simple steps to ensure the data types and data quality and import the csv into a SQL table. The admin is able as well to choose the schema where data will be assigned. The screenshots below will show the following points step by step:

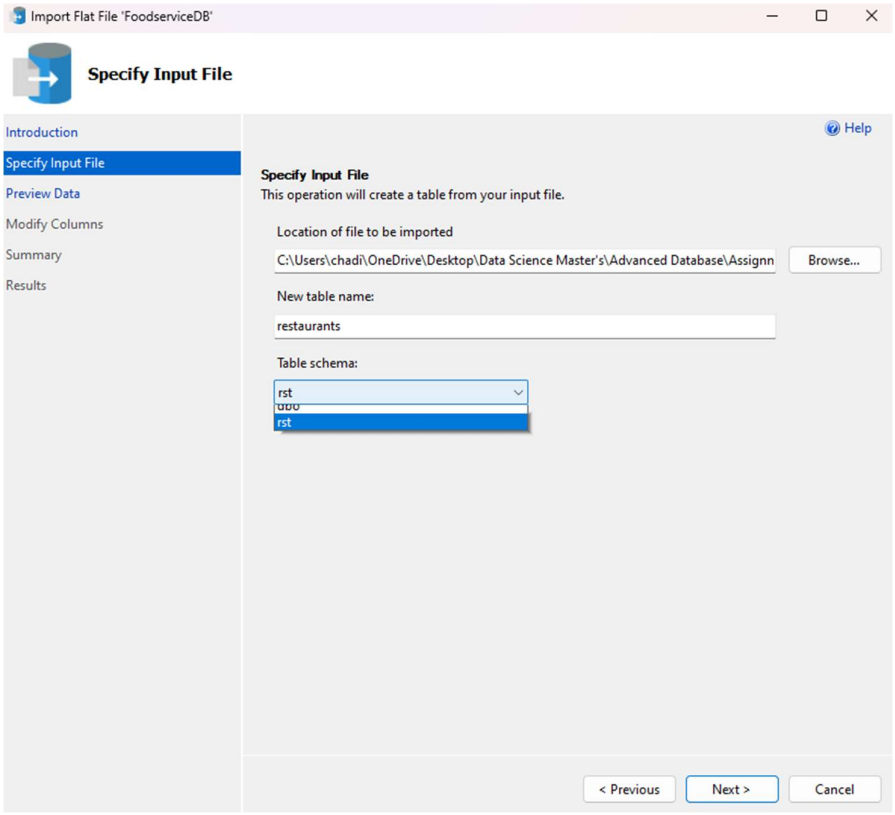
- Step 1: Launch the task “Import Flat File”
- Step 2: Browse the file and choose the schema
- Step 3: Modify columns data types, null option, and primary key

This process needs to be repeated for all four tables.

