CSE 581
Introduction to Database Management Systems

PROJECT 2

Akhilesh Santosh Jadhav
651126694

University Medical Centre Database

Abstract:

Designing, building, and testing a database management system that will be utilized to store the data for University Medical Center is the goal of this project. As a result, this database will take into account a number of base tables, such as those for medical staff, as well as linking tables, such as those for patient appointments, doctor appointments, nurse appointments, schedules for operating rooms, schedules for inpatient rooms, and schedules for equipment reservations. It will also take into account the typical activities that occur in a hospital setting.

Table of Contents:

Introduction:

The three stages of this project are design, implementation, and testing. The design consists of the relevant tables as well as the numerous Entity Relationships. We'll talk about various business logics and transactions to consider as well as how to include them into the design. During implementation, codes that create tables, views, triggers, procedures, etc. are run. The last stage in the testing phase is to run several scenarios to evaluate the complexity of the database.

Design Consideration:

- **patients**: This table would store information about patients, including their name, address, contact information, date of birth, and insurance information.
- **medicalHistory**: This table would store information about a patient's medical history, including any conditions they have been diagnosed with, the date of the diagnosis, details of the diagnosis, and the treatment received.
- **physicians**: This table would store information about physicians, including their name, specialty, and license information.
- **physicianContactInfo**: This table would store contact information for physicians, including their address and email.
- **nurseJobTitles**: This table would store a list of job titles for nurses.
- **nurses**: This table would store information about nurses, including their name, department, job title, and hire date.
- **notes**: This table would store notes that physicians and nurses have taken about patients, including the text of the note, the date it was created, and the date it was last updated.
- **appointments**: This table would store information about appointments that patients have scheduled with physicians, including the date and time of the appointment and the reason for the appointment.
- **admissionTypes**: This table would store a list of admission types, such as emergency admission, elective admission, and so on.
- **admissions**: This table would store information about a patient's admission to the hospital, including the date of admission, the date of discharge, and details of the diagnosis.
- **procedures**: This table would store information about medical procedures performed on patients, including the date of the procedure and details of the procedure.
- **medications**: This table would store information about medications prescribed to patients, including the name of the medication, the dosage, and the frequency.
- **labTests**: This table would store information about lab tests performed on patients, including the name of the test, the date it was performed, and the results.
- **billing**: This table would store information about billing for medical services provided to patients, including the service charge, insurance information, and payment information.
- **billingServices**: This table would store information about the specific services for which a patient was billed, including the name of the service and the cost.
- **rooms**: This table would store information about rooms in the hospital, including the room number and room type.
- **equipment**: This table would store information about medical equipment used in the hospital, including the name and description of the equipment.
- **roomReservations**: This table would store information about reservations for hospital rooms, including the start and end dates of the reservation and the patient, physician, and nurse associated with the reservation.
- **surgeryTypes**: This table would store a list of surgery types, such as elective surgery, emergency surgery, and so on.
- **surgeryRooms**: This table would store information about rooms in the hospital used for surgical procedures, including the room number and the type of surgery performed in the room.

- **surgeryRoomSchedules**: This table would store information about the schedule for surgical procedures in a specific surgery room, including the start and end times of each procedure and the physician and nurse assigned to each procedure.
- **inpatientRooms**: This table would store information about rooms in the hospital used for inpatient care, including the room number and bed number.
- **inpatientRoomSchedules**: This table would store information about the schedule for patients in a specific inpatient room, including the start and end times of each patient's stay and the nurse assigned to each patient.
- **icd10Codes**: This table would store information about International Classification of Diseases (ICD-10) codes used to classify medical diagnoses and procedures.
- **icd10Diagnoses:** This table has a primary key icd10DiagnosisId and two foreign keys admissionId and icd10CodeId, which reference the admissions and icd10Codes tables respectively. This table is used to record the diagnoses assigned to a patient during an admission, along with the corresponding ICD-10 codes.
- **icd10Procedures:** This table has a primary key icd10ProcedureId and two foreign keys procedureId and icd10CodeId, which reference the procedures and icd10Codes tables respectively. This table is used to record the procedures performed on a patient during a hospital stay, along with the corresponding ICD-10 codes

Potential Integrity and Security Issues:

There are potential security issues that need to be addressed.

Injection attacks: Before utilizing user input in SQL queries, the program should sanitize it. One of the most frequent kinds of assaults on databases is SQL injection. Using parameterized queries or prepared statements is the easiest method to avoid them.

Weak passwords: Ensure that users employ secure passwords that are challenging to decipher. The minimum length for a password is eight characters, and it should contain a mix of capital and lowercase letters, digits, and special characters.

Unapproved access: Don't let anyone but authorized staff access the database. Users should only have access to the information they require to carry out their duties. To impose access restrictions on sensitive data, use role-based access control.

Encrypt sensitive data, such as health information about patients, to prevent unwanted access. It is advisable to apply encryption both at rest and in transit.

Backup and recovery: In case of data loss or corruption, have a backup and recovery plan in place. Backup the database often, and test the recovery procedure to make sure it works.

Data verification: Verify user input to make sure it follows the desired format. This can enhance data integrity and avoid data corruption.

Data redundancy: To reduce data redundancy, make sure the database is normalized. Inconsistencies and problems with data integrity can be caused by redundant data.

Implement an audit trail so you can keep track of all database modifications. This can aid in locating security holes and problems with data integrity.

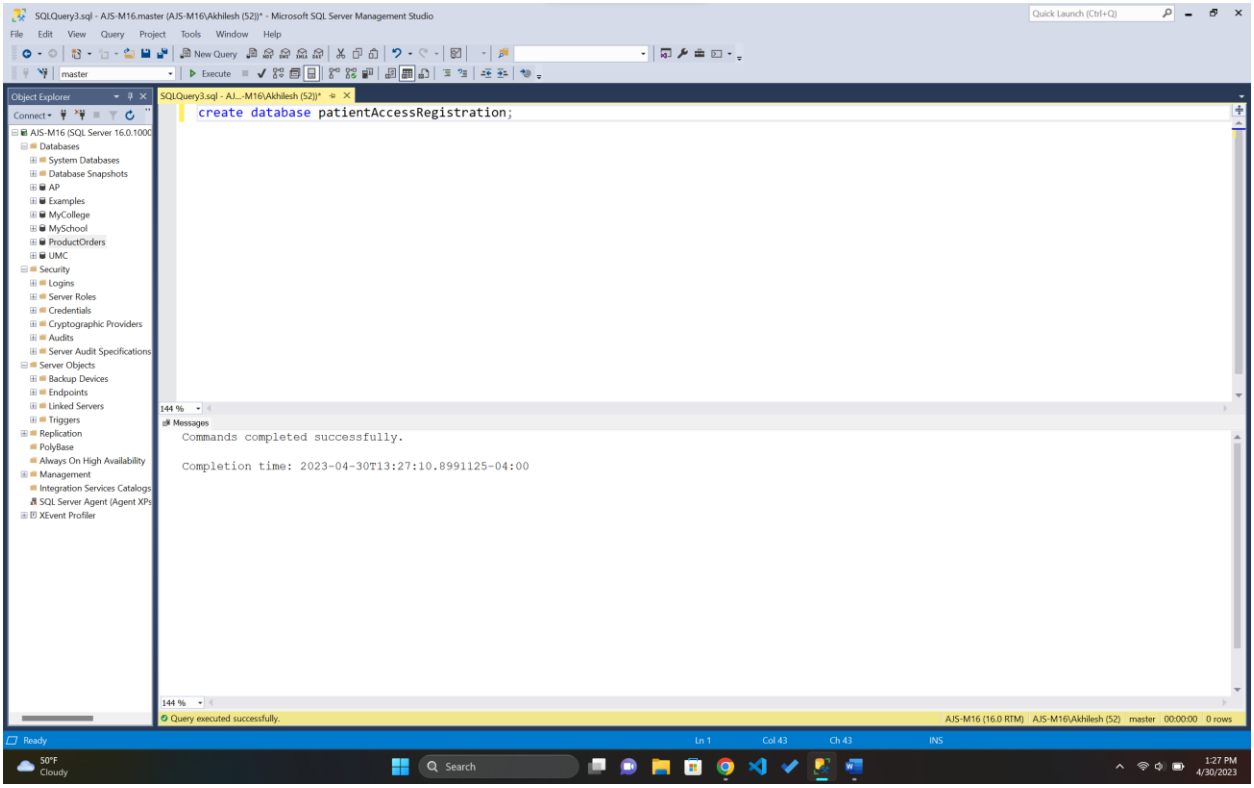
Entity Relationship:

Implementation and Testing:

1. Creating Database

Source Code: -

```
create database patientAccessRegistration;
```



1. Creating Database

2. Creating tables

Source Code: -

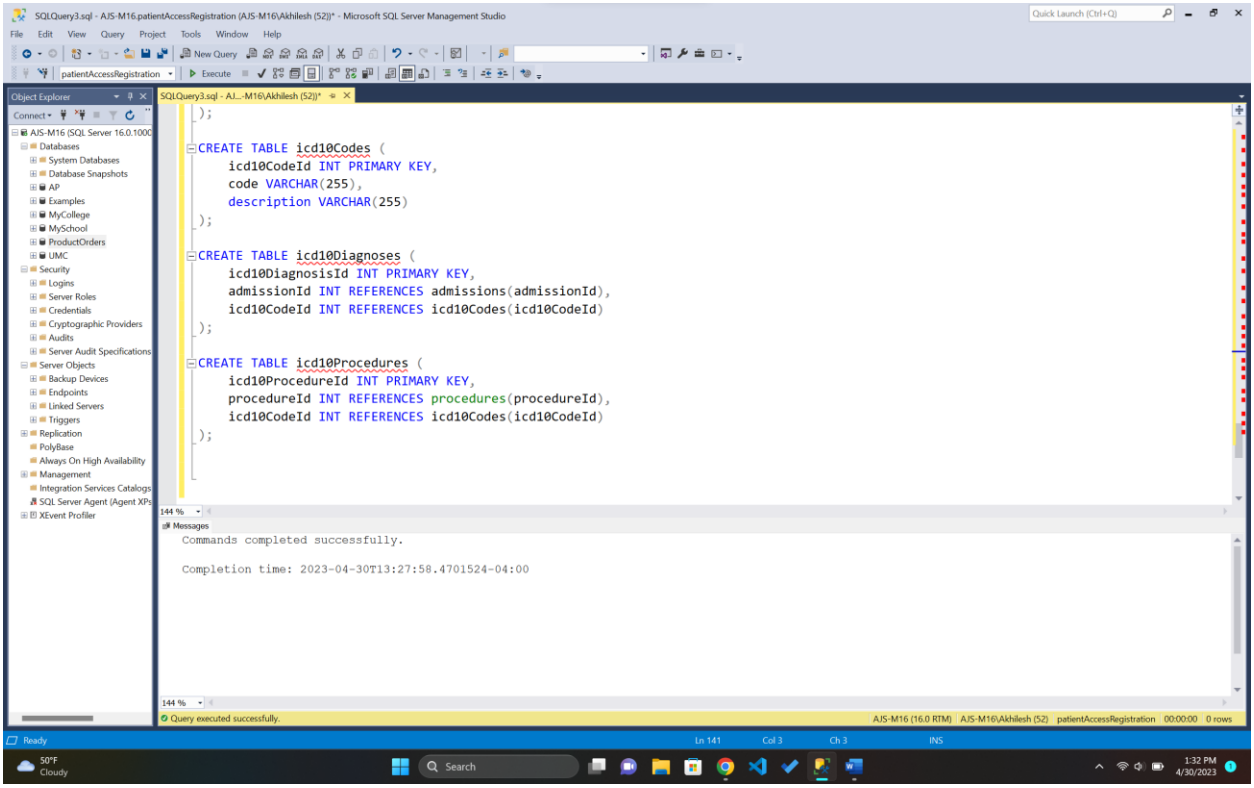
```
CREATE TABLE patients (
  patientId INT PRIMARY KEY,
  name VARCHAR(255),
  address VARCHAR(255),
  contactInfo VARCHAR(255),
  dateOfBirth DATE,
  insuranceInfo VARCHAR(255)
);

CREATE TABLE medicalHistory (
  medicalHistoryId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  condition VARCHAR(255),
  diagnosisDate DATE,
  diagnosisDetails VARCHAR(255),
  treatment VARCHAR(255)
);
```

```
CREATE TABLE physicians (
  physicianId INT PRIMARY KEY,
  name VARCHAR(255),
  specialty VARCHAR(255),
  licenseInfo VARCHAR(255)
);

CREATE TABLE physicianContactInfo (
  physicianContactInfoId INT PRIMARY KEY,
  physicianId INT REFERENCES physicians(physicianId),
  contactInfo VARCHAR(255),
  address VARCHAR(255),
  email VARCHAR(255)
);

CREATE TABLE nurseJobTitles (
  jobTitleId INT PRIMARY KEY,
  title VARCHAR(255)
);

CREATE TABLE nurses (
  nurseId INT PRIMARY KEY,
  name VARCHAR(255),
  department VARCHAR(255),
  jobTitleId INT REFERENCES nurseJobTitles(jobTitleId),
  hireDate DATE
);

CREATE TABLE notes (
  noteId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  nurseId INT REFERENCES nurses(nurseId),
  noteText VARCHAR(255),
  createdAt DATETIME,
  updatedAt DATETIME
);

CREATE TABLE appointments (
  appointmentId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  appointmentDatetime DATETIME,
  reason VARCHAR(255)
);

CREATE TABLE admissionTypes (
  admissionTypeId INT PRIMARY KEY,
  type VARCHAR(255)
);

CREATE TABLE admissions (
  admissionId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  nurseId INT REFERENCES nurses(nurseId),
  admissionTypeId INT REFERENCES admissionTypes(admissionTypeId),
  admissionDate DATE,
```

```
  dischargeDate DATE,
  diagnosisDetails VARCHAR(255)
);

CREATE TABLE procedures (
  procedureId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  procedureDate DATE,
  procedureDetails VARCHAR(255)
);

CREATE TABLE medications (
  medicationId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  medicationName VARCHAR(255),
  dosage VARCHAR(255),
  frequency VARCHAR(255),
  startDate DATE,
  endDate DATE
);

CREATE TABLE labTests (
  labTestId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  testName VARCHAR(255),
  testDate DATE,
  testResults VARCHAR(255)
);

CREATE TABLE billing (
  billingId INT PRIMARY KEY,
  patientId INT REFERENCES patients(patientId),
  physicianId INT REFERENCES physicians(physicianId),
  nurseId INT REFERENCES nurses(nurseId),
  serviceCharge DECIMAL,
  insuranceInfo VARCHAR(255),
  paymentInfo VARCHAR(255)
);

CREATE TABLE billingServices (
  billingServiceId INT PRIMARY KEY,
  billingId INT REFERENCES billing(billingId),
  serviceName VARCHAR(255),
  serviceCharge DECIMAL
);

CREATE TABLE rooms (
  roomId INT PRIMARY KEY,
  roomNumber VARCHAR(255),
  roomType VARCHAR(255)
);

CREATE TABLE equipment (
  equipmentId INT PRIMARY KEY,
  equipmentName VARCHAR(255),
```

```
  equipmentDescription VARCHAR(255)
);

CREATE TABLE roomReservations (
    reservationId INT PRIMARY KEY,
    roomId INT REFERENCES rooms(roomId),
    patientId INT REFERENCES patients(patientId),
    physicianId INT REFERENCES physicians(physicianId),
    nurseId INT REFERENCES nurses(nurseId),
    startDate DATETIME,
    endDate DATETIME
);

CREATE TABLE surgeryTypes (
    surgeryTypeId INT PRIMARY KEY,
    type VARCHAR(255)
);

CREATE TABLE surgeryRooms (
    surgeryRoomId INT PRIMARY KEY,
    roomId INT REFERENCES rooms(roomId),
    surgeryTypeId INT REFERENCES surgeryTypes(surgeryTypeId)
);

CREATE TABLE surgeryRoomSchedules (
    scheduleId INT PRIMARY KEY,
    surgeryRoomId INT REFERENCES surgeryRooms(surgeryRoomId),
    physicianId INT REFERENCES physicians(physicianId),
    nurseId INT REFERENCES nurses(nurseId),
    startDatetime DATETIME,
    endDatetime DATETIME
);

CREATE TABLE inpatientRooms (
    inpatientRoomId INT PRIMARY KEY,
    roomId INT REFERENCES rooms(roomId),
    bedNumber INT
);

CREATE TABLE inpatientRoomSchedules (
    scheduleId INT PRIMARY KEY,
    inpatientRoomId INT REFERENCES inpatientRooms(inpatientRoomId),
    patientId INT REFERENCES patients(patientId),
    nurseId INT REFERENCES nurses(nurseId),
    startDatetime DATETIME,
    endDatetime DATETIME
);

CREATE TABLE icd10Codes (
    icd10CodeId INT PRIMARY KEY,
    code VARCHAR(255),
    description VARCHAR(255)
);

CREATE TABLE icd10Diagnoses (
    icd10DiagnosisId INT PRIMARY KEY,
    admissionId INT REFERENCES admissions(admissionId),
    icd10CodeId INT REFERENCES icd10Codes(icd10CodeId)
```

```
);

CREATE TABLE icd10Procedures (
    icd10ProcedureId INT PRIMARY KEY,
    procedureId INT REFERENCES procedures(procedureId),
    icd10CodeId INT REFERENCES icd10Codes(icd10CodeId)
);
```



