Hospital Management System

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Base de Dados - P10/G10

Bernardo Marujo, Paulo Macedo
(107322) bernardomarujo@ua.pt, (102620) paulomacedo@ua.pt

June 4, 2024

Abstract

This report details the development and implementation of a hospital management system aimed at streamlining the processes of billing and prescription management. The system integrates various functionalities to provide an efficient, user-friendly platform for managing patient data, appointments, billing, and prescriptions. This report covers the system's design, key features, database integration, and the usage of custom SQL functions to enhance functionality.

Introduction

In the rapidly evolving field of healthcare, efficient management of patient information, billing, and prescriptions is crucial to providing high-quality service. The advent of digital solutions has significantly transformed the healthcare industry, offering new ways to manage and process information seamlessly. This project focuses on developing a comprehensive hospital management system designed to facilitate these essential functions.

The primary objective of this system is to automate and streamline the billing and prescription processes within a hospital setting. By doing so, it aims to reduce administrative burden, minimize errors, and enhance the overall efficiency of healthcare delivery. The system integrates several key functionalities, including patient and doctor management, appointment scheduling, and bill generation, all within a cohesive framework.

Key components of the system include:

- **Patient Management:** A module to handle patient registration, personal details, and insurance information.
- **Appointment Scheduling:** A feature to facilitate the scheduling and management of patient appointments.
- **Billing System:** A comprehensive billing module that calculates fees, generates bills, and updates payment status.
- **Prescription Management:** A module to manage and record patient prescriptions, ensuring accurate and timely dispensing of medications.

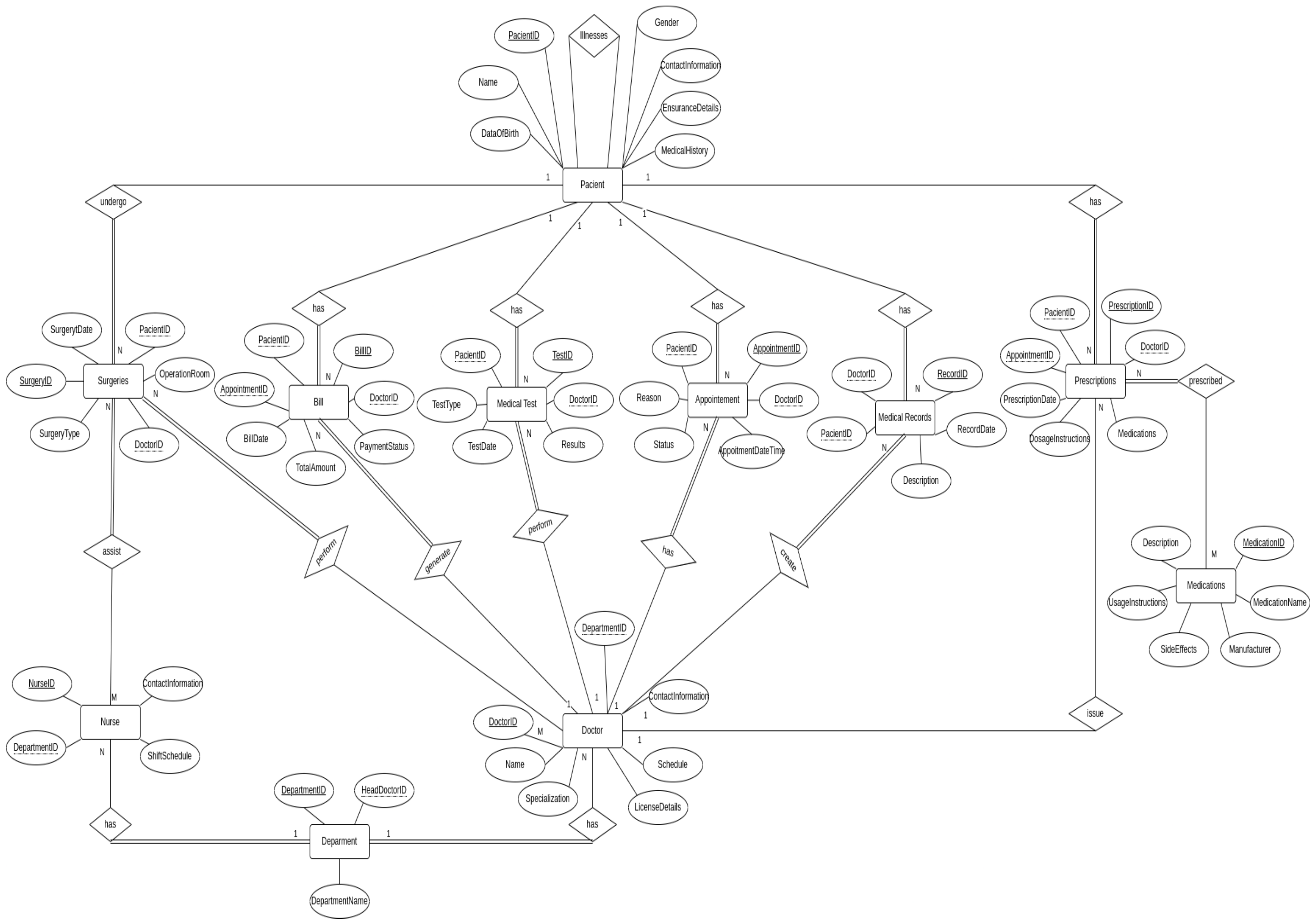
After running the project will appear the initial form which includes default values that are already correct for the database connection.