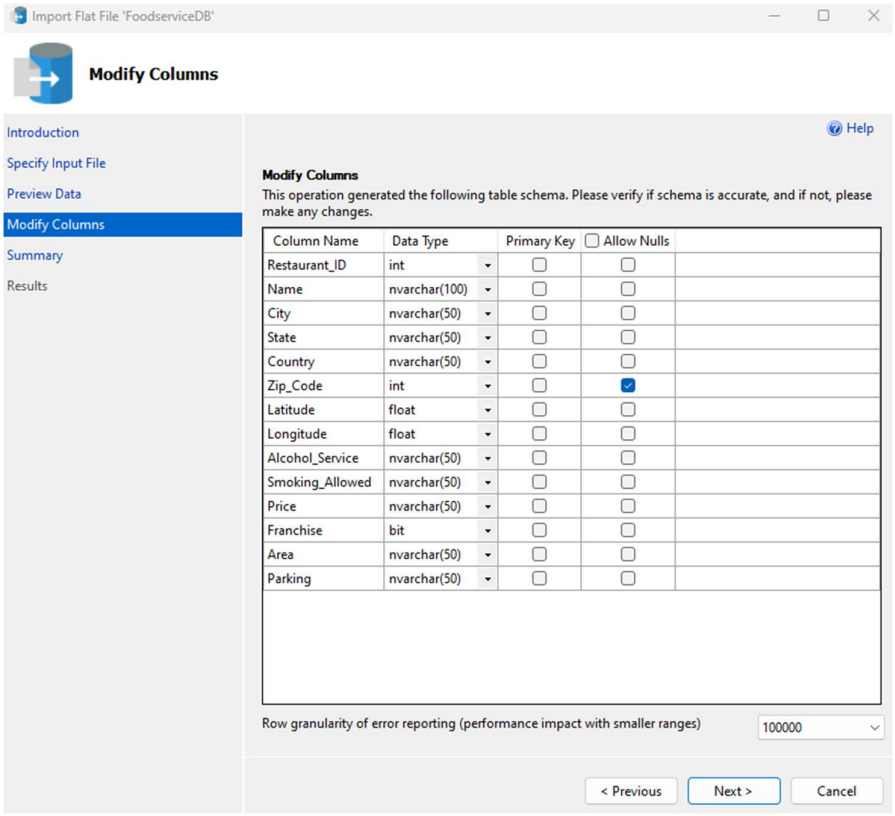
Step 1:


C-G-THINKPAD\SQLEXP
Databases
System Database
Database Snapsh
AdventureWorksl
FoodserviceDB
George_Hospit
uosdb
uosprojects
Security
Server Objects
Replication
Management
XEvent Profiler
-- Create the schema for the restaurant --
CREATE SCHEMA rst
-- Data was inserted as a flat file --
he data for dicovery --
ROM [rst].[consumers]
ROM [rst].[ratings]
ROM [rst].[restaurants]
REFERENCES
ALTER TABL
ADD FOREIG
REFERENCES
-- 1. Writ
SELECT
r.Name
New Database...
New Query
Script Database as
Tasks
Policies
Facets
Start PowerShell
Azure Data Studio
Reports
Rename
Delete
Refresh
Properties
Detach...
Take Offline
Bring Online
Encrypt Columns...
Data Discovery and Classification
Shrink
Back Up...
Restore
Generate Scripts...
Extract Data-tier Application...
Deploy Database to Microsoft Azure SQL Database...
Export Data-tier Application...
Upgrade Data-tier Application...
Import Flat File...
Import Data...
Export Data...

Step 2:


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\chadi\OneDrive\Desktop\Data Science Master's\Advanced Database\Assignn
Browse...
New table name:
restaurants
Table schema:
rst
dbo
rst
< Previous
Next >
Cancel

Step 3:



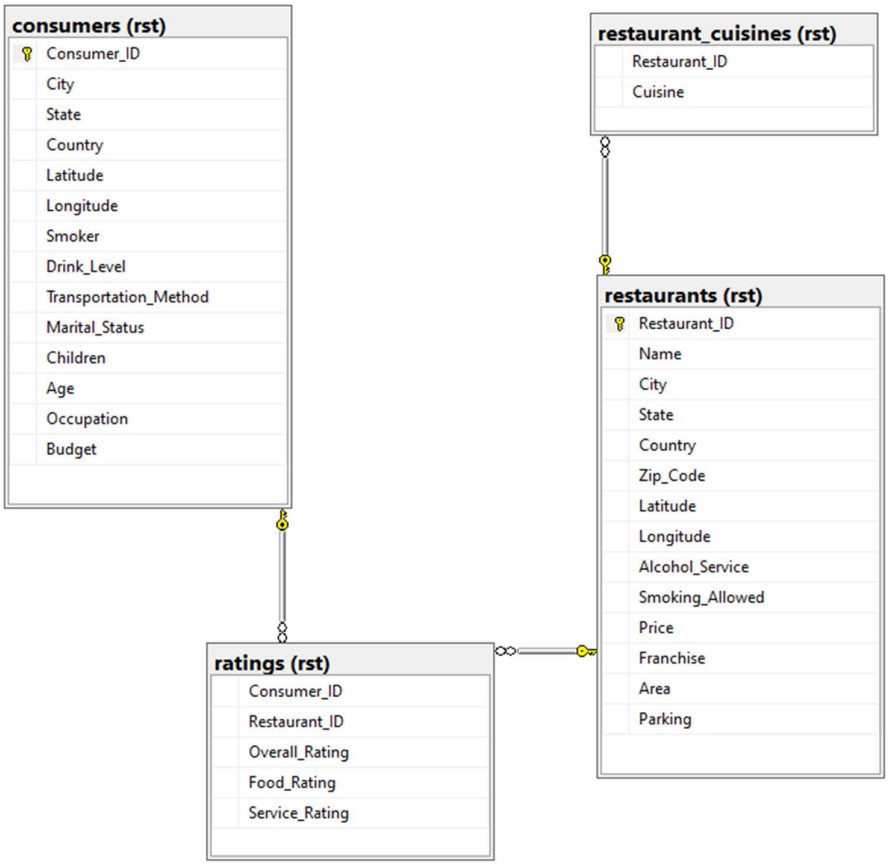
After the insertion of the data into the database, the discovery phase starts to be able to understand the data and creating the semantic model, where the primary key and foreign key are set to create the entity relational diagram.

```
-- Query the data for dicovery --
SELECT * FROM [rst].[consumers]
SELECT * FROM [rst].[ratings]
SELECT * FROM [rst].[restaurants]
SELECT * FROM [rst].[restaurant_cuisines]
```

After investigating the data, there were two main primary keys identified in the tables; Consumer ID in the consumers table and the Restaurant ID in the restaurants table. Once the PKs are set, the foreign key could now be set up which will reference a primary in another table. As such three foreign keys were assigned; Restaurant ID in the ratings table, Restaurant ID in the restaurant's cuisine table and the Consumer ID in the ratings table.

```sql
-- Add the primary keys first then the foreign ones --
ALTER TABLE [rst].[consumers]
ADD PRIMARY KEY (Consumer_ID);

ALTER TABLE [rst].[ratings]
ADD FOREIGN KEY (Consumer_ID)
REFERENCES [rst].[consumers](Consumer_ID);

ALTER TABLE [rst].[restaurants]
ADD PRIMARY KEY (Restaurant_ID);

ALTER TABLE [rst].[restaurant_cuisines]
ADD FOREIGN KEY (Restaurant_ID)
REFERENCES [rst].[restaurants](Restaurant_ID);

ALTER TABLE [rst].[ratings]
ADD FOREIGN KEY (Restaurant_ID)
REFERENCES [rst].[restaurants](Restaurant ID);
```

Finally, based on the model that was created after assigning the appropriate keys, a clear snowflake schema is built where the dimensions tables are restaurants, restaurant cuisines and consumers, and the factual table is the ratings.



Part 2

1. Write a query that lists all restaurants with a “Medium” range price with open area, serving Mexican food.

A simple select query was created to achieve the above ask. Most of the fields required reside in one table, restaurants, but the restaurant cuisine field needed is in the restaurant cuisines table, thus the INNER JOIN between the two table on the Restaurant ID field. Now that the data is queried, the conditions will be added in the WHERE clause to filter non required information. As per the ask above, the Price should be medium, the area should be open, and the cuisine should be Mexican.

Query:

```
35  SELECT
36      r.Name,
37      r.City,
38      r.State,
39      r.Country,
40      r.Price,
41      r.Area,
42      rc.Cuisine
43  FROM [rst].[restaurants] r
44  INNER JOIN [rst].[restaurant_cuisines] rc
45      ON r.Restaurant_ID = rc.Restaurant_ID
46  WHERE r.Price = 'Medium'
47      and r.Area = 'Open'
48      and rc.Cuisine = 'Mexican'
```

Result:

	Name	City	State	Country	Price	Area	Cuisine
1	El Oceano Dorado	Cuernavaca	Morelos	Mexico	Medium	Open	Mexican
2	El Rincón De San Francisco	San Luis Potosi	San Luis Potosi	Mexico	Medium	Open	Mexican