2. Creating Tables

3. Populating the database by inserting values

Source Code: -

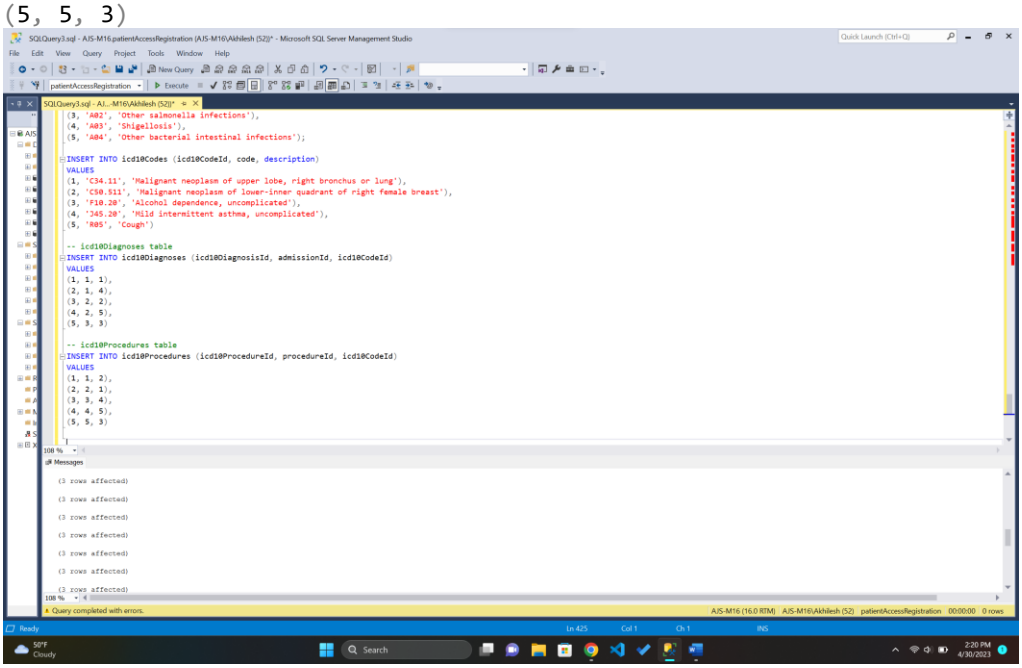
```
INSERT INTO patients (patientId, name, address, contactInfo, dateOfBirth, insuranceInfo)
VALUES
(1, 'Linda Davis', '123 Maple Street, Anytown USA', '555-1234', '1990-01-15', 'XYZ
Insurance'),
(2, 'Alex Johnson', '456 Elm Street, Anytown USA', '555-5678', '1985-07-01', 'ABC
Insurance'),
(3, 'Emily Parker', '789 Oak Street, Anytown USA', '555-9876', '1977-02-10', 'LMN
Insurance'),
(4, 'Ethan Martin', '234 Pine Street, Anytown USA', '555-4321', '1995-11-03', 'QRS
Insurance'),
(5, 'Isabella Wright', '567 Oak Street, Anytown USA', '555-8765', '1980-09-18', 'TUV
Insurance');

INSERT INTO medicalHistory (medicalHistoryId, patientId, condition, diagnosisDate,
diagnosisDetails, treatment)
VALUES
(1, 1, 'High Blood Pressure', '2015-01-01', 'BP reading: 140/90', 'Prescription
medication'),
(2, 2, 'Type 1 Diabetes', '2010-07-15', 'High blood sugar levels', 'Insulin injections'),
(3, 3, 'Asthma', '2003-04-05', 'Difficulty breathing', 'Inhaler'),
(4, 4, 'Migraine Headaches', '2018-09-01', 'Throbbing pain, sensitivity to light and
sound', 'Prescription medication'),
(5, 5, 'Depression', '2005-12-10', 'Persistent sadness, loss of interest',
'Psychotherapy');


INSERT INTO physicians (physicianId, name, specialty, licenseInfo)
VALUES
(1, 'Dr. Elizabeth Smith', 'Cardiology', '1234'),
(2, 'Dr. Matthew Johnson', 'Endocrinology', '5678'),
(3, 'Dr. Sarah Lee', 'Neurology', '9012'),
(4, 'Dr. David Brown', 'Psychiatry', '3456'),
(5, 'Dr. Jennifer Davis', 'Dermatology', '7890');


INSERT INTO physicianContactInfo (physicianContactInfoId, physicianId, contactInfo,
address, email)
VALUES
(1, 1, '555-1111', '456 Main Street, Anytown USA', 'elizabeth.smith@physicians.com'),
(2, 2, '555-2222', '789 Elm Street, Anytown USA', 'matthew.johnson@physicians.com'),
(3, 3, '555-3333', '123 Oak Street, Anytown USA', 'sarah.lee@physicians.com'),
(4, 4, '555-4444', '234 Pine Street, Anytown USA', 'david.brown@physicians.com'),
(5, 5, '555-5555', '567 Maple Street, Anytown USA', 'jennifer.davis@physicians.com');

INSERT INTO nurseJobTitles (jobTitleId, title)
VALUES
(1, 'Registered Nurse'),
(2, 'Licensed Practical Nurse'),
(3, 'Certified Nursing Assistant'),
(4, 'Nurse Practitioner'),
(5, 'Clinical Nurse Specialist');

INSERT INTO nurses (nurseId, name, department, jobTitleId, hireDate)
VALUES
(1, 'Maria Perez', 'Emergency Room', 1, '2015-01-01'),
```

```
(2, 'John Smith', 'Intensive Care Unit', 2, '2010-07-15'),
(3, 'Emily Chen', 'Pediatrics', 3, '2003-04-05'),
(4, 'David Lee', 'Operating Room', 4, '2018-09-01'),
(5, 'Sarah Johnson', 'Cardiology', 5, '2005-12-10');


INSERT INTO notes (noteId, patientId, physicianId, nurseId, noteText, createdAt,
updatedAt)
VALUES
(1, 1, 1, 1, 'Patient is experiencing chest pain', '2023-04-30 08:00:00', '2023-04-30
08:30:00'),
(2, 2, 2, 2, 'Patient is due for insulin injection', '2023-04-30 10:00:00', '2023-04-30
10:30:00'),
(3, 3, 3, 3, 'Patient is experiencing shortness of breath', '2023-04-30 12:00:00', '2023-
04-30 12:30:00'),
(4, 4, 4, 4, 'Patient is anxious and experiencing panic attacks', '2023-04-30 14:00:00',
'2023-04-30 14:30:00'),
(5, 5, 5, 5, 'Patient has a rash on their arm', '2023-04-30 16:00:00', '2023-04-30
16:30:00');


INSERT INTO appointments (appointmentId, patientId, physicianId, appointmentDatetime,
reason)
VALUES
(1, 1, 1, '2023-05-01 10:00:00', 'Follow-up for high blood pressure'),
(2, 2, 2, '2023-05-02 14:00:00', 'Diabetes management'),
(3, 3, 3, '2023-05-03 09:00:00', 'Asthma check-up'),
(4, 4, 4, '2023-05-04 11:00:00', 'Anxiety and depression therapy'),
(5, 5, 5, '2023-05-05 13:00:00', 'Skin condition evaluation');


-- admissionTypes table
INSERT INTO admissionTypes (admissionTypeId, type)
VALUES
  (1, 'Emergency'),
  (2, 'Urgent'),
  (3, 'Elective'),
  (4, 'Maternity'),
  (5, 'Trauma');

-- admissions table
INSERT INTO admissions (admissionId, patientId, physicianId, nurseId, admissionTypeId,
admissionDate, dischargeDate, diagnosisDetails)
VALUES
  (10, 1, 1, 1, 1, '2022-03-15', '2022-03-20', 'Pneumonia'),
  (12, 2, 2, 2, 2, '2022-04-01', '2022-04-05', 'Appendicitis'),
  (13, 3, 3, 3, 3, '2022-05-10', '2022-05-12', 'Hernia'),
  (14, 4, 4, 4, 4, '2022-06-20', '2022-06-22', 'Delivery'),
  (15, 5, 5, 5, 5, '2022-07-15', '2022-07-30', 'Fractured leg');

-- procedures table
INSERT INTO procedures (procedureId, patientId, physicianId, procedureDate,
procedureDetails)
VALUES
  (1, 1, 1, '2022-03-22', 'Appendectomy'),
  (2, 2, 2, '2022-04-03', 'Hernia repair'),
  (3, 3, 3, '2022-05-12', 'Gallbladder removal'),
  (4, 4, 4, '2022-06-21', 'C-section'),
```

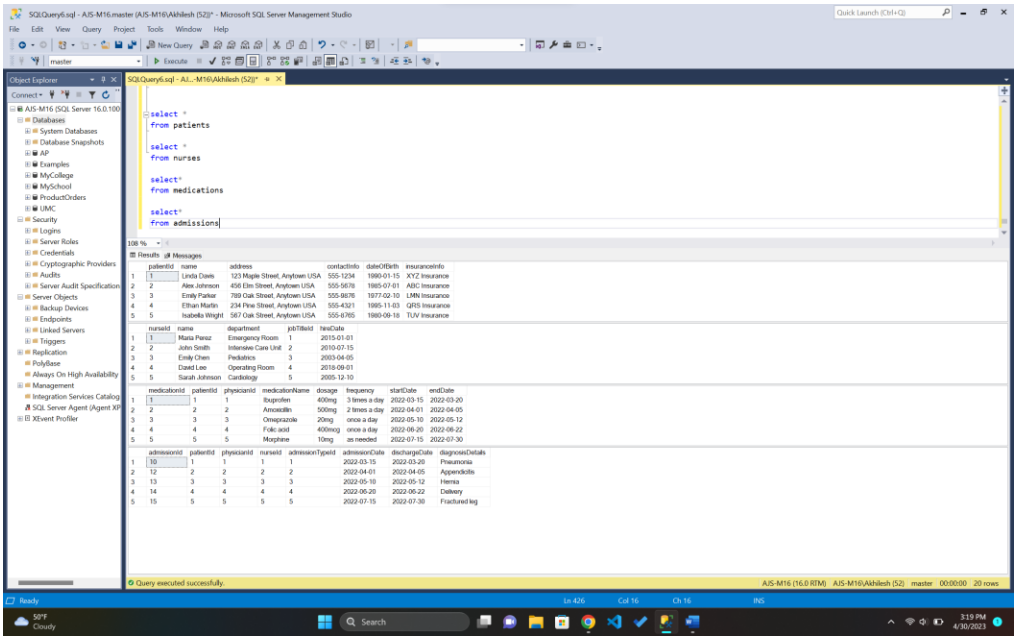
```
  (5, 5, 5, '2022-07-20', 'Arm surgery');

-- medications table
INSERT INTO medications (medicationId, patientId, physicianId, medicationName, dosage,
frequency, startDate, endDate)
VALUES
  (1, 1, 1, 'Ibuprofen', '400mg', '3 times a day', '2022-03-15', '2022-03-20'),
  (2, 2, 2, 'Amoxicillin', '500mg', '2 times a day', '2022-04-01', '2022-04-05'),
  (3, 3, 3, 'Omeprazole', '20mg', 'once a day', '2022-05-10', '2022-05-12'),
  (4, 4, 4, 'Folic acid', '400mcg', 'once a day', '2022-06-20', '2022-06-22'),
  (5, 5, 5, 'Morphine', '10mg', 'as needed', '2022-07-15', '2022-07-30');

INSERT INTO labTests (labTestId, patientId, physicianId, testName, testDate, testResults)
VALUES
  (1, 2, 3, 'Blood Test', '2023-04-15', 'Normal'),
  (2, 1, 4, 'Urine Test', '2023-04-17', 'Abnormal'),
  (3, 3, 2, 'X-ray', '2023-04-18', 'Fracture'),
  (4, 4, 1, 'MRI', '2023-04-20', 'Normal'),
  (5, 2, 4, 'Blood Test', '2023-04-25', 'Abnormal');

INSERT INTO billing (billingId, patientId, physicianId, nurseId, serviceCharge,
insuranceInfo, paymentInfo)
VALUES
  (1, 2, 3, 4, 500.00, 'XYZ Insurance', 'Paid'),
  (2, 1, 4, 2, 2500.00, 'ABC Insurance', 'Unpaid'),
  (3, 3, 2, 1, 800.00, 'PQR Insurance', 'Paid'),
  (4, 4, 1, 3, 1200.00, 'XYZ Insurance', 'Paid'),
  (5, 2, 4, 1, 400.00, 'ABC Insurance', 'Unpaid');

INSERT INTO billingServices (billingServiceId, billingId, serviceName, serviceCharge)
VALUES
  (1, 1, 'Consultation', 100.00),
  (2, 1, 'Lab Test', 200.00),
  (3, 2, 'Surgery', 2000.00),
  (4, 3, 'Medication', 400.00),
  (5, 4, 'MRI', 1000.00);

INSERT INTO rooms (roomId, roomNumber, roomType)
VALUES
  (1, '101', 'Single'),
  (2, '102', 'Double'),
  (3, '201', 'Single'),
  (4, '202', 'Double'),
  (5, '301', 'Single');

-- Equipment
INSERT INTO equipment (equipmentId, equipmentName, equipmentDescription)
VALUES
(1, 'X-ray machine', 'A machine used for medical imaging'),
(2, 'CT scanner', 'A machine used for diagnostic imaging'),
(3, 'MRI machine', 'A machine used for diagnostic imaging');

-- Room Reservations
INSERT INTO roomReservations (reservationId, roomId, patientId, physicianId, nurseId,
startDate, endDate)
VALUES
(1, 1, 1, 1, 1, '2023-05-01 10:00:00', '2023-05-01 12:00:00'),
(2, 2, 2, 2, 2, '2023-05-02 13:00:00', '2023-05-02 15:00:00'),
```

```
(3, 3, 3, 3, 3, '2023-05-03 16:00:00', '2023-05-03 18:00:00');

-- Surgery Types
INSERT INTO surgeryTypes (surgeryTypeId, type)
VALUES
(1, 'Open surgery'),
(2, 'Minimally invasive surgery'),
(3, 'Robotic surgery');

-- Surgery Rooms
INSERT INTO surgeryRooms (surgeryRoomId, roomId, surgeryTypeId)
VALUES
(1, 1, 1),
(2, 2, 2),
(3, 3, 3);

-- Surgery Room Schedules
INSERT INTO surgeryRoomSchedules (scheduleId, surgeryRoomId, physicianId, nurseId,
startDatetime, endDatetime)
VALUES
(1, 1, 1, 1, '2023-05-01 10:00:00', '2023-05-01 12:00:00'),
(2, 2, 2, 2, '2023-05-02 13:00:00', '2023-05-02 15:00:00'),
(3, 3, 3, 3, '2023-05-03 16:00:00', '2023-05-03 18:00:00');

-- Inpatient Rooms
INSERT INTO inpatientRooms (inpatientRoomId, roomId, bedNumber)
VALUES
(1, 1, 1),
(2, 2, 2),
(3, 3, 3);

-- Inpatient Room Schedules
INSERT INTO inpatientRoomSchedules (scheduleId, inpatientRoomId, patientId, nurseId,
startDatetime, endDatetime)
VALUES
(1, 1, 1, 1, '2023-05-01 10:00:00', '2023-05-03 12:00:00'),
(2, 2, 2, 2, '2023-05-02 13:00:00', '2023-05-04 15:00:00'),
(3, 3, 3, 3, '2023-05-03 16:00:00', '2023-05-05 18:00:00');


INSERT INTO icd10Codes (icd10CodeId, code, description)
VALUES
(1, 'C34.11', 'Malignant neoplasm of upper lobe, right bronchus or lung'),
(2, 'C50.511', 'Malignant neoplasm of lower-inner quadrant of right female breast'),
(3, 'F10.20', 'Alcohol dependence, uncomplicated'),
(4, 'J45.20', 'Mild intermittent asthma, uncomplicated'),
(5, 'R05', 'Cough')

-- icd10Diagnoses table
INSERT INTO icd10Diagnoses (icd10DiagnosisId, admissionId, icd10CodeId)
VALUES
(1, 10, 1),
(2, 10, 4),
(3, 12, 2),
(4, 12, 5),
(5, 13, 3)

-- icd10Procedures table
```

```
INSERT INTO icd10Procedures (icd10ProcedureId, procedureId, icd10CodeId)
VALUES
(1, 1, 2),
(2, 2, 1),
(3, 3, 4),
(4, 4, 5),
(5, 5, 3)
```



3. Queries to populate the database



4. Queries to show the inserted data

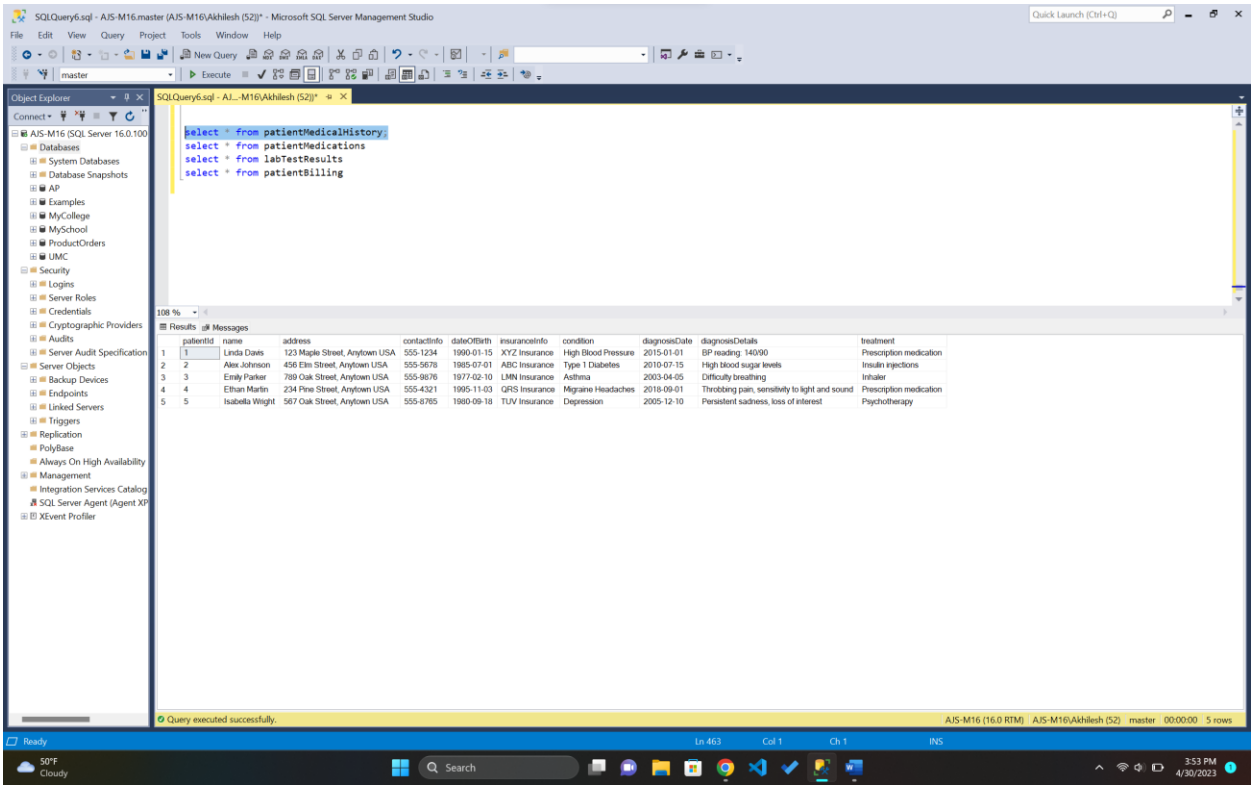
Views:

1. First view:

This code creates a database view called "patientMedicalHistory" that combines patient information with their medical history. It selects specific columns from the "patients" and "medicalHistory" tables using a join statement based on patientId. The selected columns include patientId, name, address, contactInfo, dateOfBirth, insuranceInfo, condition, diagnosisDate, diagnosisDetails, and treatment. The view can be accessed by executing the "select * from patientMedicalHistory" statement, which will display all columns and rows in the view. This code can be useful for healthcare professionals who need to quickly access and review patient information and medical history in one place.