Análise de Requisitos

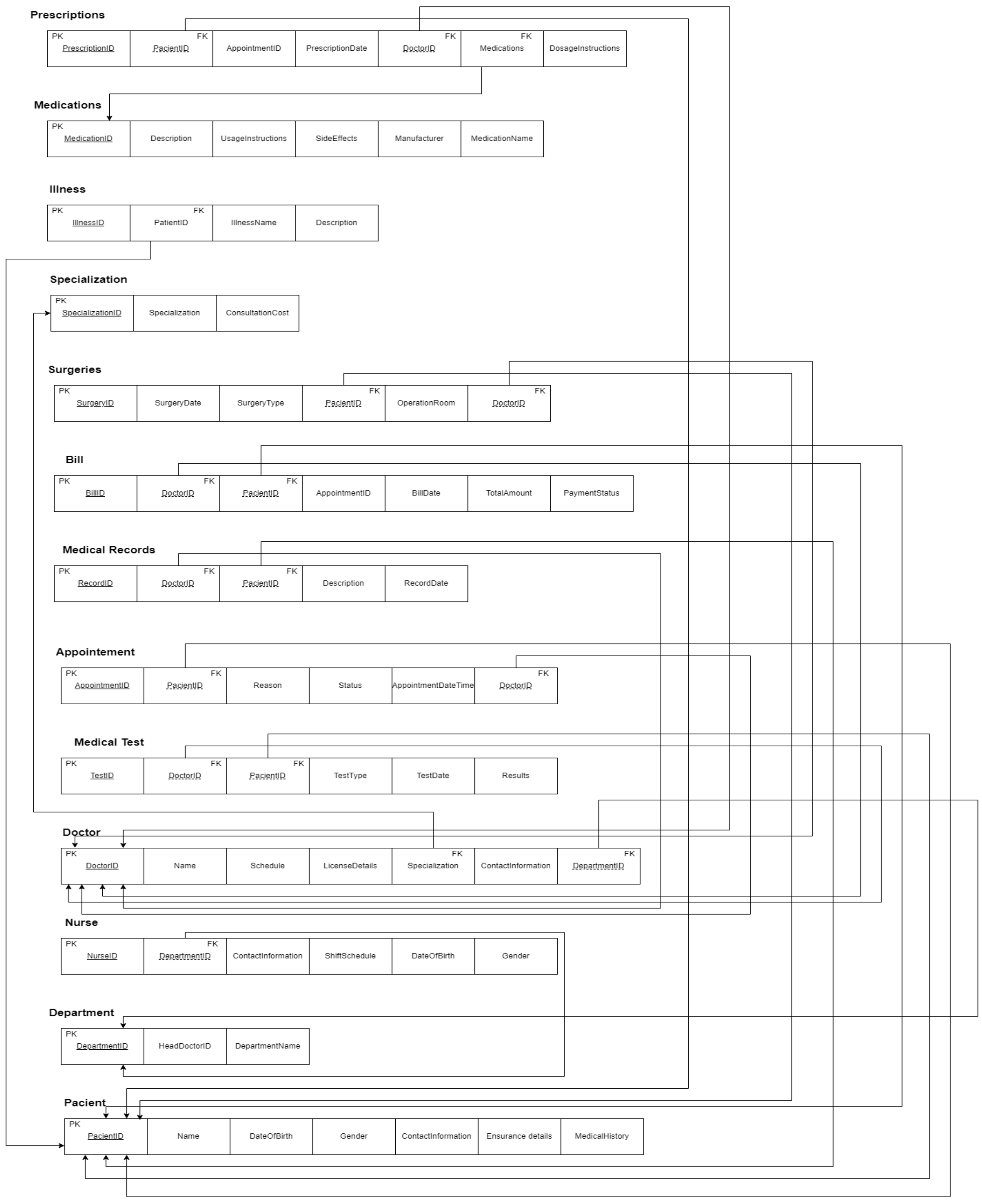
Functional Requirements

1. **Patient Registration**
   - The system should allow the registration of patients, including information such as name, date of birth, gender, contact information, medical history and insurance.
2. **Patient Management**
   - The system should allow the modification and updating of patient information.
3. **Appointment Scheduling and Management**
   - The system should allow scheduling appointments on a specific date for a particular patient with a doctor.
   - The system should allow viewing and managing pending appointments.
4. **Saving Consultation Details**
   - The system should allow recording the outcome of a specific consultation.
5. **Medical Prescriptions**
   - Doctors should be able to prescribe medications to patients.
   - The system should associate prescriptions with the corresponding consultations.
6. **Performing Medical Examinations**
   - Doctors should be able to order medical examinations for patients, specifying the type of test needed.
   - The system should record test results and any relevant observations.
7. **Billing**
   - The system should generate invoices for patients based on the consultations and the respective doctors.
   - It should be possible to pay the bills.