2. Write a query that returns the total number of restaurants who have the overall rating as 1 and are serving Mexican food. Compare the results with the total number of restaurants who have the overall rating as 1 serving Italian food (please give explanations on their comparison)

To be able to achieve the above ask, as well as provide a terrific way to display the information, a UNION ALL function was used between two queries. The first query starts with a SELECT function and counting the number of restaurants in the restaurants table, in addition to a CONCAT function that will add the cuisine to the count. Then, in the WHERE clause, 2 conditions were added, using two subqueries. The first condition checks if the Restaurant ID from the restaurants table is in a list,

which is created by a select query of the Restaurant IDs from the ratings table, where the overall rating is 1. The second condition check if the Restaurant ID from the restaurants table is in a list, which is created by a select query of the Restaurant IDs from the restaurant cuisine table where the cuisine is Mexican. Moreover, these two conditions are separated by an “and” operator than ensures that the records have to meet both conditions. The same query was created for the Italian restaurants, though the cuisine is changed from “Mexican” to “Italian”. To be able to achieve the output below, the UNION ALL came to place as mentioned previously.

Query:

```
SELECT
    CONCAT(COUNT(res.Name), ' Mexican Restaurants') AS [Tot. nbr of restaurants (Overall rating 1)]
FROM [rst].[restaurants] res
WHERE res.Restaurant_ID in (
    SELECT DISTINCT
        Restaurant_ID
    FROM [rst].[ratings]
    WHERE Overall_Rating = 1
)
    and res.Restaurant_ID in (
    SELECT DISTINCT
        Restaurant_ID
    FROM [rst].[restaurant_cuisines]
    WHERE Cuisine = 'Mexican'
)
UNION ALL
SELECT
    CONCAT(COUNT(res.Name), ' Italian Restaurants') AS [Tot. nbr of restaurants (Overall rating 1)]
FROM [rst].[restaurants] res
WHERE res.Restaurant_ID in (
    SELECT DISTINCT
        Restaurant_ID
    FROM [rst].[ratings]
    WHERE Overall_Rating = 1
)
    and res.Restaurant_ID in (
    SELECT DISTINCT
        Restaurant_ID
    FROM [rst].[restaurant_cuisines]
    WHERE Cuisine = 'Italian'
)
```

Result:

	Tot. nbr of restaurants (Overall rating 1)
1	27 Mexican Restaurants
2	4 Italian Restaurants

There is a clear significant difference in the number of Mexican restaurants with overall rating of 1, compared to the number of Italian restaurants.

3. Calculate the average age of consumers who have given a 0 rating to the 'Service rating' column. (NB: round off the value if it is a decimal)

Based on the ask, the Age field is in the consumers table, and the service rating is stored in the ratings table. Thus, the need to INNER JOIN both tables on the Consumer ID field. The select function is used to get the SUM of the Age column and divided by the COUNT of the age column. Then, due to the fact that the data is stored as integers, and because the AVG could not be used as the result is a whole number, a CAST function was used to wrap the numbers and convert them to a decimal numbers. Finally, the ROUND function was used to round the division of the SUM and the

COUNT to two decimal points. In the condition clause, a filter for service rating was added to ensure it is a 0 rating as required.

Query:

```
SELECT
    ROUND(
        CAST(
            SUM(c.Age)
            AS FLOAT)
        /
        CAST(
            COUNT(c.Age)
            AS FLOAT)
        , 2) AS [Average Age Who Gave 0 Service Rating]
FROM [rst].[consumers] c
INNER JOIN [rst].[ratings] r
    ON c.Consumer_ID = r.Consumer_ID
WHERE r.Service_Rating = 0
```

Result:

	Average Age Who Gave 0 Service Rating
1	26.22

4. Write a query that returns the restaurants ranked by the youngest consumer. You should include the restaurant name and food rating that is given by that customer to the restaurant in your result. Sort the results based on food rating from high to low.