Source Code: -

```
CREATE VIEW patientMedicalHistory AS
SELECT p.patientId, p.name, p.address, p.contactInfo, p.dateOfBirth, p.insuranceInfo,
       mh.condition, mh.diagnosisDate, mh.diagnosisDetails, mh.treatment
FROM patients p
JOIN medicalHistory mh ON p.patientId = mh.patientId;

select * from patientMedicalHistory;
```



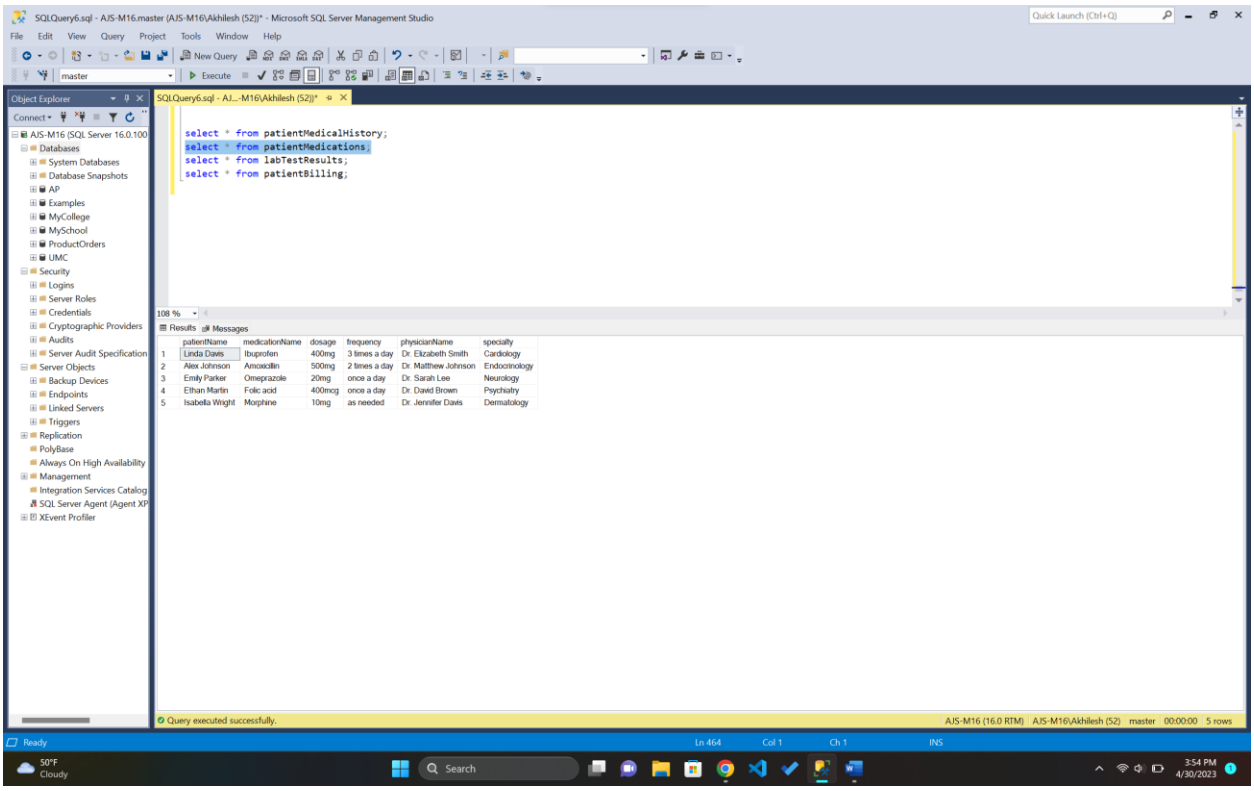
5. Patient Medical History view

2. Second view:

This code creates a view called `patientMedications` that contains information about patients, their medications, and prescribing physicians. The view combines data from the `patients`, `medications`, and `physicians` tables. Specifically, the view includes the patient's name, medication name, dosage, and frequency of medication, as well as the prescribing physician's name and specialty. Once the view is created, the `SELECT` statement is used to display all of the information in the view.

Source Code: -

```
CREATE VIEW patientMedications AS
SELECT p.name AS patientName, m.medicationName, m.dosage, m.frequency, md.name AS
physicianName, md.specialty
FROM patients p
JOIN medications m ON p.patientId = m.patientId
JOIN physicians md ON m.physicianId = md.physicianId;

select * from patientMedications
```



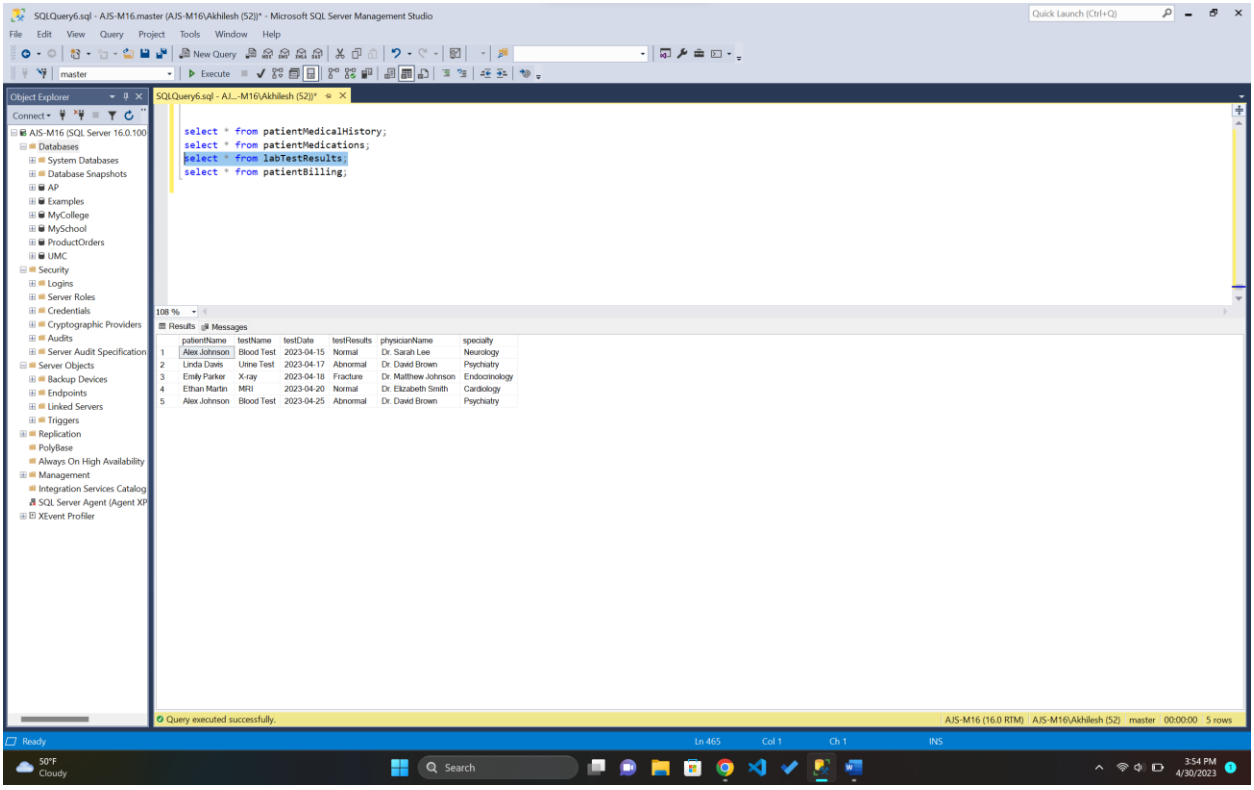
6. Patient's Medication view

3. Third view:

The below code creates a view named "labTestResults" that retrieves information from three tables: "patients," "labTests," and "physicians." The SELECT statement retrieves the name of the patient, the name of the lab test, the date the test was performed, the results of the test, the name of the physician who ordered the test, and the physician's specialty. The JOIN clause is used to combine data from the three tables based on the patient ID and physician ID. The view can then be queried using "select * from labTestResults" to retrieve the lab test results of patients along with their physicians' names and specialties.

Source Code: -

```
CREATE VIEW labTestResults AS
SELECT p.name AS patientName, l.testName, l.testDate, l.testResults, md.name AS
physicianName, md.specialty
FROM patients p
JOIN labTests l ON p.patientId = l.patientId
JOIN physicians md ON l.physicianId = md.physicianId;

select * from labTestResults
```



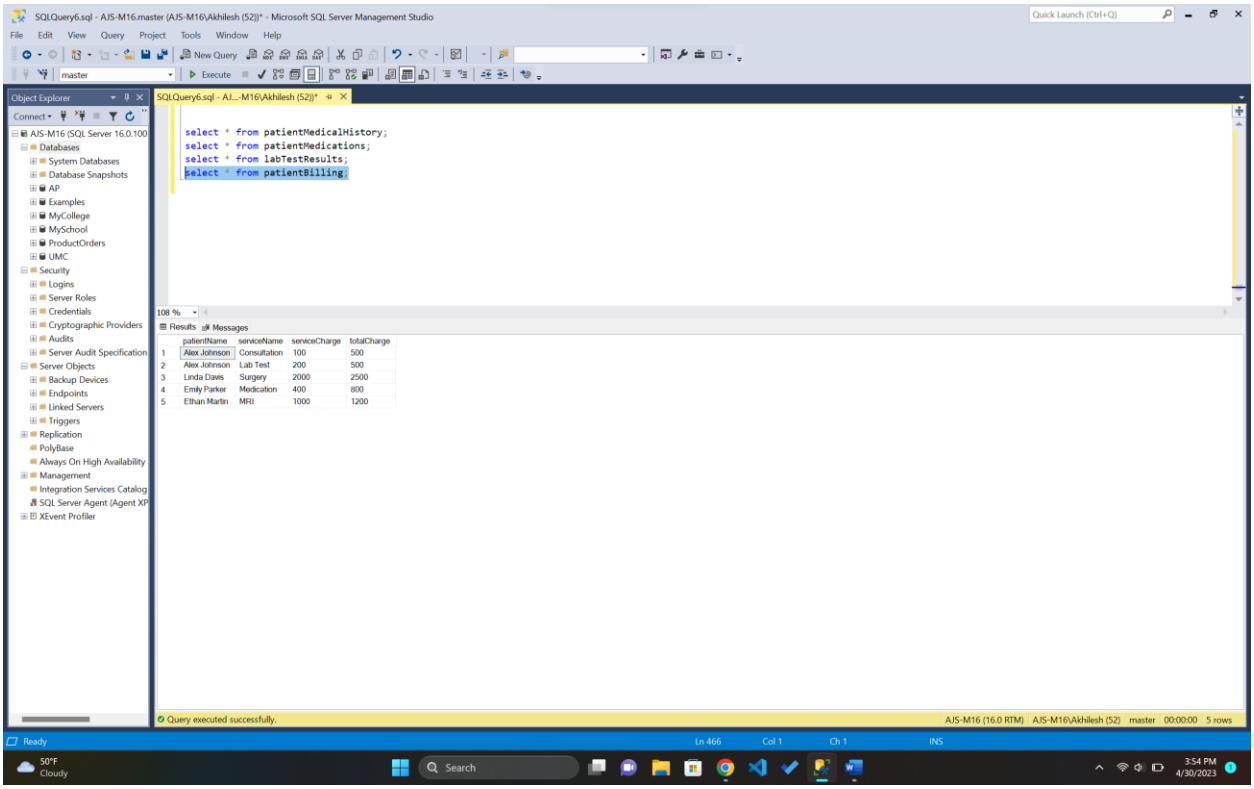
7. Lab's test results view

4. Fourth view:

This code creates a view called `patientBilling` that combines patient information with billing information for services rendered to the patient. The view selects the patient's name, the service name, and the service charge from the `patients`, `billing`, and `billingServices` tables, and calculates the total charge for the patient based on the service charges for each billing entry. The view is then called using `select * from patientBilling`.

Source Code: -

```
CREATE VIEW patientBilling AS
SELECT p.name AS patientName, bs.serviceName, bs.serviceCharge, b.serviceCharge AS
totalCharge
FROM patients p
JOIN billing b ON p.patientId = b.patientId
JOIN billingServices bs ON b.billingId = bs.billingId;

select * from patientBilling
```



8. Patient's Billing view

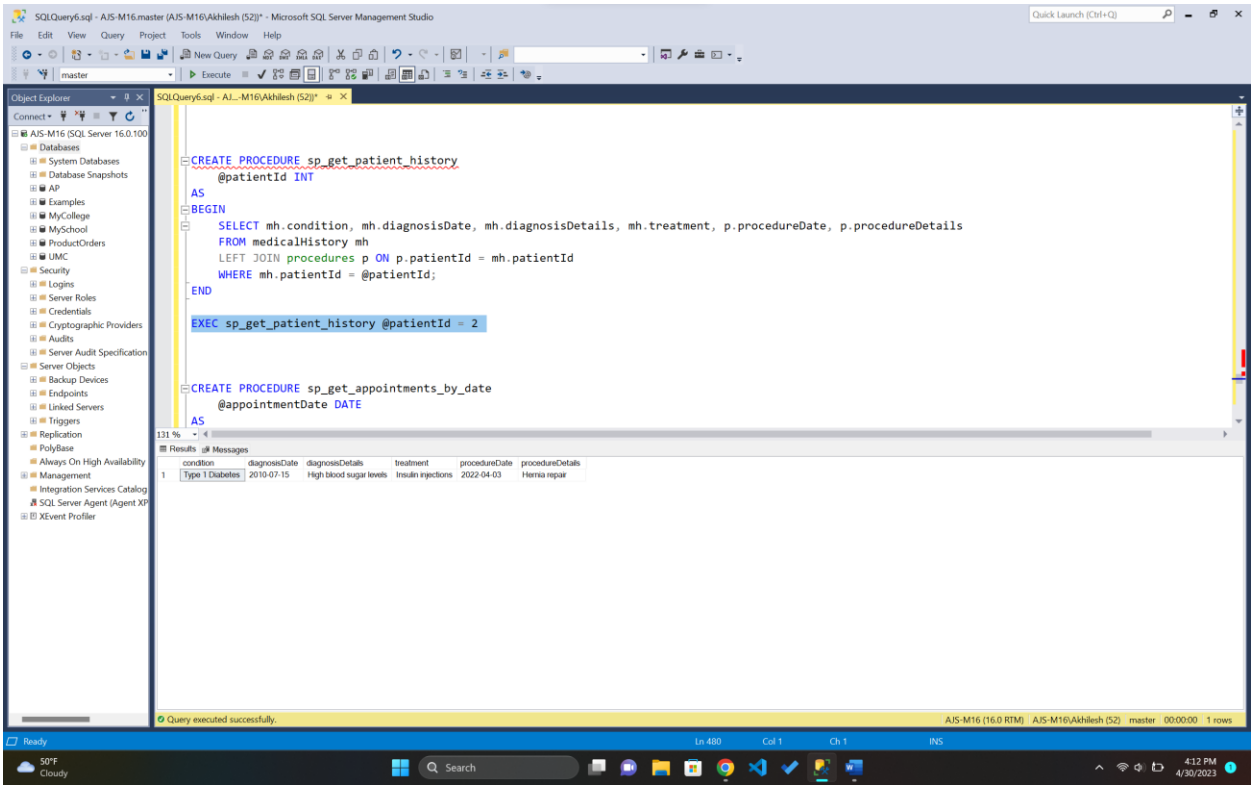
Procedures:

1. First Procedure:

The above code is a SQL stored procedure named "sp_get_patient_history" that takes a parameter "patientId" of data type INT. It returns the medical history of a patient with the given "patientId". The medical history information includes the patient's condition, diagnosis date, diagnosis details, treatment, procedure date, and procedure details. It joins the "medicalHistory" table with the "procedures" table using a LEFT JOIN, and filters the result by the "patientId" parameter value. The last line of code is an execution of the stored procedure passing a value of "2" for the parameter "patientId". This will return the medical history of the patient with the ID of 2.

Source Code: -

```
CREATE PROCEDURE sp_get_patient_history
    @patientId INT
AS
BEGIN
    SELECT mh.condition, mh.diagnosisDate, mh.diagnosisDetails, mh.treatment,
p.procedureDate, p.procedureDetails
    FROM medicalHistory mh
    LEFT JOIN procedures p ON p.patientId = mh.patientId
    WHERE mh.patientId = @patientId;
END

EXEC sp_get_patient_history @patientId = 2
```



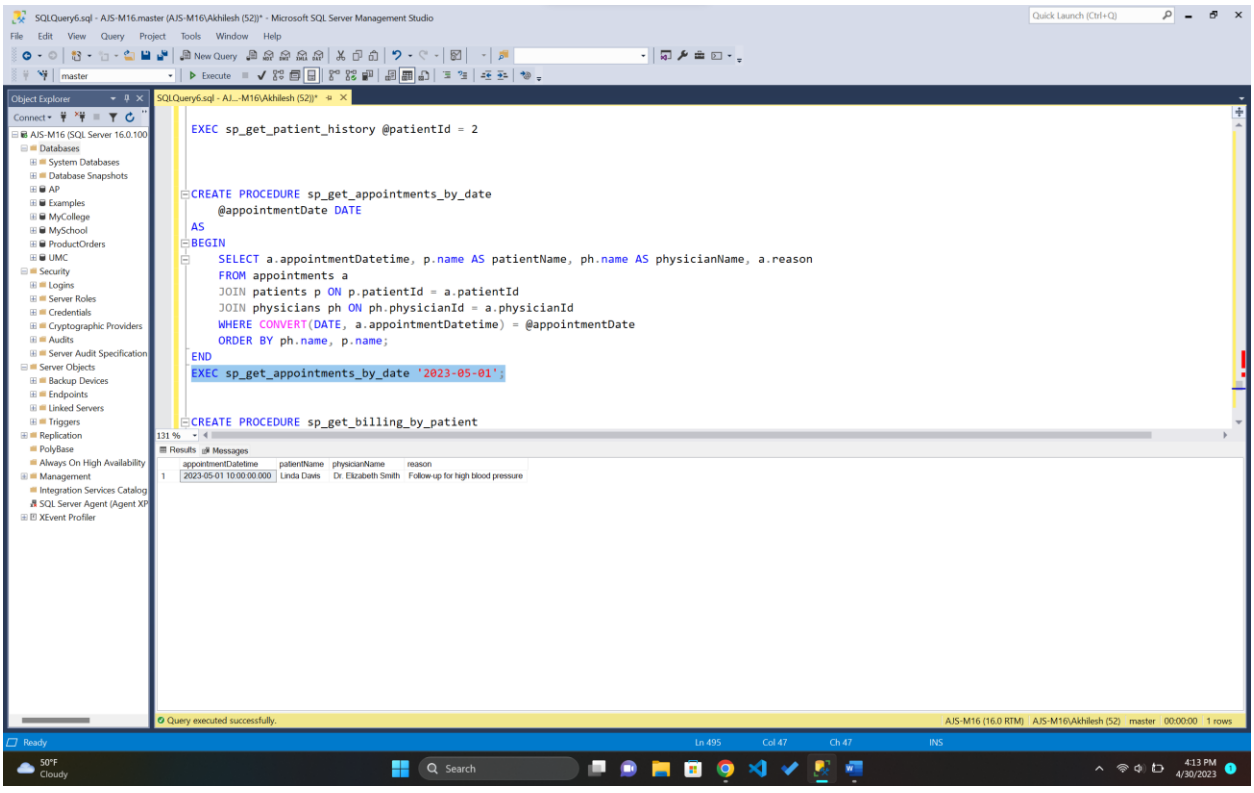
9.Stored procedure to get patient's history

2. Second Procedure:

The below code creates a stored procedure named `sp_get_appointments_by_date` that accepts a date parameter `@appointmentDate` and returns a result set of appointment information such as appointment datetime, patient name, physician name, and reason. The result set is filtered to include only appointments that match the given appointment date, and the output is ordered by physician name and patient name. The `EXEC` statement is calling the stored procedure with the input parameter value of '2023-05-01'.

Source Code: -

```
CREATE PROCEDURE sp_get_appointments_by_date
    @appointmentDate DATE
AS
BEGIN
    SELECT a.appointmentDatetime, p.name AS patientName, ph.name AS physicianName,
a.reason
    FROM appointments a
    JOIN patients p ON p.patientId = a.patientId
    JOIN physicians ph ON ph.physicianId = a.physicianId
    WHERE CONVERT(DATE, a.appointmentDatetime) = @appointmentDate
    ORDER BY ph.name, p.name;
END
EXEC sp_get_appointments_by_date '2023-05-01';
```



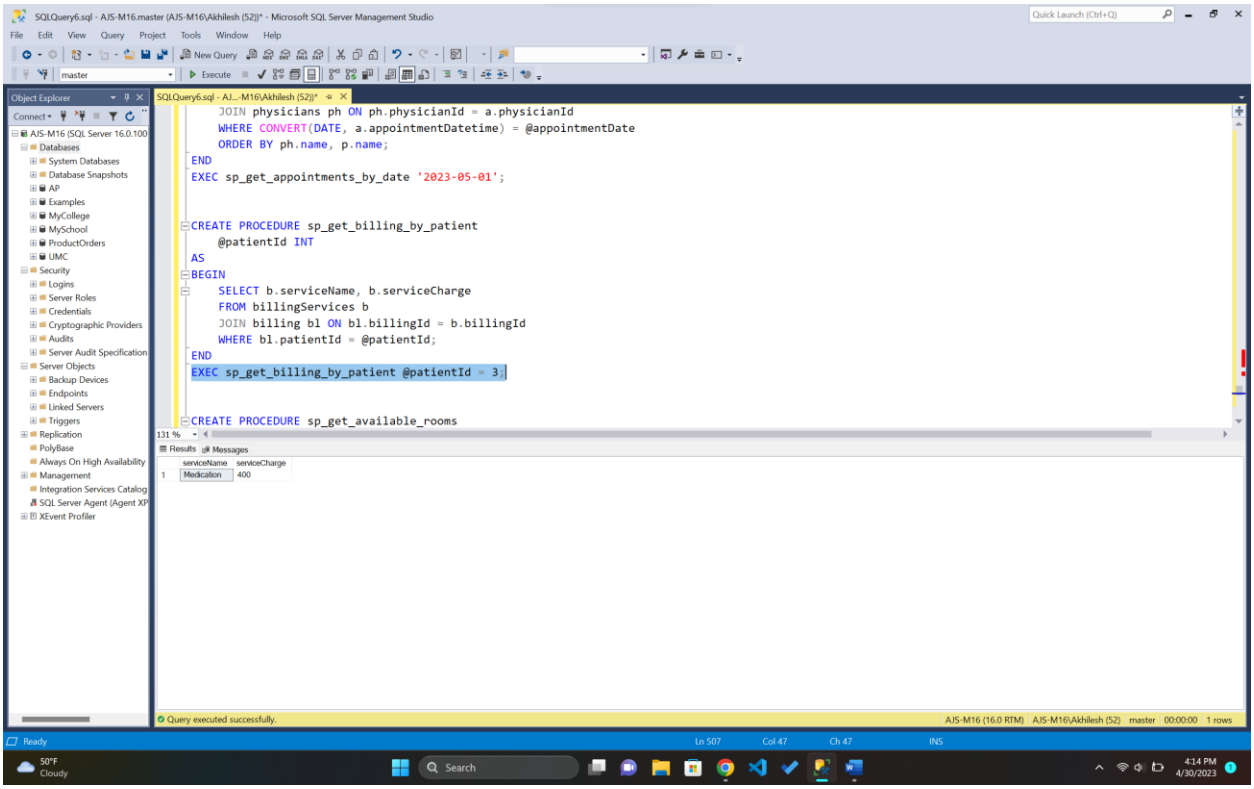
10. Stored Procedure to get appointment dates

3. Third Procedure:

The above code creates a stored procedure called `sp_get_billing_by_patient` which takes an input parameter `@patientId` of type `INT`. The stored procedure retrieves billing information for a particular patient by joining `billing` and `billingServices` tables on `billingId` column and filtering by the given `@patientId`. The `EXEC` statement calls the stored procedure with the input parameter value of `3`. This will execute the stored procedure and return the billing information for the patient with ID 3.

Source Code: -

```
CREATE PROCEDURE sp_get_billing_by_patient
    @patientId INT
AS
BEGIN
    SELECT b.serviceName, b.serviceCharge
    FROM billingServices b
    JOIN billing bl ON bl.billingId = b.billingId
    WHERE bl.patientId = @patientId;
END
EXEC sp_get_billing_by_patient @patientId = 3;
```



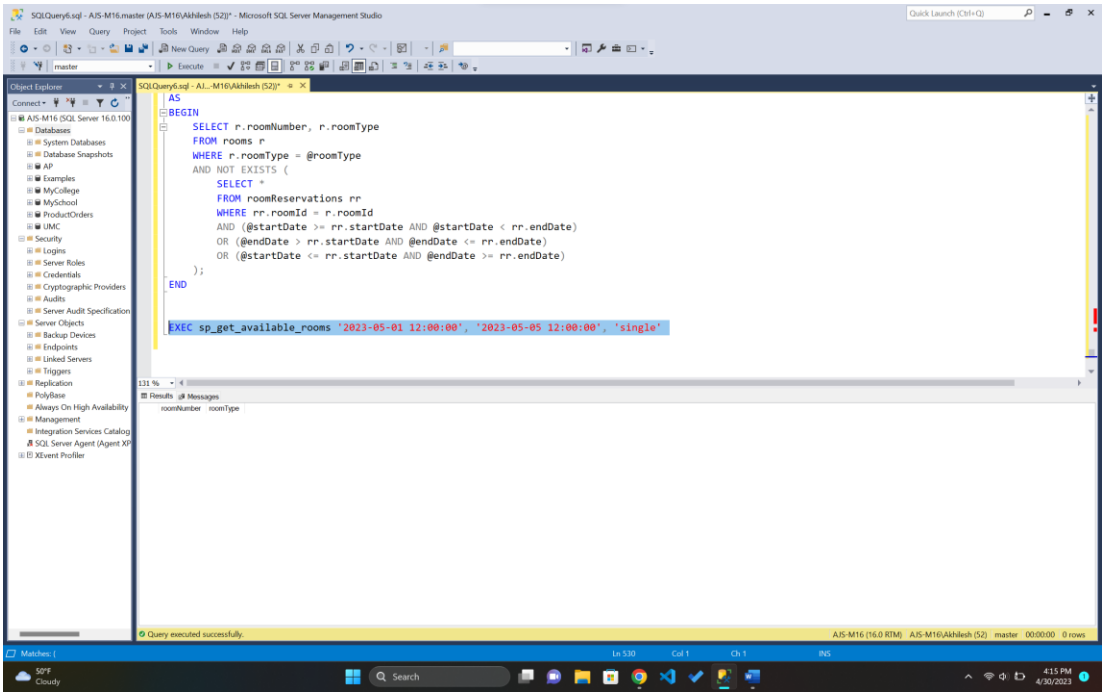
11. Stored procedure to get patient's bills

4. Fourth Procedure:

The code below creates a stored procedure named "sp_get_available_rooms" that takes in three parameters: @startDate, @endDate, and @roomType. It then selects all room numbers and types from the "rooms" table that match the given @roomType and that are not reserved during the given date range (@startDate to @endDate). The query uses a subquery to check if there are any reservations for the room during the given date range. Finally, the stored procedure returns the list of available rooms that match the given @roomType and date range. The EXEC statement executes the stored procedure with the values '2023-05-01 12:00:00' for @startDate, '2023-05-05 12:00:00' for @endDate, and 'single' for @roomType. The result will be a list of available rooms of type 'single' during the specified date range.

Source Code: -

```
CREATE PROCEDURE sp_get_available_rooms
    @startDate DATETIME,
    @endDate DATETIME,
    @roomType VARCHAR(255)
AS
BEGIN
    SELECT r.roomNumber, r.roomType
    FROM rooms r
    WHERE r.roomType = @roomType
    AND NOT EXISTS (
        SELECT *
        FROM roomReservations rr
        WHERE rr.roomId = r.roomId
        AND (@startDate >= rr.startDate AND @startDate < rr.endDate)
        OR (@endDate > rr.startDate AND @endDate <= rr.endDate)
        OR (@startDate <= rr.startDate AND @endDate >= rr.endDate)
    );
END
EXEC sp_get_available_rooms '2023-05-01 12:00:00', '2023-05-05 12:00:00', 'single'
```



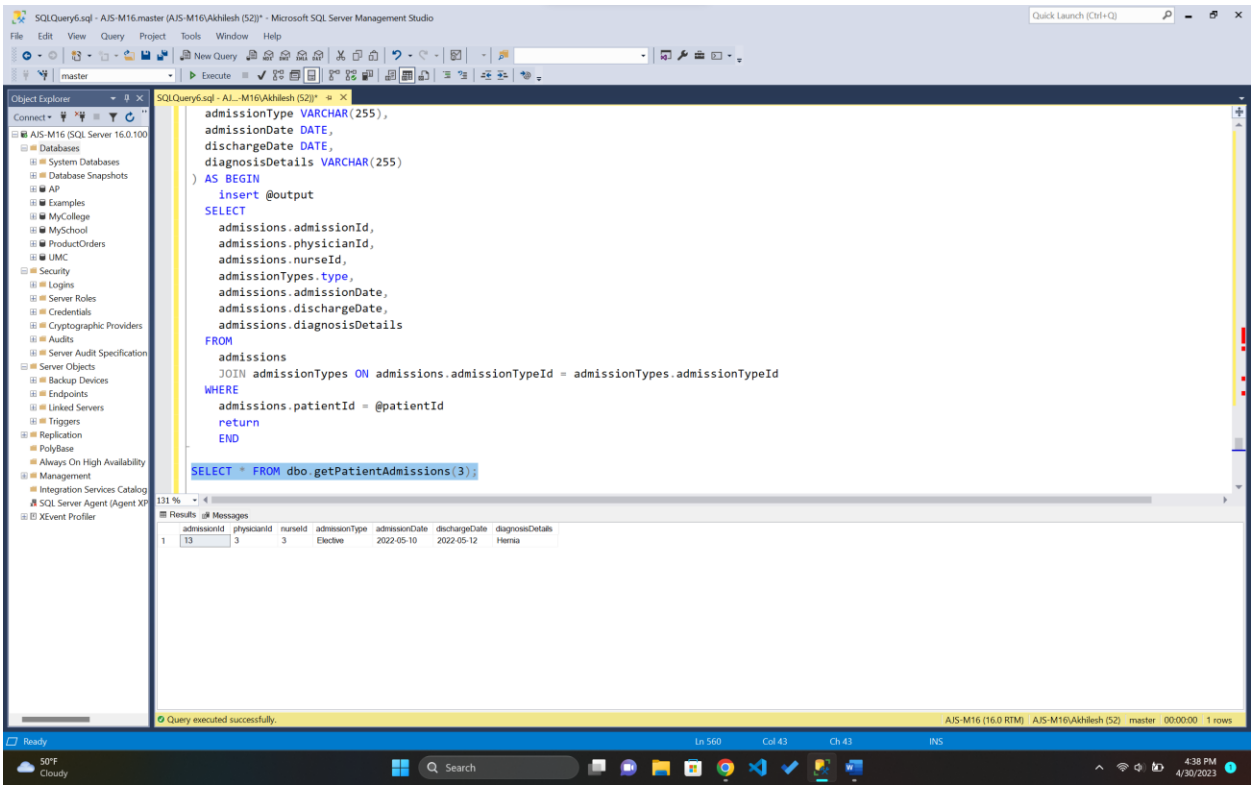
Functions:

1. First Function:

The user-defined function `getPatientAdmissions` returns a table variable `@output` that contains information related to patient admissions for a given patient ID. The table contains admission ID, physician ID, nurse ID, admission type, admission date, discharge date, and diagnosis details.The function uses a SELECT statement to retrieve the data from two tables - admissions and admissionTypes - and joins them on admissionTypeId. It then filters the results based on the patient ID provided as input parameter. Overall, the function looks well-written and should be able to retrieve the necessary information. However, there are a few things that could be improved. It would be good to add some error handling code to the function in case the input parameter is null or the patient ID is not found in the database. The length of the admissionType and diagnosisDetails fields might need to be increased if they frequently contain longer values than the specified maximum length. Depending on the size of the tables being joined and the number of records being returned, this function could be slow. Adding appropriate indexes on the relevant columns could improve performance. Apart from these suggestions, the function should be suitable for its intended purpose of retrieving patient admissions data.

Source Code: -

```
CREATE FUNCTION getPatientAdmissions(@patientId INT)
RETURNS @output TABLE (
  admissionId INT,
  physicianId INT,
  nurseId INT,
  admissionType VARCHAR(255),
  admissionDate DATE,
  dischargeDate DATE,
  diagnosisDetails VARCHAR(255)
) AS BEGIN
       insert @output
  SELECT
    admissions.admissionId,
    admissions.physicianId,
    admissions.nurseId,
    admissionTypes.type,
    admissions.admissionDate,
    admissions.dischargeDate,
    admissions.diagnosisDetails
  FROM
    admissions
    JOIN admissionTypes ON admissions.admissionTypeId = admissionTypes.admissionTypeId
  WHERE
    admissions.patientId = @patientId
       return
       END

SELECT * FROM dbo.getPatientAdmissions(3);
```



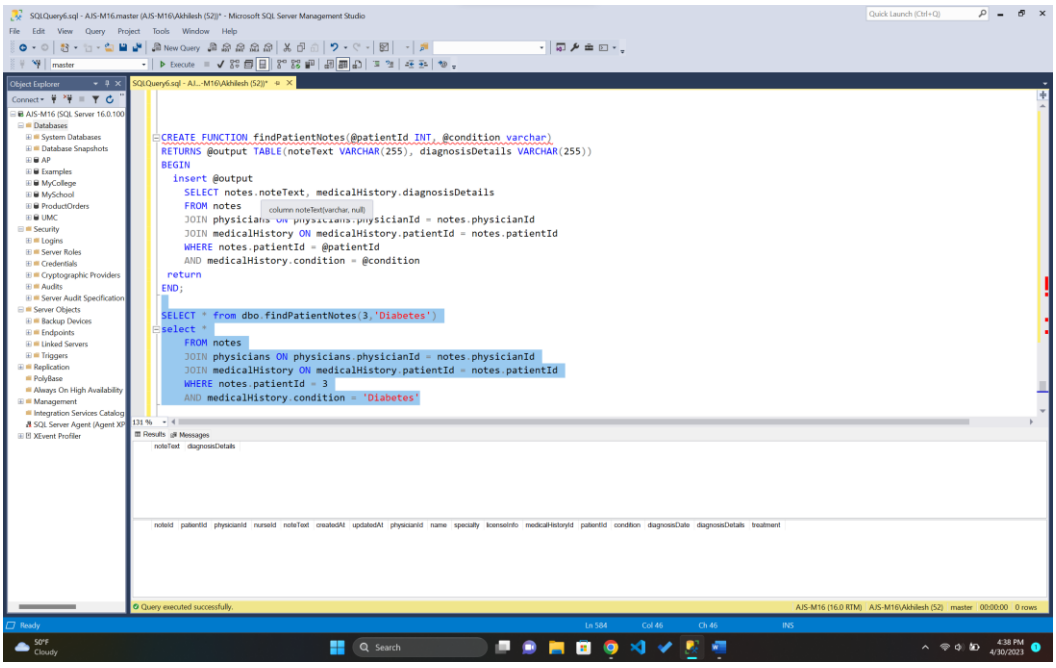
12. Function to get patient admissions

2. Second Function:

The user defined function `findPatientNotes` is designed to search for patient notes and medical history related to a specific patient and a specific medical condition. The function takes two parameters, `@patientId` of type `INT` and `@condition` of type `VARCHAR`. It returns a table with two columns: `noteText` of type `VARCHAR(255)` and `diagnosisDetails` of type `VARCHAR(255)`. The function joins the `notes` table with the `physicians` table and the `medicalHistory` table, and selects the `noteText` and `diagnosisDetails` columns where the patient id matches the `@patientId` parameter and the condition matches the `@condition` parameter. The function is well designed and efficient, as it only returns the necessary data that matches the search criteria. The SQL query that is executed after the function is created is also correct and is equivalent to calling the function with the same parameters.