The query starts by selecting the restaurants names from the restaurant table, Consumer ID and Age from the consumers table, and the food rating from the ratings table. To be able to select all of these fields together, two table INNER JOINs are required, one with the ratings table on the Restaurant ID, and the other is with the consumers table, but using the Consumer ID from the ratings and the consumers' table. Moreover, in the WHERE clause, a condition is added, where the Age should be in a list of the following subquery; a select statement of the MIN age from the consumers table. And finally, an ORDER function is used, which orders the records by the food rating in a descending manner.

Query:

```
107 SELECT
108     res.Name,
109     con.Consumer_ID,
110     con.Age,
111     r.Food_Rating
112 FROM [rst].[restaurants] res
113 INNER JOIN [rst].[ratings] r
114     ON r.Restaurant_ID = res.Restaurant_ID
115 INNER JOIN [rst].[consumers] con
116     ON con.Consumer_ID = r.Consumer_ID
117 WHERE con.Age in (
118     SELECT
119         MIN(Age)
120     FROM [rst].[consumers]
121 )
122 ORDER BY r.Food_Rating Desc
```

Result:

	Name	Consumer_ID	Age	Food_Rating
1	Giovannis	U1040	18	2
2	Restaurant Bar Coty Y Pablo	U1040	18	2
3	El Cotorreo	U1040	18	1
4	Kiku Cuernavaca	U1040	18	1

5. Write a stored procedure for the query given as: Update the Service rating of all restaurants to '2' if they have parking available, either as 'yes' or 'public'

The query starts by CREATE PROCEDURE to be able to store it for later use. As there are no variables required, the wrapper BEGIN indicates the start of the procedure. The query is an UPDATE statement, on the ratings table, which will be applied on the Service Rating field, changing its value to two. All of the conditions are stored in the WHERE clause, where the Restaurant ID should be within a list defined by a subquery; selecting the restaurant IDs from the restaurant table where parking is available with a value “Yes” or “Public”. Then, the END marks the end of the query. To be able to execute the procedure, it should simply be run by using the EXEC function followed by the name of the procedure.

Query:

```
126 CREATE PROCEDURE Update_Service_Rating AS
127 BEGIN
128     UPDATE [rst].[ratings]
129     SET
130         Service_Rating = 2
131     WHERE
132         Restaurant_ID in (
133             SELECT
134                 Restaurant_ID
135             FROM [rst].[restaurants]
136             WHERE Parking in ('Yes', 'Public')
137         );
138 END
139
140 EXEC Update_Service_Rating;
```

Before query:

```
144 SELECT
145     res.Restaurant_ID,
146     res.Parking,
147     r.Service_Rating
148 FROM [rst].[restaurants] res
149 INNER JOIN [rst].[ratings] r
150     ON r.Restaurant_ID = res.Restaurant_ID
151 WHERE res.Parking in ('Yes', 'Public')
152     and r.Service_Rating <> 2
```

After query:

```
144 SELECT
145     res.Restaurant_ID,
146     res.Parking,
147     r.Service_Rating
148 FROM [rst].[restaurants] res
149 INNER JOIN [rst].[ratings] r
150     ON r.Restaurant_ID = res.Restaurant_ID
151 WHERE res.Parking in ('Yes', 'Public')
```

Before Result:

	Restaurant_ID	Parking	Service_Rating
1	132560	Public	0
2	132560	Public	0
3	132560	Public	0
4	132560	Public	1
5	132572	Yes	1
6	132572	Yes	1
7	132572	Yes	0
8	132572	Yes	0
9	132572	Yes	0
10	132572	Yes	0
11	132572	Yes	0
12	132572	Yes	1
13	132572	Yes	1
14	132572	Yes	0
15	132584	Yes	0
16	132584	Yes	1
17	132584	Yes	1
18	132584	Yes	0
19	132594	Public	0
20	132594	Public	1

After Result:

	Restaurant_ID	Parking	Service_Rating
1	132560	Public	2
2	132560	Public	2
3	132560	Public	2
4	132560	Public	2
5	132572	Yes	2
6	132572	Yes	2
7	132572	Yes	2
8	132572	Yes	2
9	132572	Yes	2
10	132572	Yes	2
11	132572	Yes	2
12	132572	Yes	2
13	132572	Yes	2
14	132572	Yes	2
15	132572	Yes	2
16	132572	Yes	2
17	132572	Yes	2
18	132572	Yes	2
19	132572	Yes	2
20	132584	Yes	2

6. You should also write four queries of your own and provide a brief explanation of the results which each query returns. You should make use of all of the following at least once:
    a. Nested queries-EXISTS

For this query, restaurant information from the restaurants table such as alcohol service, smoking, price, area, and parking were queried. Though in the WHERE clause, it uses the EXISTS operator where it will search in a subquery, the valid records selected, from the ratings table where the restaurant id from the ratings table is equal to the restaurant ids from the restaurants table, and food rating and service rating are both higher than one.

Query:

```
166 SELECT
167     Name,
168     Alcohol_Service,
169     Smoking_Allowed,
170     Price,
171     Area,
172     Parking
173 FROM [rst].[restaurants]
174 WHERE EXISTS (
175     SELECT *
176     FROM [rst].[ratings]
177     WHERE [rst].[ratings].Restaurant_ID = [rst].[restaurants].Restaurant_ID
178         and Food_Rating > 1
179         and Service_Rating > 1
180 )
```

Result (117 records):

	Name	Alcohol_Service	Smoking_Allowed	Price	Area	Parking
1	Puesto de Gorditas	None	Yes	Low	Open	Public
2	Cafe Ambar	None	No	Low	Closed	None
3	Church's	None	No	Low	Closed	None
4	Cafe Chaires	None	No	Low	Closed	Yes
5	McDonalds Centro	None	No	Low	Closed	None
6	Gorditas Doña Tota	None	No	Medium	Closed	Yes
7	Tacos De Barbacoa Enfrente Del Tec	None	No	Low	Open	Public
8	Hamburguesas La Perica	None	Yes	Low	Open	Public
9	Pollo Frito Buenos Aires	None	No	Low	Closed	Yes
10	Carnitas Mata	None	Yes	Medium	Closed	Yes
11	La Perica Hamburguesa	None	No	Medium	Closed	Yes
12	Palomo Tec	None	No	Low	Closed	None
13	Carnitas Mata Calle Emilio Portes Gil	None	No	Low	Closed	None
14	Tacos Correcaminos	None	No	Low	Closed	None
15	Little Pizza Emilio Portes Gil	None	No	Low	Closed	None
16	Tacos El Guero	None	No	Low	Closed	None
17	Gorditas Dona Tota	None	No	Medium	Closed	Public
18	Tortas Hawaii	None	No	Medium	Closed	Public
19	Gordas De Morales	Full Bar	Smoking Section	Medium	Closed	Public
20	Taqueria El Amigo	None	No	Low	Open	None

b. Nested queries-IN

This query is calculating the average overall rating, average food rating and the average service rating of a specific strata of consumers from the ratings table. The condition is using the IN operator where the Consumer ID is in a list created by a subquery, which is selecting the Consumer IDs from the consumers table, with three conditions in the WHERE statement: Country should be Mexico, Occupation should be Student and Budget should be Low. The “and” operator was used thus only records where all three conditions are met will be returned.