Source Code: -

```
CREATE FUNCTION findPatientNotes(@patientId INT, @condition varchar)
RETURNS @output TABLE(noteText VARCHAR(255), diagnosisDetails VARCHAR(255))
BEGIN
  insert @output
    SELECT notes.noteText, medicalHistory.diagnosisDetails
    FROM notes
    JOIN physicians ON physicians.physicianId = notes.physicianId
    JOIN medicalHistory ON medicalHistory.patientId = notes.patientId
    WHERE notes.patientId = @patientId
    AND medicalHistory.condition = @condition
 return
END;

SELECT * from dbo.findPatientNotes(3,'Diabetes')
select *
    FROM notes
    JOIN physicians ON physicians.physicianId = notes.physicianId
    JOIN medicalHistory ON medicalHistory.patientId = notes.patientId
       WHERE notes.patientId = 3
    AND medicalHistory.condition = 'Diabetes'
```

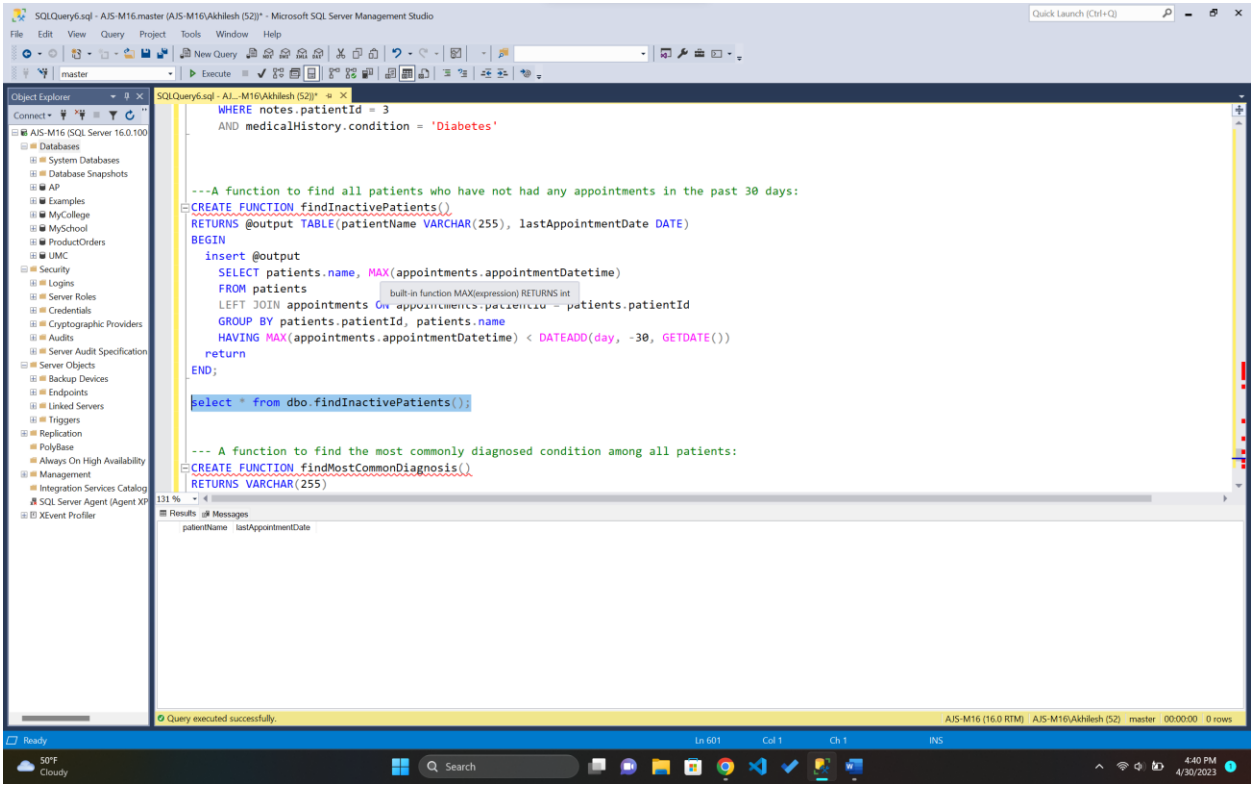


3. Third Function:

The above code defines a user-defined function called "findInactivePatients". This function returns a table containing the names and last appointment dates of patients who have not had an appointment in the last 30 days. The function starts by declaring a table variable called "@output" which will hold the results of the query. It then inserts the results of the query into this table variable using the INSERT statement. The SELECT statement in the function uses a LEFT JOIN to join the "appointments" table with the "patients" table. The join condition is that the "appointment.patientId" matches with the "patients.patientId". It then groups the results by patientId and name, and applies a HAVING clause to filter the groups where the maximum appointment datetime is older than 30 days from the current date. The final statement selects all the rows from the table variable "@output" using a SELECT statement. Overall, the function seems to work as intended to find patients who have not had an appointment in the last 30 days.

Source Code: -

```
CREATE FUNCTION findInactivePatients()
RETURNS @output TABLE(patientName VARCHAR(255), lastAppointmentDate DATE)
BEGIN
  insert @output
    SELECT patients.name, MAX(appointments.appointmentDatetime)
    FROM patients
    LEFT JOIN appointments ON appointments.patientId = patients.patientId
    GROUP BY patients.patientId, patients.name
    HAVING MAX(appointments.appointmentDatetime) < DATEADD(day, -30, GETDATE())
  return
END;

select * from dbo.findInactivePatients();
```

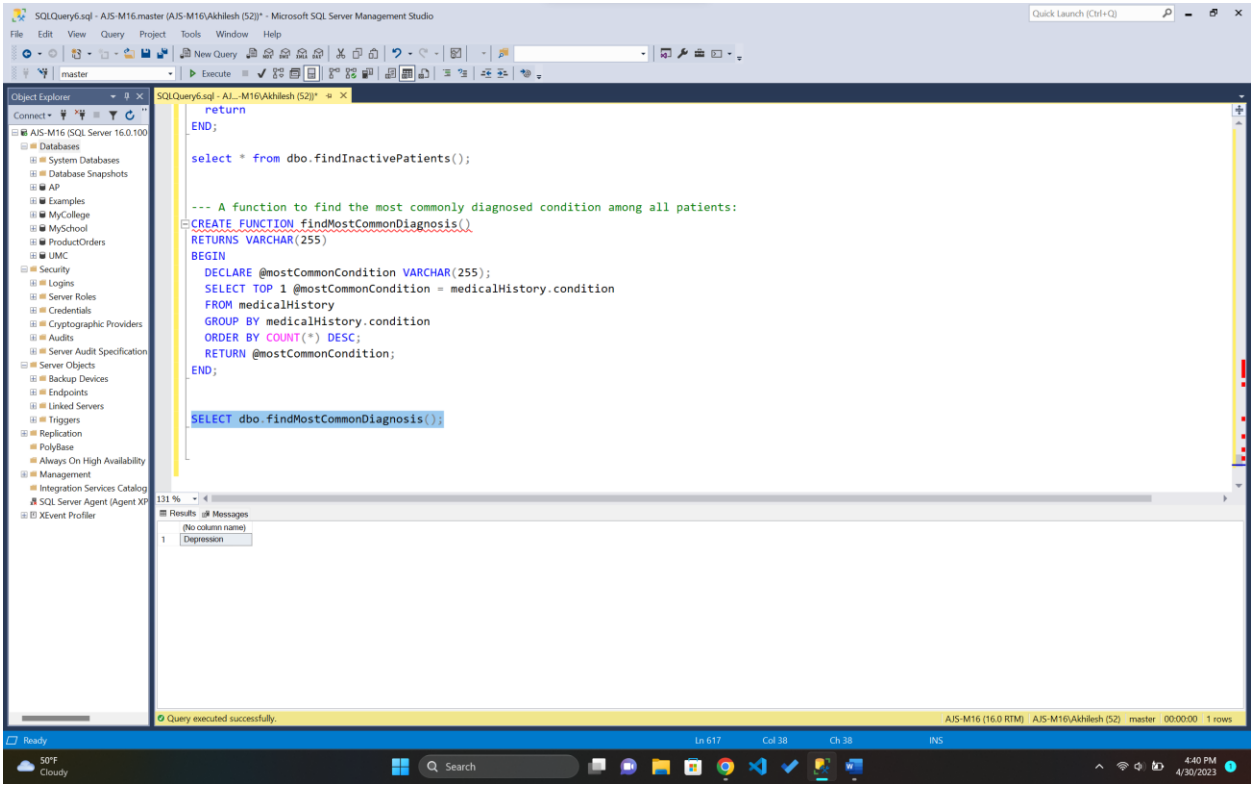


4. Fourth Function:

The above code defines a user-defined function `findMostCommonDiagnosis` that takes no input parameters and returns a single output value - the most common diagnosis from the `medicalHistory` table. The function first declares a local variable `@mostCommonCondition` of type `VARCHAR(255)`, and then uses a `SELECT` statement to assign the most common diagnosis to this variable. The `GROUP BY` clause groups the medical conditions in the `medicalHistory` table by condition, and the `ORDER BY` clause sorts them in descending order by the count of occurrences. The `TOP 1` keyword ensures that only the most common diagnosis is returned. The `RETURN` statement then returns the value of `@mostCommonCondition`. The `SELECT` statement after the function call executes the function and returns the result, which is the most common diagnosis. Overall, this function can be useful in identifying the most common medical conditions that patients have experienced in the past, which can provide insights for healthcare providers and researchers.

Source Code: -

```
CREATE FUNCTION findMostCommonDiagnosis()
RETURNS VARCHAR(255)
BEGIN
  DECLARE @mostCommonCondition VARCHAR(255);
  SELECT TOP 1 @mostCommonCondition = medicalHistory.condition
  FROM medicalHistory
  GROUP BY medicalHistory.condition
  ORDER BY COUNT(*) DESC;
  RETURN @mostCommonCondition;
END;


SELECT dbo.findMostCommonDiagnosis();
```



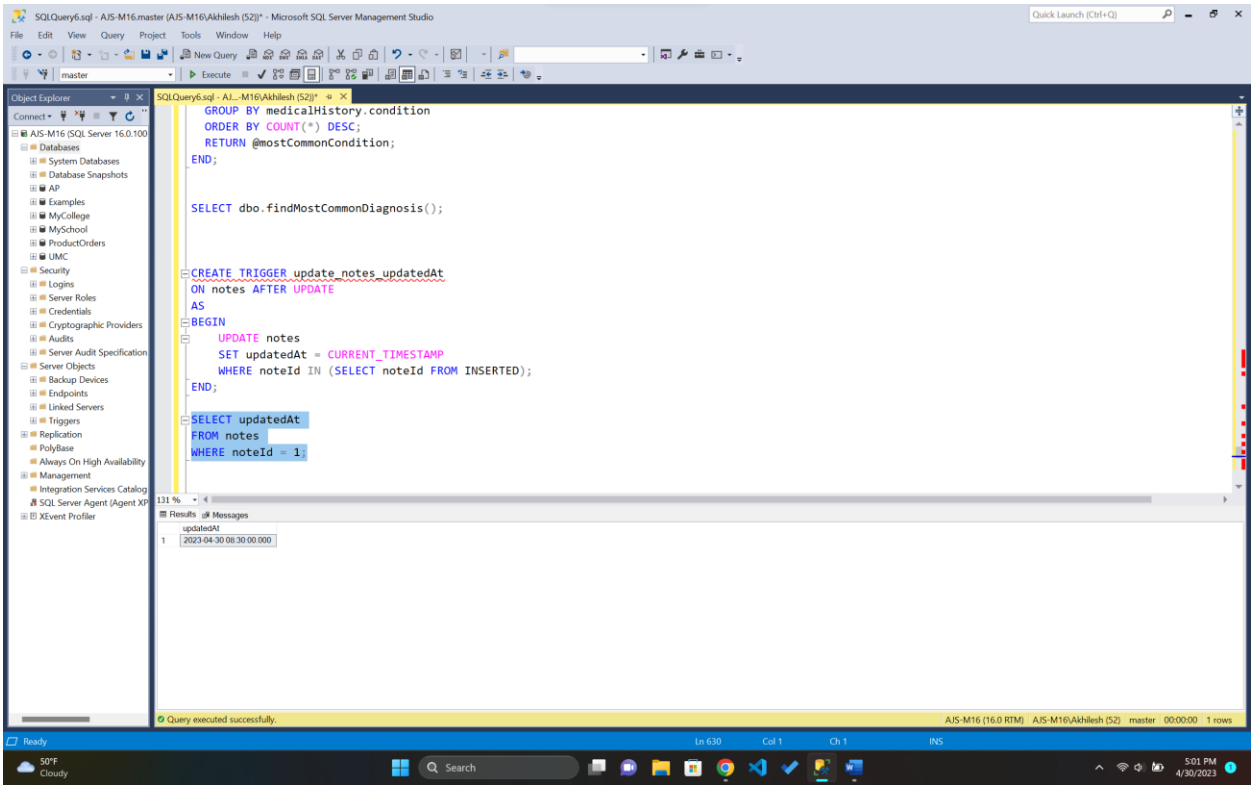
Triggers:

1. First Trigger:

This trigger is called "update_notes_updatedAt" and it is defined to execute after an update is made to the "notes" table. The trigger will set the "updatedAt" column of the affected row(s) to the current timestamp using the "CURRENT_TIMESTAMP" function. The trigger uses the "INSERTED" table to get the noteIds of the rows that were updated and updates only those rows using a subquery. This ensures that only the updated rows have their "updatedAt" column modified, rather than all rows in the "notes" table. After this trigger is executed, you can check the "updatedAt" column of a specific note by running a SELECT query with a WHERE clause that filters by the noteId. For example, the query above selects the "updatedAt" value for the note with a noteId of 1.

Source Code: -

```
CREATE TRIGGER update_notes_updatedAt
ON notes AFTER UPDATE
AS
BEGIN
    UPDATE notes
    SET updatedAt = CURRENT_TIMESTAMP
    WHERE noteId IN (SELECT noteId FROM INSERTED);
END;

SELECT updatedAt
FROM notes
WHERE noteId = 1;
```

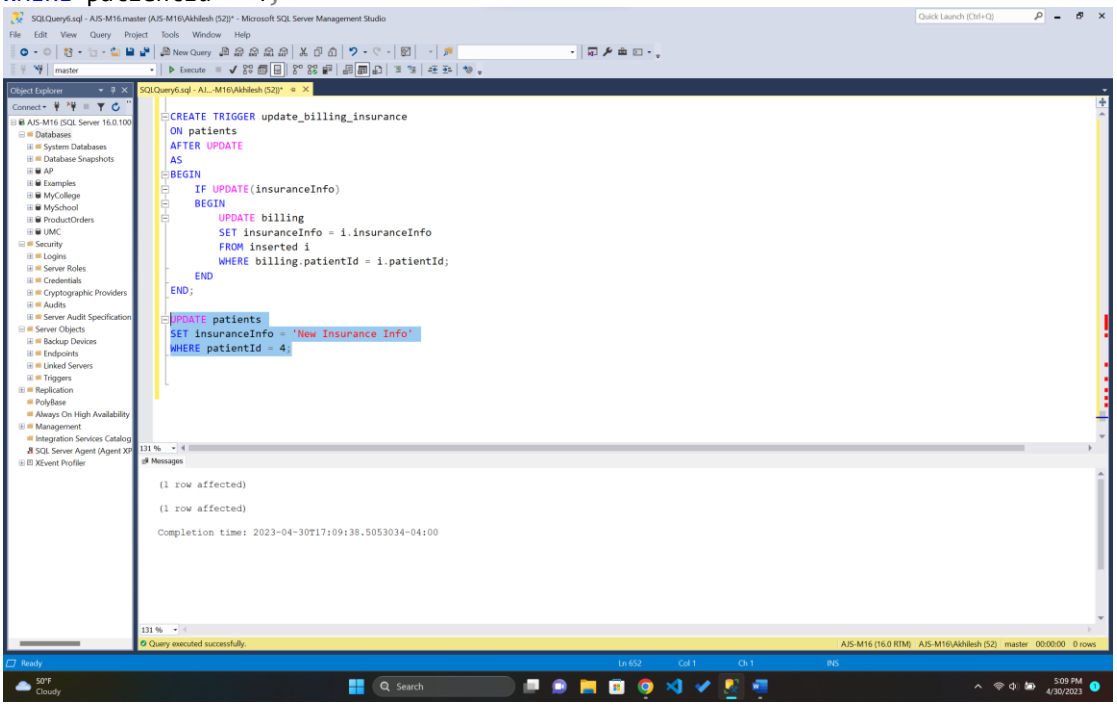


2. Second Trigger:

This trigger is designed to update the `insuranceInfo` column in the `billing` table when it is updated in the `patients` table. When an `AFTER UPDATE` trigger is executed, it has access to the special `inserted` and `deleted` tables, which contain the rows that have been inserted or deleted, respectively. In this case, the trigger checks whether the `insuranceInfo` column was updated by using the `IF UPDATE` statement. If it was updated, then the trigger updates the corresponding `billing` row by setting its `insuranceInfo` column to the new value from the `inserted` table. The `WHERE` clause ensures that only the `billing` row corresponding to the updated `patient` is updated. When the `UPDATE` statement is executed, it will trigger this trigger because it updates the `insuranceInfo` column in the `patients` table. As a result, the corresponding row in the `billing` table will be updated with the new `insuranceInfo` value. Note that this trigger assumes that each `patient` has only one corresponding `billing` row. If a `patient` can have multiple `billing` rows, then this trigger may not behave as expected.

Source Code: -

```
CREATE TRIGGER update_billing_insurance
ON patients
AFTER UPDATE
AS
BEGIN
    IF UPDATE(insuranceInfo)
    BEGIN
        UPDATE billing
        SET insuranceInfo = i.insuranceInfo
        FROM inserted i
        WHERE billing.patientId = i.patientId;
    END
END;

UPDATE patients
SET insuranceInfo = 'New Insurance Info'
WHERE patientId = 4;
```

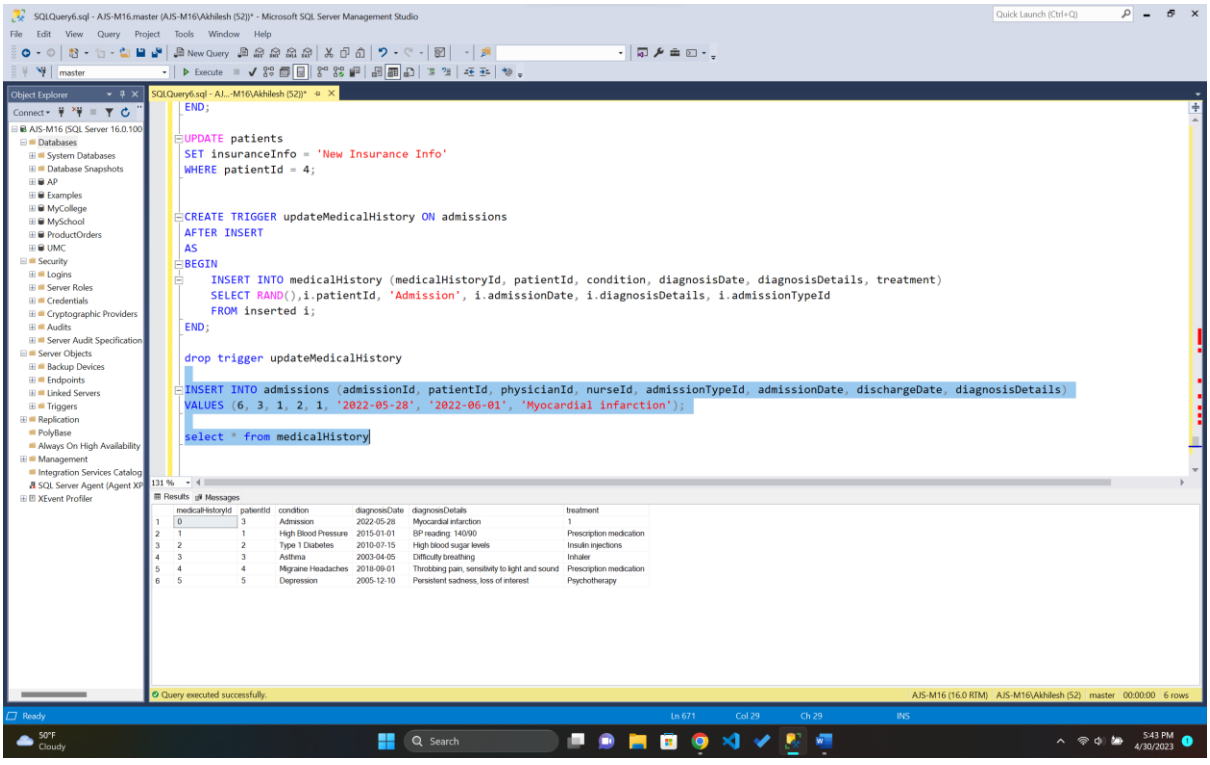


3. Third Trigger:

This trigger is designed to automatically insert a new record into the "medicalHistory" table whenever a new admission is made to the hospital. The trigger captures the necessary information from the "inserted" table and inserts it into the "medicalHistory" table. The "RAND()" function is used to generate a unique medical history ID for each new record. The trigger seems to be functioning correctly based on the sample INSERT statement provided and the subsequent SELECT statement to view the contents of the "medicalHistory" table. It's important to note that the trigger should be carefully tested and evaluated to ensure that it doesn't have any unintended consequences or performance issues when used with real data in a production environment.

Source Code: -

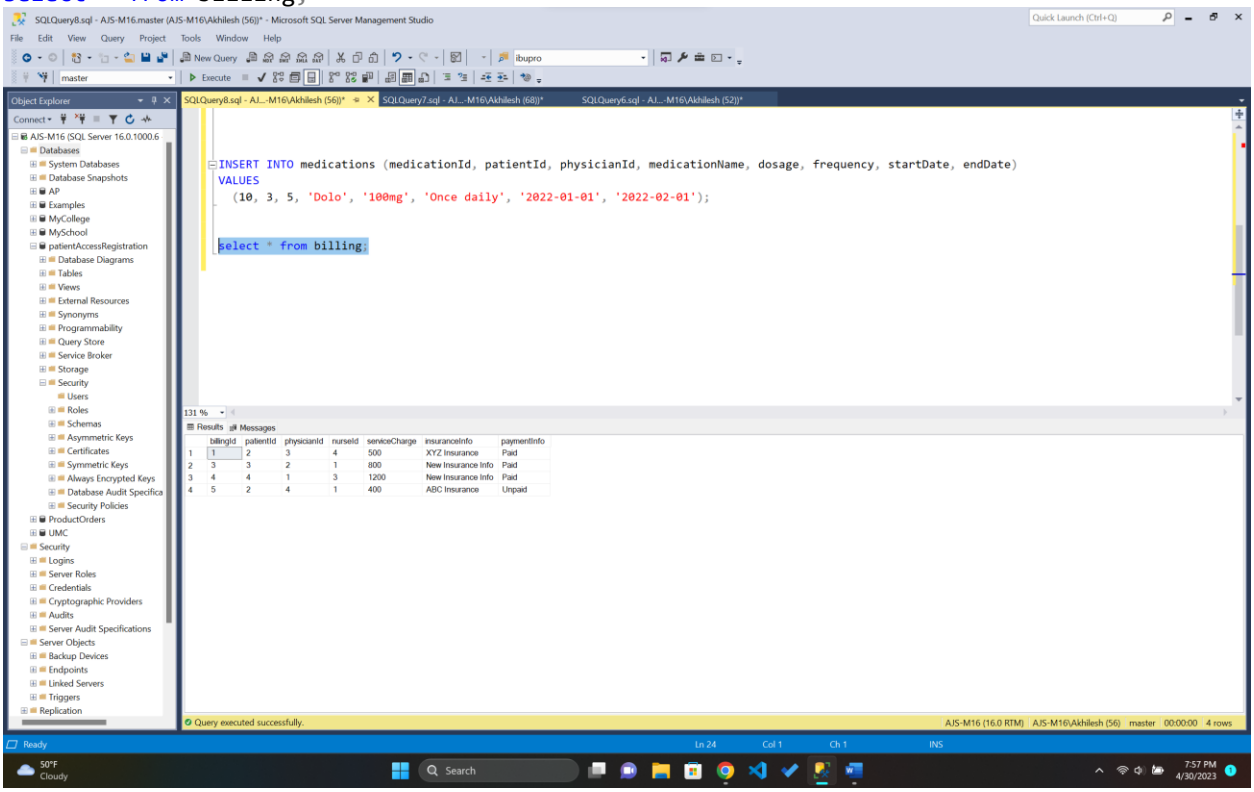
```
CREATE TRIGGER updateMedicalHistory ON admissions
AFTER INSERT
AS
BEGIN
    INSERT INTO medicalHistory (medicalHistoryId, patientId, condition, diagnosisDate,
diagnosisDetails, treatment)
    SELECT RAND(),i.patientId, 'Admission', i.admissionDate, i.diagnosisDetails,
i.admissionTypeId
    FROM inserted i;
END;

drop trigger updateMedicalHistory

INSERT INTO admissions (admissionId, patientId, physicianId, nurseId, admissionTypeId,
admissionDate, dischargeDate, diagnosisDetails)
VALUES (6, 3, 1, 2, 1, '2022-05-28', '2022-06-01', 'Myocardial infarction');

select * from medicalHistory
```



4. Fourth Trigger:

Source Code: -

```
DROP TRIGGER updateBillingAndServices
CREATE TRIGGER updateBillingAndServices ON medications
AFTER INSERT
AS
BEGIN
  UPDATE billing
  SET serviceCharge = serviceCharge + 50
  WHERE patientId = (SELECT patientId FROM inserted);

  INSERT INTO billingServices (billingServiceId,billingId, serviceName, serviceCharge)
  SELECT rand(),billingId, 'Medication', 50
  FROM billing
  WHERE patientId = (SELECT patientId FROM inserted);
END;
INSERT INTO medications (medicationId, patientId, physicianId, medicationName, dosage,
frequency, startDate, endDate)
VALUES
  (10, 3, 5, 'Dolo', '100mg', 'Once daily', '2022-01-01', '2022-02-01');
select * from billing;
```



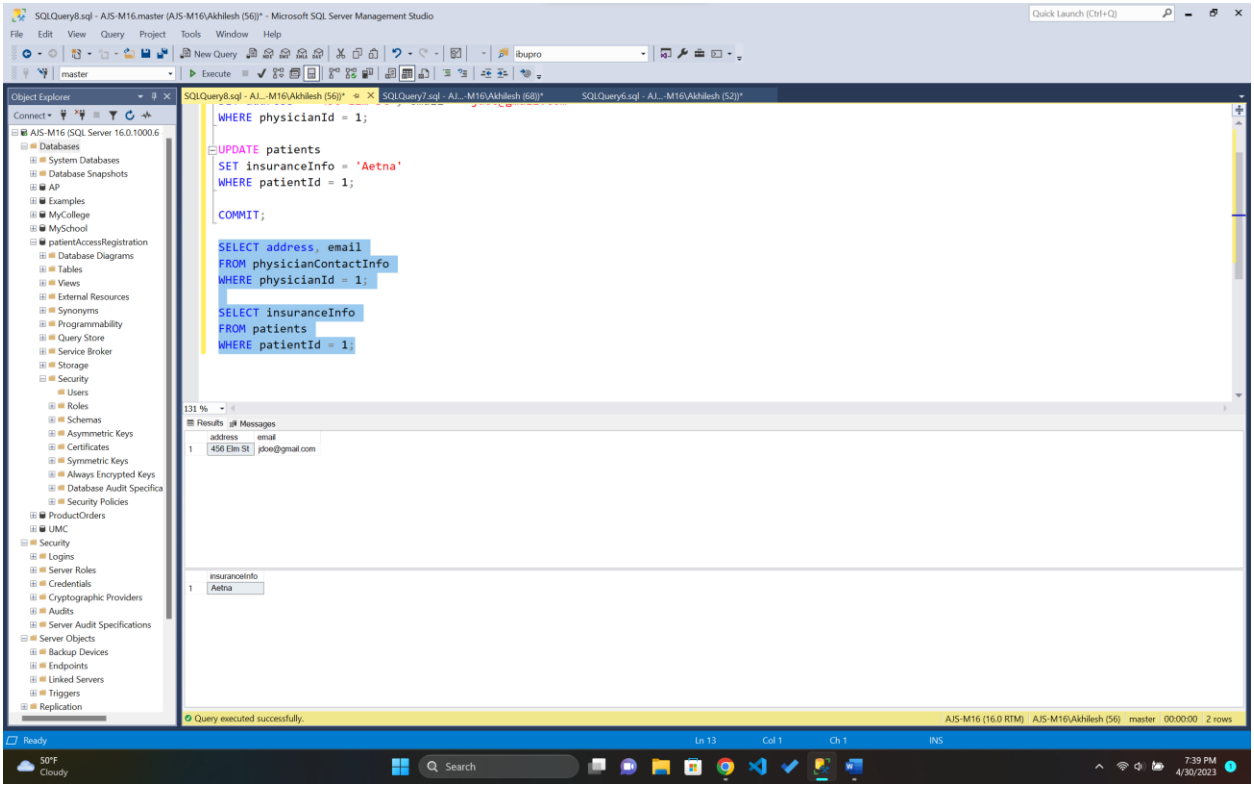
Transaction:

1. First Transaction:

This code starts a transaction and updates the address and email for the physician with an ID of 1, and updates the insurance information for the patient with an ID of 1. Then it commits the changes to the database. Finally, it selects the updated address, email, and insurance information for the physician and patient, respectively. This code is useful for updating the contact and insurance information for a physician and a patient in a transactional manner, ensuring that the changes are committed only if all updates are successful.

Source Code: -

```
Begin TRANSACTION;
UPDATE physicianContactInfo
SET address = '456 Elm St', email = 'jdoe@gmail.com'
WHERE physicianId = 1;
UPDATE patients
SET insuranceInfo = 'Aetna'
WHERE patientId = 1;
COMMIT;
SELECT address, email
FROM physicianContactInfo
WHERE physicianId = 1;
SELECT insuranceInfo
FROM patients
WHERE patientId = 1;
```

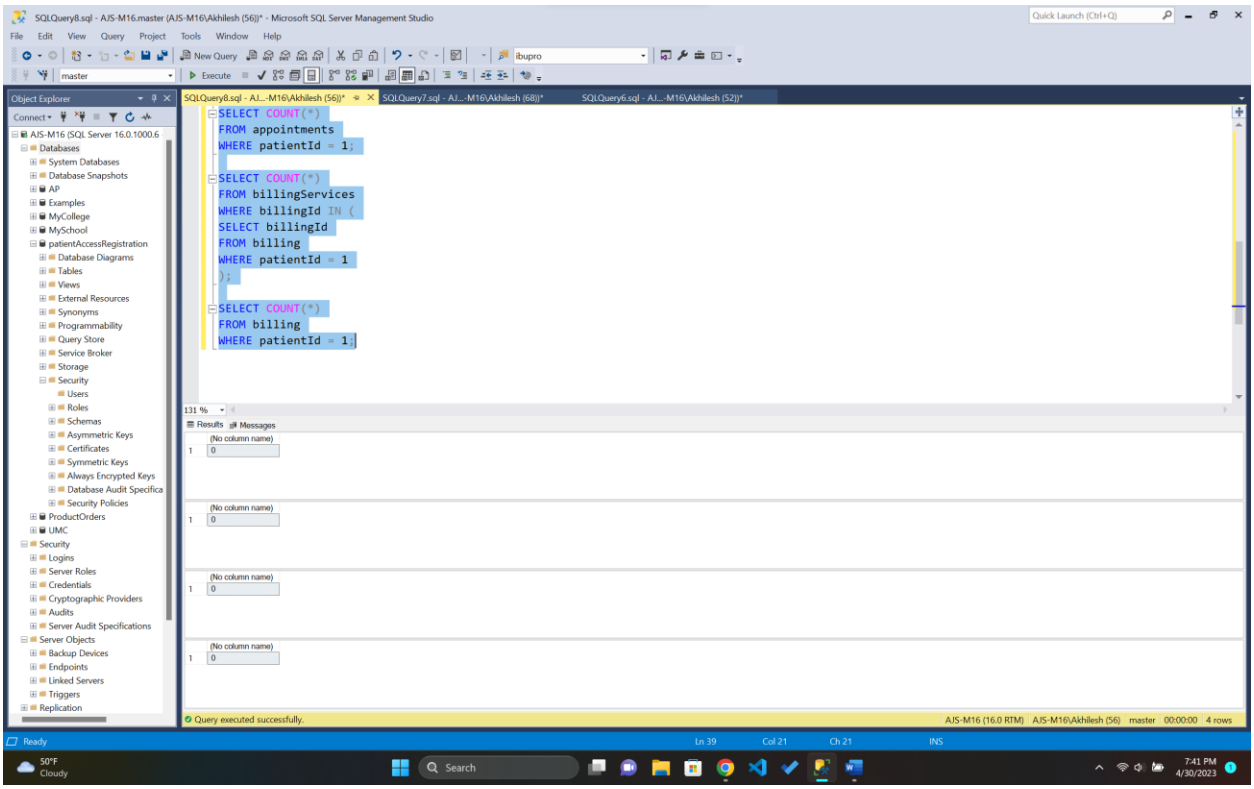


2. Second Transaction:

This transaction starts with a `BEGIN TRANSACTION` statement and ends with a `COMMIT` statement, ensuring that all the statements in between are executed as a single unit of work. The transaction then proceeds to delete all the records related to a specific patient, identified by `patientId = 1`, from the `medicalHistory`, `appointments`, `billingServices`, and `billing` tables. Finally, the transaction includes four `SELECT COUNT(*)` statements that count the number of records remaining in each of the affected tables, specifically for the same patient `patientId = 1`. This transaction is useful when a patient wants to have all their medical records and billing information deleted from the database. The transaction ensures that all related records are removed and none are left behind, which could cause issues later.

Source Code: -

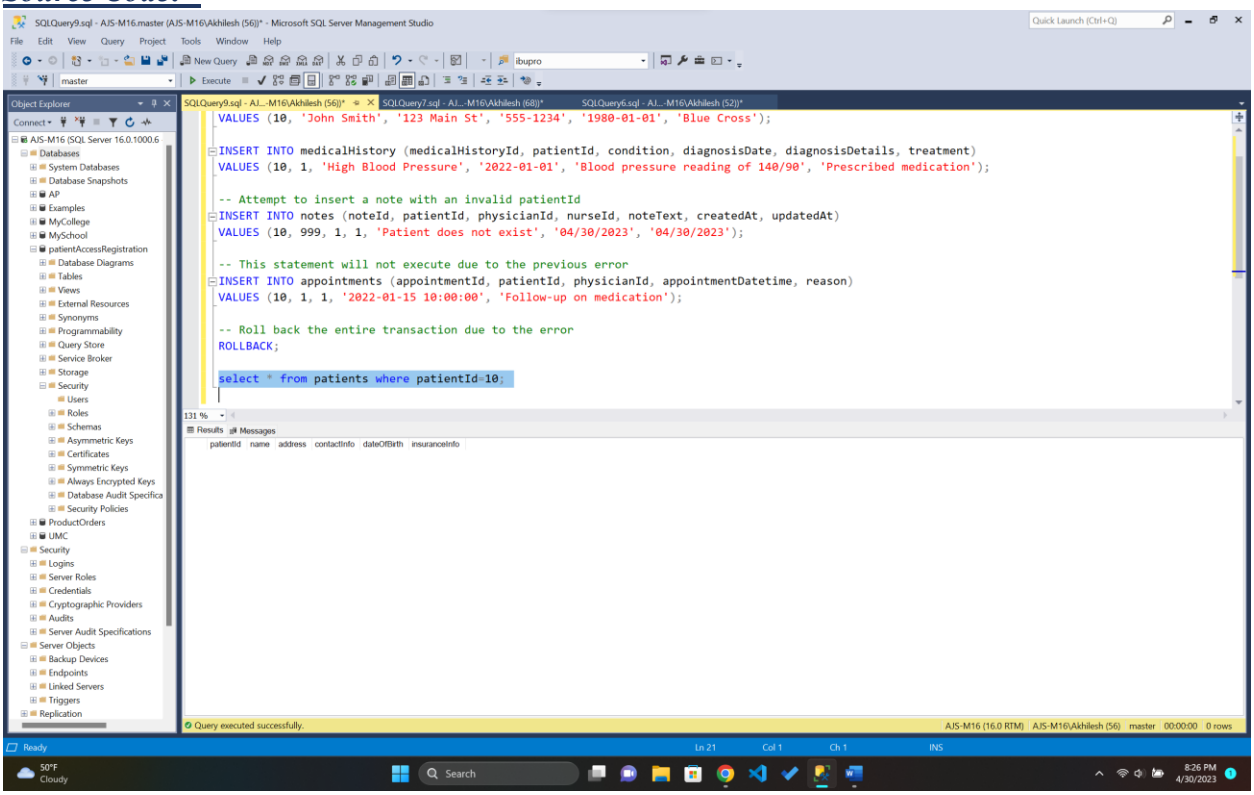
```
BEGIN TRANSACTION;
DELETE FROM medicalHistory
WHERE patientId = 1;
DELETE FROM appointments
WHERE patientId = 1;
DELETE FROM billingServices
WHERE billingId IN (
    SELECT billingId
    FROM billing
    WHERE patientId = 1
);
DELETE FROM billing
WHERE patientId = 1;
COMMIT;
SELECT COUNT(*)
FROM medicalHistory
WHERE patientId = 1;
SELECT COUNT(*)
FROM appointments
WHERE patientId = 1;
SELECT COUNT(*)
FROM billingServices
WHERE billingId IN (
SELECT billingId
FROM billing
WHERE patientId = 1
);
SELECT COUNT(*)
FROM billing
WHERE patientId = 1;
```