Query:

```
SELECT
    ROUND(
        CAST(SUM(Overall_Rating) AS FLOAT)
        /CAST(COUNT(Overall_Rating) AS FLOAT)
        , 2) AS [Average Overall Rating],
    ROUND(
        CAST(SUM(Food_Rating) AS FLOAT)
        /CAST(COUNT(Food_Rating) AS FLOAT)
        , 2) AS [Average Food Rating],
    ROUND(
        CAST(SUM(Service_Rating) AS FLOAT)
        /CAST(COUNT(Service_Rating) AS FLOAT)
        , 2) AS [Average Service Rating]
FROM [rst].[ratings]
WHERE Consumer_ID in (
    SELECT
        Consumer_ID
    FROM [rst].[consumers]
    WHERE Country = 'Mexico'
        and Occupation = 'Student'
        and Budget = 'Low'
)
```

Result:

	Average Overall Rating	Average Food Rating	Average Service Rating
1	1.14	1.12	1.51

c. System functions

This query starts by getting the full restaurant name, where it uses CONCAT to join the restaurant name with the word “serving” as well as the cuisine. Then, to be able to get the email address of the restaurant, the REPLACE function was used on the name to be able to remove any spaces, in addition to the LOWER function that will make all the letters as small characters, and a CONCAT function that will join the “info@”, the formatted name, the cuisine, and “.com” as the email address. Then the final column is getting the average overall rating and dividing it by 2 which is the maximum rating, so that the overall success rate percentage could be achieved. Two INNER JOINS were applied with the restaurants cuisine table on Restaurant ID and with ratings table on Restaurant ID to be able to query all mentioned fields. The GROUP BY function contains non-aggregated fields such as the restaurant name and cuisine. The results of this query are pushed into a virtual temporary table using the INTO function where the table name is “#Temp”.

Another query is created, pulling its data from “#Temp”, where it queries the restaurant name and email, uses the FORMAT function to display the success overall rate as percentages, and finally uses the CASE function which uses “WHEN something occurs, THEN output”, which was used in this case to specify if improvements are needed based on the success rate. The virtual temporary table is dropped using DROP TABLE at the end as when the next time the query is ran, there will not be a conflict where there is already a “#Temp” table in the database.

Query:

```
-- 6) C. System functions --
SELECT DISTINCT
    CONCAT(res.Name, ' serving ', resc.Cuisine) AS [Restaurant Cuisine],
    CONCAT('info@',REPLACE(LOWER(res.Name), ' ', ''), resc.Cuisine, '.com') AS [Restaurant Email],
    ROUND(
        CAST(SUM(ra.Overall_Rating) AS FLOAT)
        /
        CAST(COUNT(ra.Overall_Rating) AS FLOAT)
        , 2)
    /
    MAX(ra.Overall_Rating) AS [Success Overall Rating]
INTO #Temp
FROM [rst].[restaurants] res
LEFT JOIN [rst].[restaurant_cuisines] resc
    ON res.Restaurant_ID = resc.Restaurant_ID
LEFT JOIN [rst].[ratings] ra
    ON ra.Restaurant_ID = res.Restaurant_ID
GROUP BY res.Name, resc.Cuisine

SELECT
    [Restaurant Cuisine],
    [Restaurant Email],
    FORMAT([Success Overall Rating], 'P') AS [Success Overall Rating Percentage],
    CASE
        WHEN [Success Overall Rating] <= 0.35
            THEN 'Need immediate action'
        WHEN [Success Overall Rating] > 0.35 and [Success Overall Rating] <= 0.7
            THEN 'Improvement Needed'
        WHEN [Success Overall Rating] > 0.7
            THEN 'Great Rating'
    END AS [Rating Status]
FROM #Temp

DROP TABLE #Temp
```

Result (147 records):

	Restaurant Cuisine	Restaurant Email	Success Overall Rating Percentage	Rating Status
1	Abondance Restaurante Bar serving Bar	info@abondancerestaurantebarBar.com	50.00%	Improvement Needed
2	Arrachela Grill serving	info@arrachelagrill.com	50.00%	Improvement Needed
3	Cabana Huasteca serving Mexican	info@cabanahuastecaMexican.com	73.00%	Great Rating
4	Cafe Ambar serving	info@cafeambar.com	75.00%	Great Rating
5	Cafe Chaires serving Cafeteria	info@cafechairesCafeteria.com	50.00%	Improvement Needed
6	Cafe Punta Del Cielo serving Cafeteria	info@cafepuntadelcieloCafeteria.com	91.50%	Great Rating
7	Cafeteria Cenidet serving Cafeteria	info@cafeteriacenidetCafeteria.com	50.00%	Improvement Needed
8	Cafeteria Y Restaurant El Pacifico serving Cafeteria	info@cafeteriayrestaurantelpacificoCafeteria.c...	59.00%	Improvement Needed
9	Cafeteria Y Restaurant El Pacifico serving Contem...	info@cafeteriayrestaurantelpacificoContempor...	59.00%	Improvement Needed
10	Carl's Jr serving Burgers	info@carl'sjrBurgers.com	71.50%	Great Rating
11	Carnitas Mata Calle 16 de Septiembre serving	info@carnitasmatacalle16deseptiembre.com	25.00%	Need immediate action
12	Carnitas Mata Calle Emilio Portes Gil serving	info@carnitasmatacalleemilioportesgil.com	70.00%	Improvement Needed
13	Carnitas Mata serving Mexican	info@carnitasmataMexican.com	58.50%	Improvement Needed
14	Carreton De Flautas Y Migadas serving Mexican	info@carretondeflautasymigadasMexican.com	37.50%	Improvement Needed
15	Cenaduria El Rincón De Tlaquepaque serving Me...	info@cenaduriaelrincóndetlaquepaqueMexica...	40.00%	Improvement Needed
16	Chaires serving Bakery	info@chairesBakery.com	70.00%	Improvement Needed
17	Chaires serving Cafeteria	info@chairesCafeteria.com	70.00%	Improvement Needed
18	Chilis Cuernavaca serving	info@chiliscuernavaca.com	50.00%	Improvement Needed
19	Church's serving	info@church's.com	62.50%	Improvement Needed
20	Crudalia serving Bar	info@crudaliaBar.com	62.00%	Improvement Needed

d. Use of GROUP BY, HAVING and ORDER BY clauses

The query starts by specifying that it is selecting only the TOP twenty records from the outputted records. It uses the COUNT function to count the consumer IDs as the number of reviews and displays the restaurant names. The INNER JOIN allows the field from restaurant table to be displayed, and the GROUP BY function holds the non-aggregated field, in addition to the fields that will be used in the HAVING clause and not present within the SELECT statement. The HAVING clause will re-route the query's output to records where the overall rating is not zero. And finally, the ORDER BY function orders the counts of the consumer IDs in a descending manner.

Query:

```
242 SELECT TOP 20
243     COUNT(ra.Consumer_ID) AS [Number of Reviews],
244     res.Name AS [Restaurant Name]
245 FROM [rst].[ratings] ra
246 INNER JOIN [rst].[restaurants] res
247     ON ra.Restaurant_ID = res.Restaurant_ID
248 GROUP BY res.Name, ra.Overall_Rating
249 HAVING ra.Overall_Rating <> 0
250 ORDER BY COUNT(ra.Consumer_ID) DESC;
```

Result:

	Number of Reviews	Restaurant Name
1	18	Tortas Locas Hipocampo
2	15	Puesto De Tacos
3	14	La Cantina Restaurante
4	12	Tortas Locas Hipocampo
5	12	Cafeteria Y Restaurant El Pacifico
6	11	Restaurante Marisco Sam
7	11	Gorditas Doa Gloria
8	11	Puesto De Tacos
9	11	El Rincon De San Francisco
10	10	La Posada Del Virrey
11	10	Mariscos El Pescador
12	10	Restaurant La Chalita
13	10	Restaurante El Cielo Potosino
14	10	Vips
15	9	Restaurant La Chalita
16	9	Cafeteria Y Restaurant El Pacifico
17	9	Cafe Chaires
18	9	Restaurant Oriental Express
19	9	La Virreina
20	8	Restaurant Las Mañanitas

In conclusion, the database created based on the four csv spreadsheets have been imported, and the tables altered in a way that it reflects the primary keys and foreign keys assigned. The entity relational diagram follows a snowflake schema with the ratings table as the factual table and the rest as dimensions.