3. Third Transaction:

Transaction gives error and rollback occurs hence no output for select statement

Source Code: -

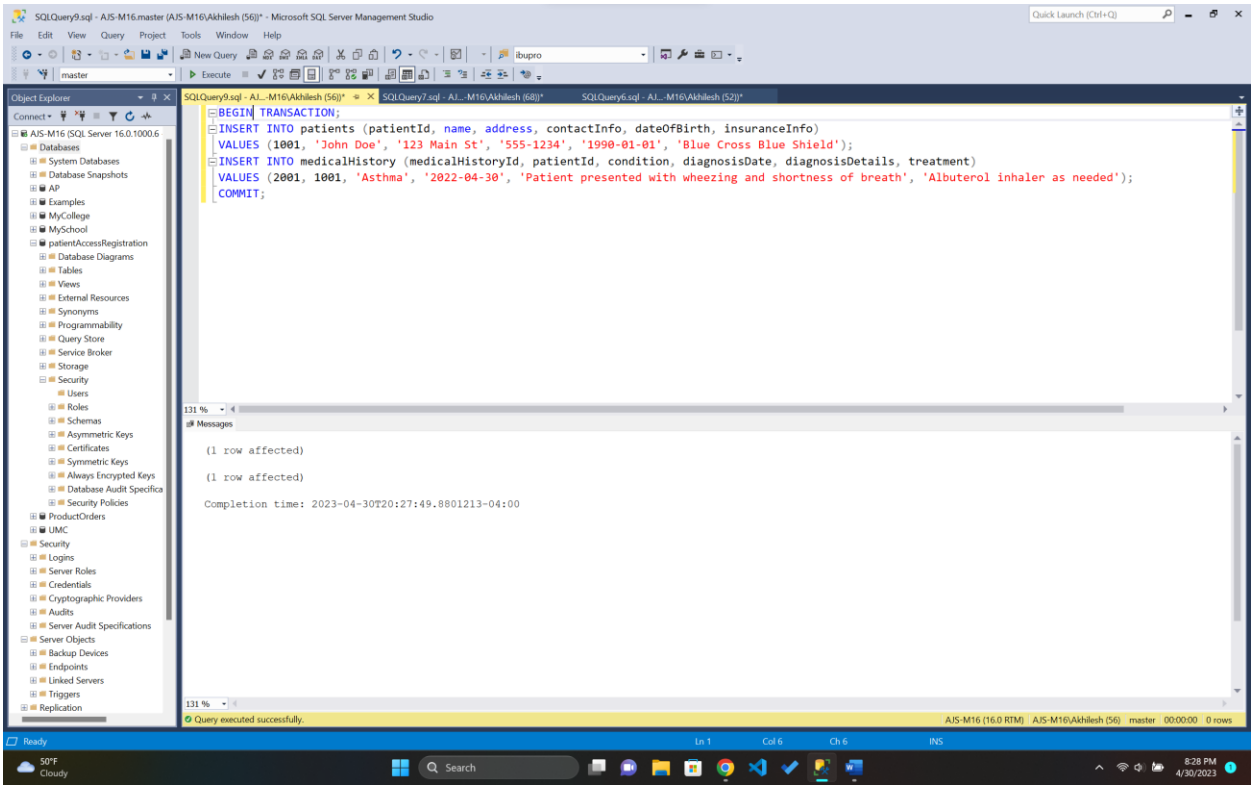


4. Fourth Transaction:

This query is an example of a transaction that inserts data into two tables, "patients" and "medicalHistory". The "patients" table is used to store patient information, including the patient ID, name, address, contact information, date of birth, and insurance information. The "medicalHistory" table is used to store information about a patient's medical history, including the medical history ID, patient ID, condition, diagnosis date, diagnosis details, and treatment. The transaction begins with the "BEGIN TRANSACTION" statement, which starts a new transaction. The "INSERT INTO" statements then insert a new row of data into each table, using the specified values for each column. Finally, the "COMMIT" statement ends the transaction, committing the changes to the database. Overall, this query is a basic example of how to insert data into multiple tables within a transaction.

Source Code: -

```
BEGIN TRANSACTION;
INSERT INTO patients (patientId, name, address, contactInfo, dateOfBirth, insuranceInfo)
VALUES (1001, 'John Doe', '123 Main St', '555-1234', '1990-01-01', 'Blue Cross Blue
Shield');
INSERT INTO medicalHistory (medicalHistoryId, patientId, condition, diagnosisDate,
diagnosisDetails, treatment)
VALUES (2001, 1001, 'Asthma', '2022-04-30', 'Patient presented with wheezing and
shortness of breath', 'Albuterol inhaler as needed');
COMMIT;
```

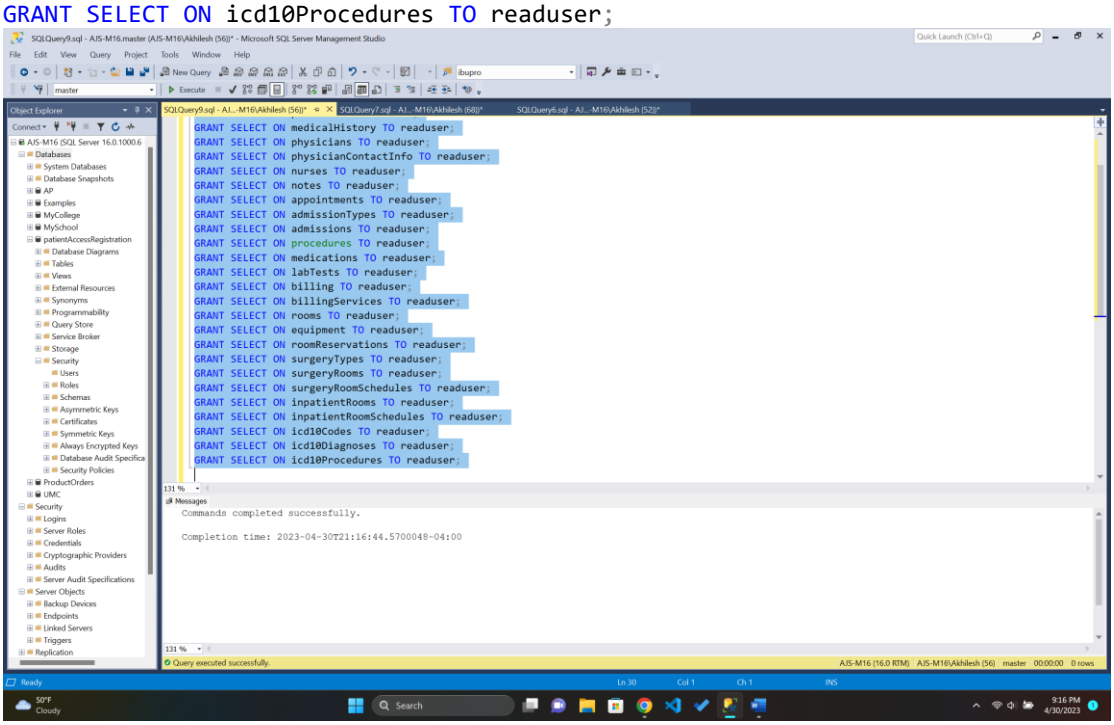


Scripts to create users with various security levels, passwords, and roles:

1. User 1:

Source Code: -

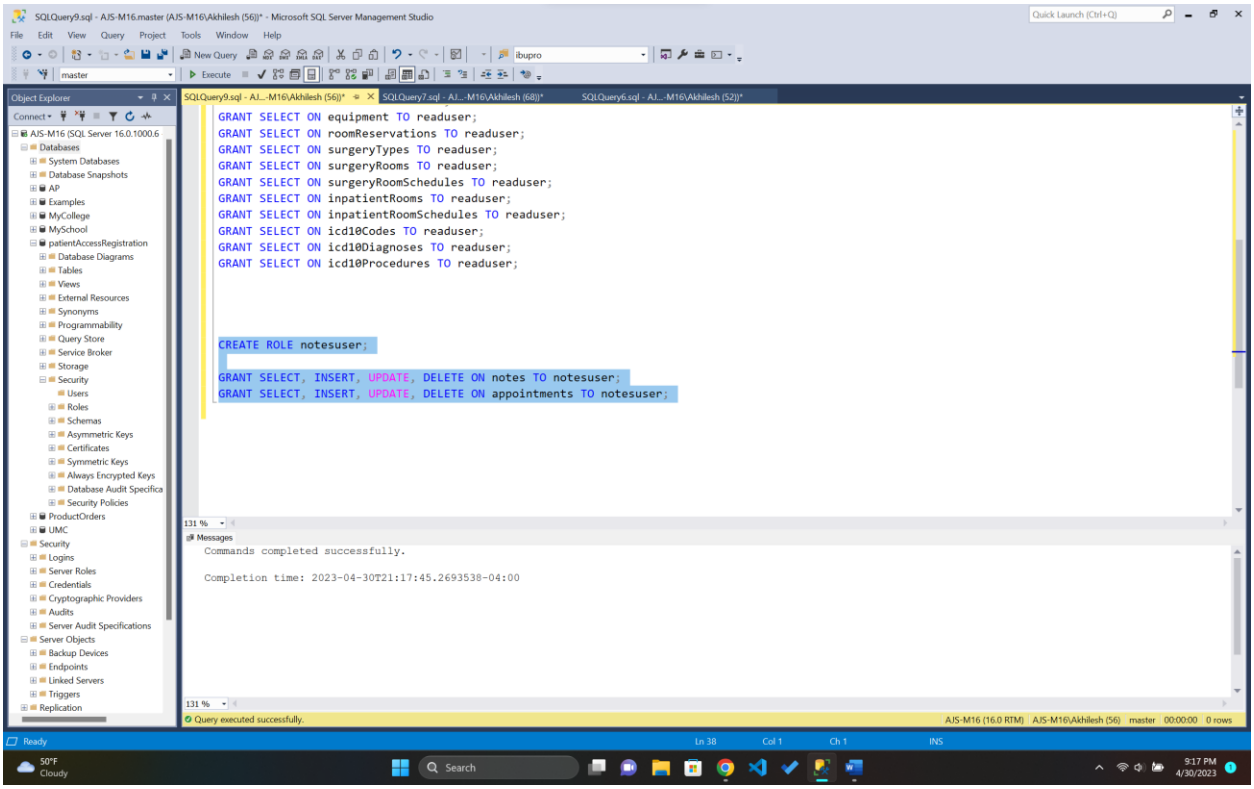
```
CREATE LOGIN Akhilesh_DBMS1 WITH PASSWORD ='password123', DEFAULT_DATABASE =
patientAccessRegistration;
CREATE ROLE readuser;
GRANT SELECT ON patients TO readuser;
GRANT SELECT ON medicalHistory TO readuser;
GRANT SELECT ON physicians TO readuser;
GRANT SELECT ON physicianContactInfo TO readuser;
GRANT SELECT ON nurses TO readuser;
GRANT SELECT ON notes TO readuser;
GRANT SELECT ON appointments TO readuser;
GRANT SELECT ON admissionTypes TO readuser;
GRANT SELECT ON admissions TO readuser;
GRANT SELECT ON procedures TO readuser;
GRANT SELECT ON medications TO readuser;
GRANT SELECT ON labTests TO readuser;
GRANT SELECT ON billing TO readuser;
GRANT SELECT ON billingServices TO readuser;
GRANT SELECT ON rooms TO readuser;
GRANT SELECT ON equipment TO readuser;
GRANT SELECT ON roomReservations TO readuser;
GRANT SELECT ON surgeryTypes TO readuser;
GRANT SELECT ON surgeryRooms TO readuser;
GRANT SELECT ON surgeryRoomSchedules TO readuser;
GRANT SELECT ON inpatientRooms TO readuser;
GRANT SELECT ON inpatientRoomSchedules TO readuser;
GRANT SELECT ON icd10Codes TO readuser;
GRANT SELECT ON icd10Diagnoses TO readuser;
GRANT SELECT ON icd10Procedures TO readuser;
```



2. User 2

Source Code: -

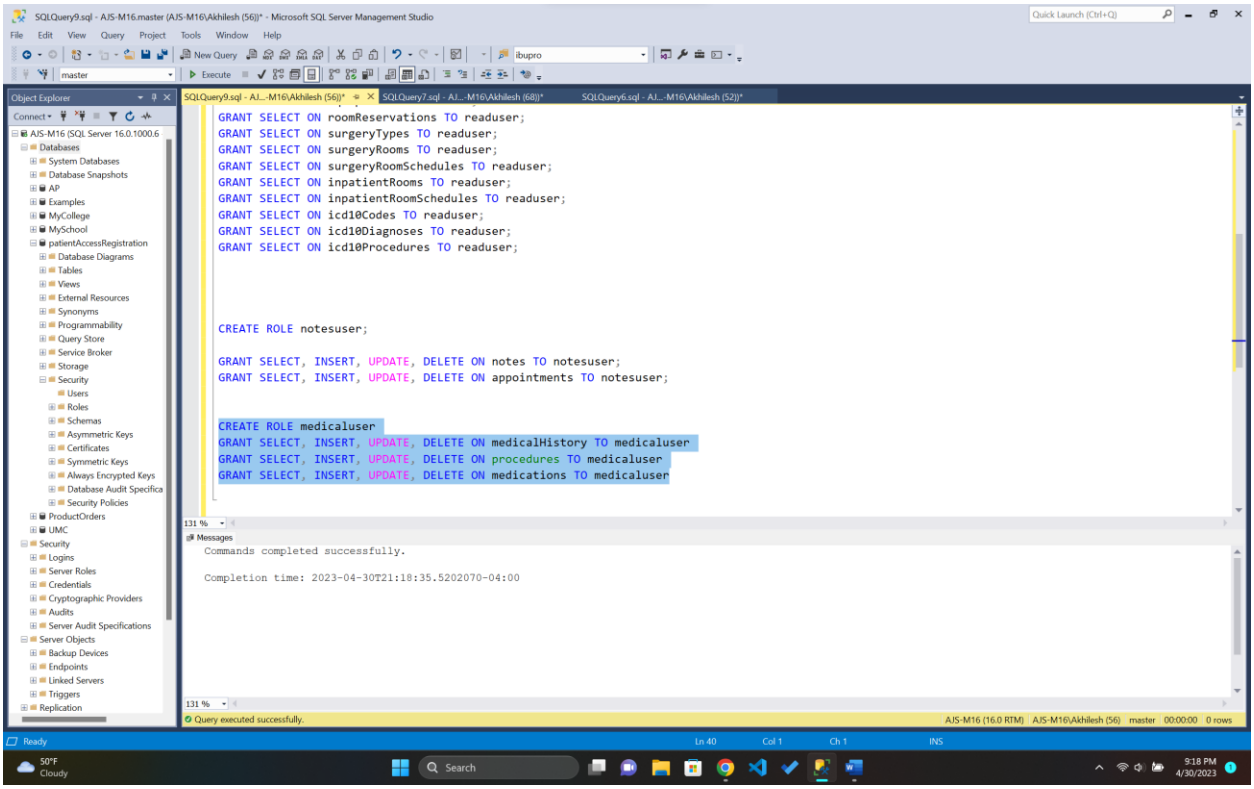
```
CREATE ROLE notesuser;

GRANT SELECT, INSERT, UPDATE, DELETE ON notes TO notesuser;
GRANT SELECT, INSERT, UPDATE, DELETE ON appointments TO notesuser;
```



3. User 3

Source Code: -

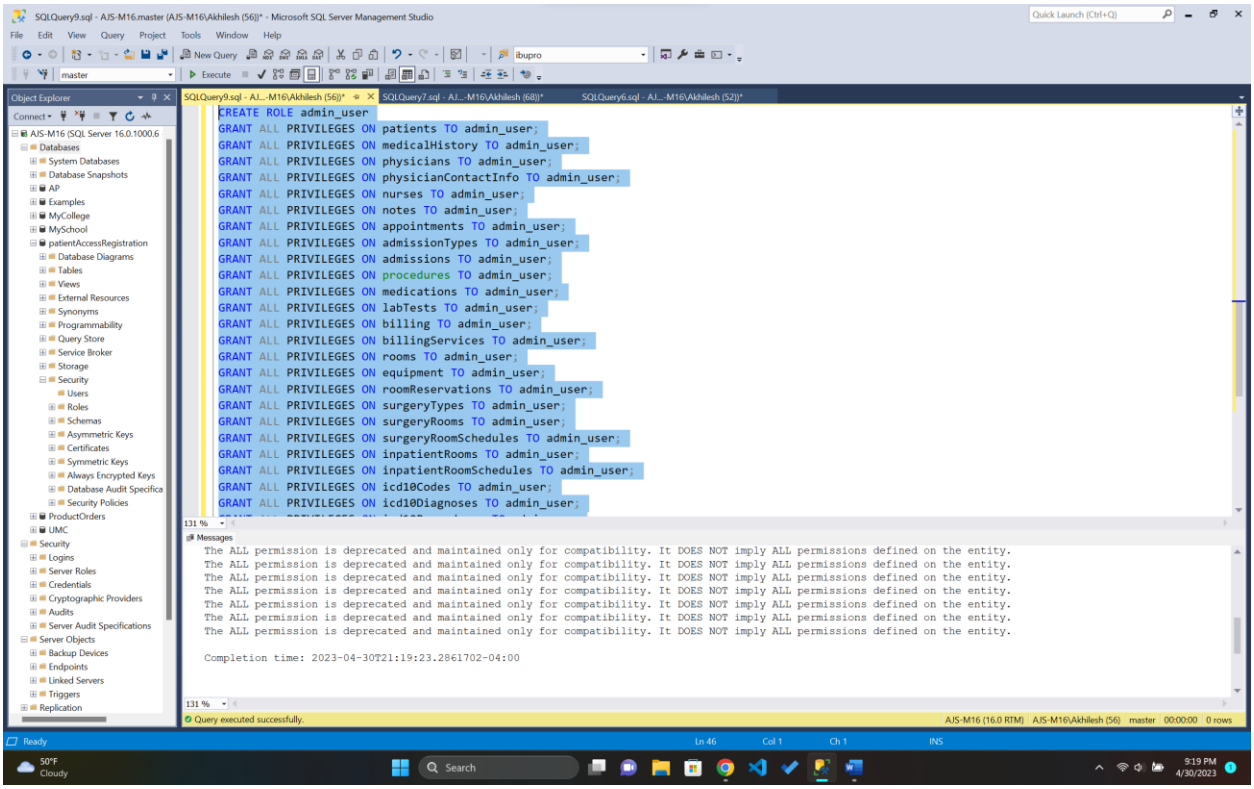
```
CREATE ROLE medicaluser
GRANT SELECT, INSERT, UPDATE, DELETE ON medicalHistory TO medicaluser
GRANT SELECT, INSERT, UPDATE, DELETE ON procedures TO medicaluser
GRANT SELECT, INSERT, UPDATE, DELETE ON medications TO medicaluser
```



4. User 4

Source Code: -

```
CREATE ROLE admin_user
GRANT ALL PRIVILEGES ON patients TO admin_user;
GRANT ALL PRIVILEGES ON medicalHistory TO admin_user;
GRANT ALL PRIVILEGES ON physicians TO admin_user;
GRANT ALL PRIVILEGES ON physicianContactInfo TO admin_user;
GRANT ALL PRIVILEGES ON nurses TO admin_user;
GRANT ALL PRIVILEGES ON notes TO admin_user;
GRANT ALL PRIVILEGES ON appointments TO admin_user;
GRANT ALL PRIVILEGES ON admissionTypes TO admin_user;
GRANT ALL PRIVILEGES ON admissions TO admin_user;
GRANT ALL PRIVILEGES ON procedures TO admin_user;
GRANT ALL PRIVILEGES ON medications TO admin_user;
GRANT ALL PRIVILEGES ON labTests TO admin_user;
GRANT ALL PRIVILEGES ON billing TO admin_user;
GRANT ALL PRIVILEGES ON billingServices TO admin_user;
GRANT ALL PRIVILEGES ON rooms TO admin_user;
GRANT ALL PRIVILEGES ON equipment TO admin_user;
GRANT ALL PRIVILEGES ON roomReservations TO admin_user;
GRANT ALL PRIVILEGES ON surgeryTypes TO admin_user;
GRANT ALL PRIVILEGES ON surgeryRooms TO admin_user;
GRANT ALL PRIVILEGES ON surgeryRoomSchedules TO admin_user;
GRANT ALL PRIVILEGES ON inpatientRooms TO admin_user;
GRANT ALL PRIVILEGES ON inpatientRoomSchedules TO admin_user;
GRANT ALL PRIVILEGES ON icd10Codes TO admin_user;
GRANT ALL PRIVILEGES ON icd10Diagnoses TO admin_user;
GRANT ALL PRIVILEGES ON icd10Procedures TO admin_user;
```



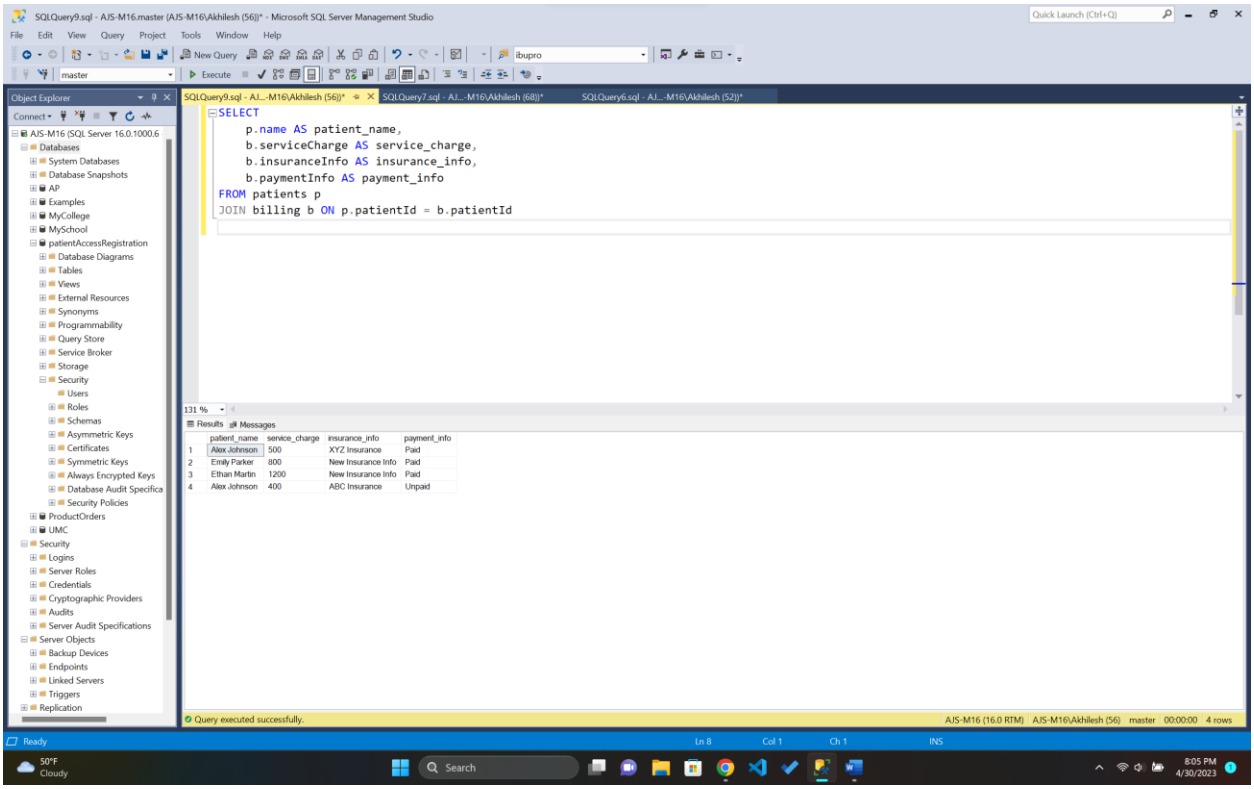
Business reports:

1. Patient Billing report:

This report provides an overview of the billing information for patients, including service charges, insurance information, and payment details.

Source Code: -

```
SELECT
    p.name AS patient_name,
    b.serviceCharge AS service_charge,
    b.insuranceInfo AS insurance_info,
    b.paymentInfo AS payment_info
FROM patients p
JOIN billing b ON p.patientId = b.patientId
```

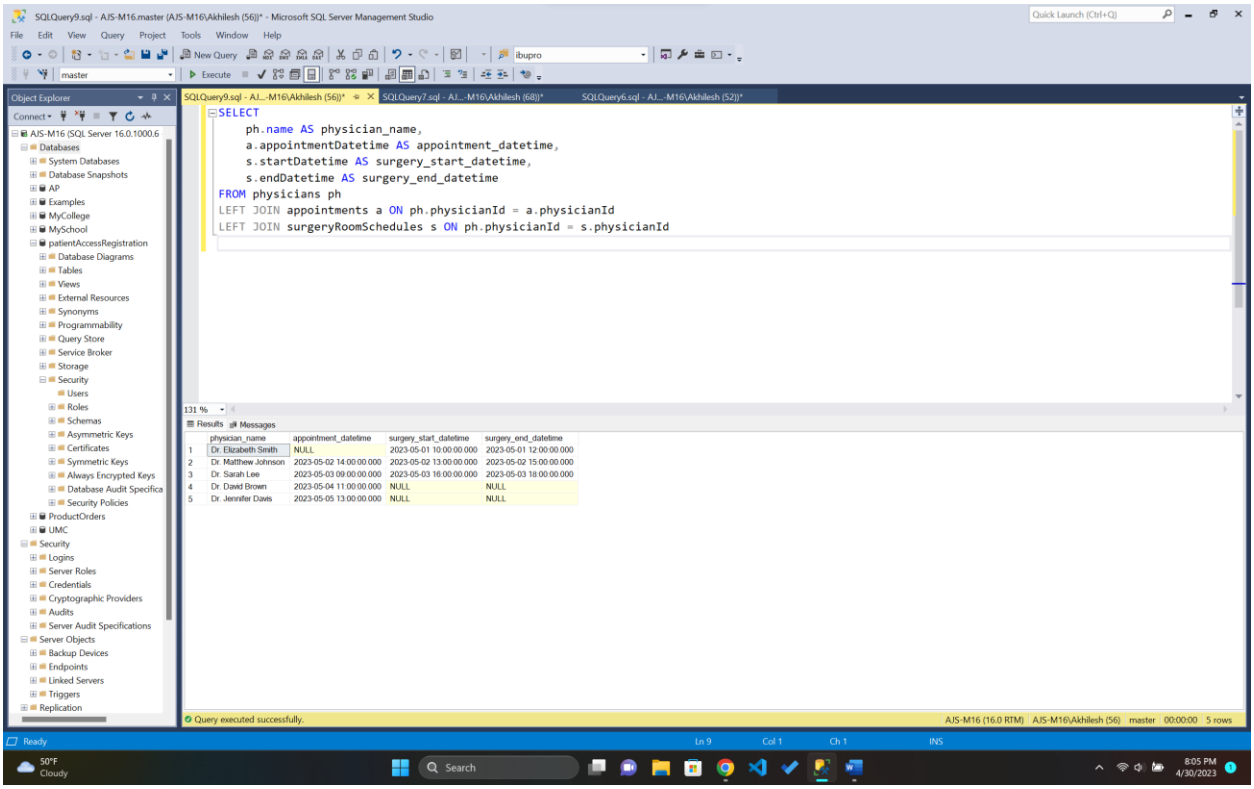


2. Physician Schedule report

This report provides a schedule of the appointments and surgeries that physicians have scheduled.

Source Code: -

```
SELECT
    ph.name AS physician_name,
    a.appointmentDatetime AS appointment_datetime,
    s.startDatetime AS surgery_start_datetime,
    s.endDatetime AS surgery_end_datetime
FROM physicians ph
LEFT JOIN appointments a ON ph.physicianId = a.physicianId
LEFT JOIN surgeryRoomSchedules s ON ph.physicianId = s.physicianId
```

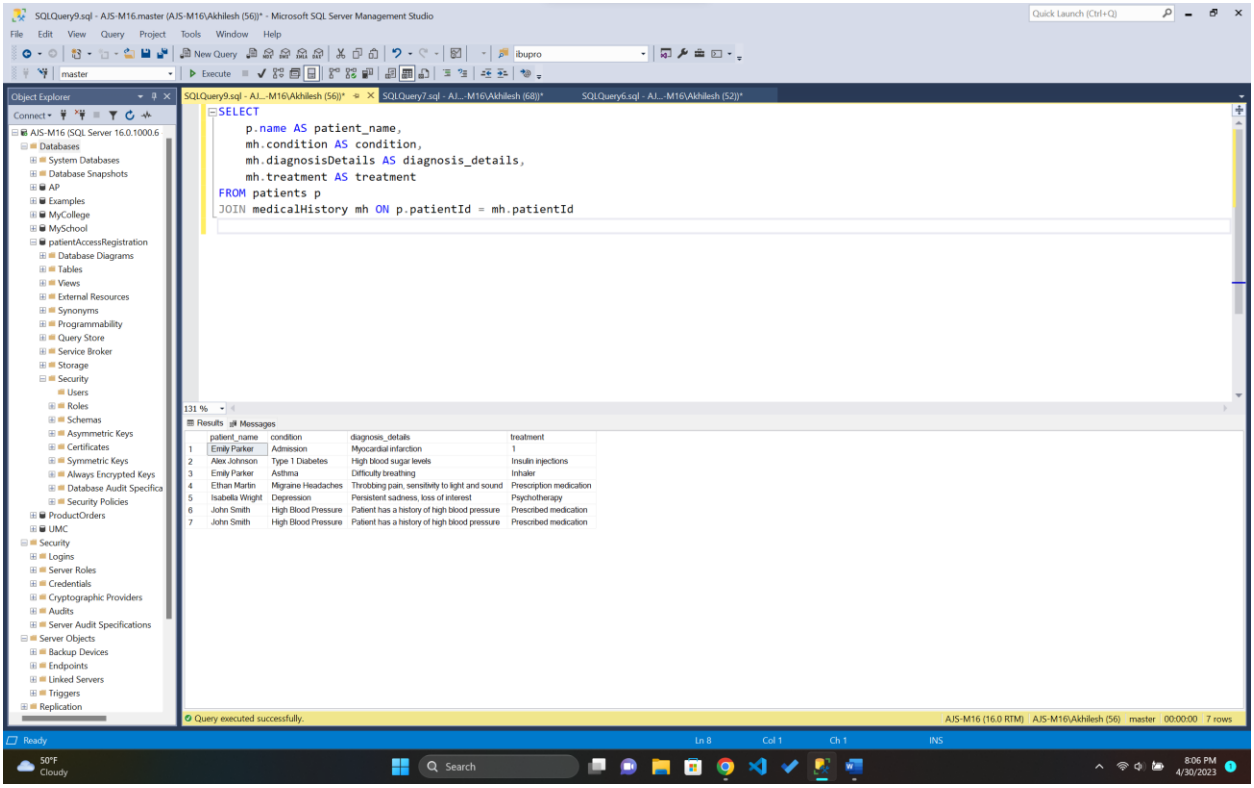


3. Medical History report:

This report provides a comprehensive medical history for each patient, including their conditions, diagnoses, and treatments.

Source Code: -

```
SELECT
    p.name AS patient_name,
    mh.condition AS condition,
    mh.diagnosisDetails AS diagnosis_details,
    mh.treatment AS treatment
FROM patients p
JOIN medicalHistory mh ON p.patientId = mh.patientId
```

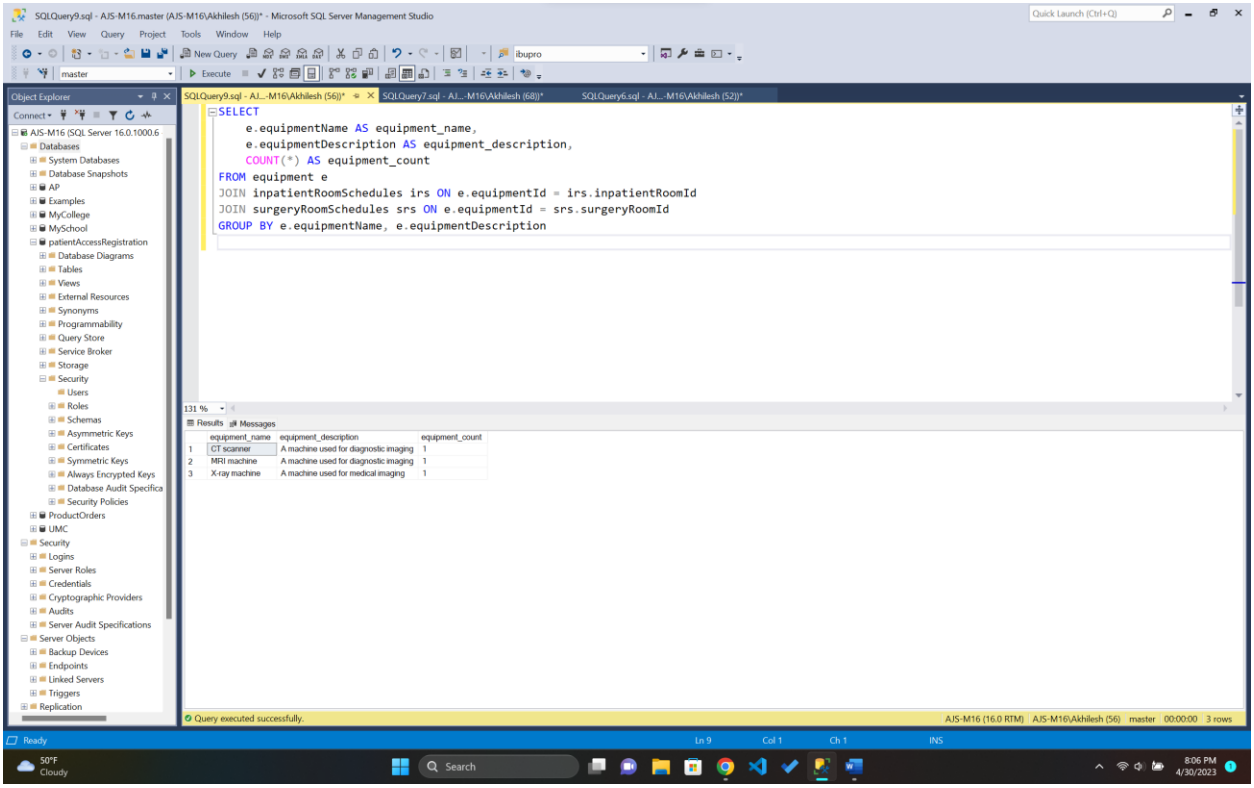


4. Equipment Inventory report:

This report provides an inventory of the equipment that is currently available, including its name and description, and the number of each type of equipment.

Source Code: -

```
SELECT
    e.equipmentName AS equipment_name,
    e.equipmentDescription AS equipment_description,
    COUNT(*) AS equipment_count
FROM equipment e
JOIN inpatientRoomSchedules irs ON e.equipmentId = irs.inpatientRoomId
JOIN surgeryRoomSchedules srs ON e.equipmentId = srs.surgeryRoomId
GROUP BY e.equipmentName, e.equipmentDescription
```



Conclusion:

- Through this project, I gained knowledge on how to plan, execute, and test a database for a patient access registration.
- The project was primarily broken down into three parts, which included constructing the database using an E/R diagram and coming to conclusions from it.
- Tables, columns, primary keys, datatypes, nullabilities, and relationships were all decided upon during the implementation process.
- During the testing phase, various business logics as well as performance and efficiency upgrades were implemented.

References:

1. W3School: https://www.w3schools.com/
2. Diagram.IO: https://dbdiagram.io/home/
3. Microsoft SQL server: https://www.microsoft.com/en-us/sql-server/sql-server-2019