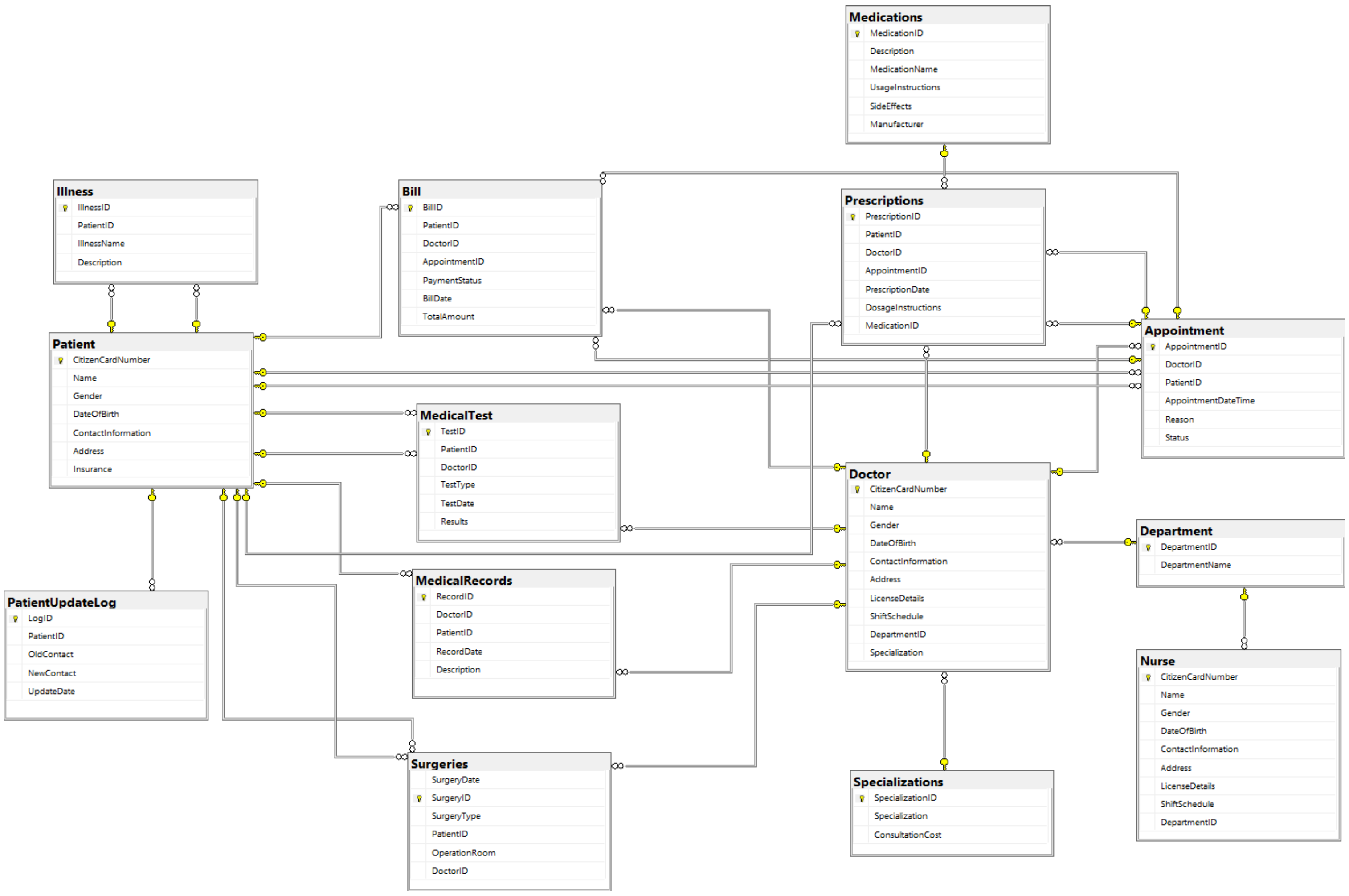
DER


PacientID
Illnesses
Gender
Name
ContactInformation
EnsuranceDetails
DataOfBirth
MedicalHistory
Pacient
undergo
has
has
has
has
has
Surgeries
SurgerytDate
PacientID
SurgeryID
OperationRoom
SurgeryType
DoctorID
Bill
PacientID
BillID
AppointmentID
DoctorID
BillDate
PaymentStatus
TotalAmount
Medical Test
PacientID
TestID
TestType
DoctorID
TestDate
Results
Appointement
PacientID
AppointmentID
Reason
DoctorID
Status
AppoitmentDateTime
Medical Records
DoctorID
RecordID
PacientID
RecordDate
Description
Prescriptions
PacientID
PrescriptionID
AppointmentID
DoctorID
PrescriptionDate
DosageInstructions
Medications
prescribed
Medications
Description
MedicationID
UsageInstructions
MedicationName
SideEffects
Manufacturer
assist
perform
generate
perform
has
create
issue
Nurse
NurseID
ContactInformation
DepartmentID
ShiftSchedule
Doctor
DepartmentID
ContactInformation
DoctorID
Name
Schedule
Specialization
LicenseDetails
has
has
Deparment
DepartmentID
HeadDoctorID
DepartmentName

Relational Schema



Entity Relation



Database Schema Definition

The database schema for the hospital management system is designed to support the comprehensive management of patients, doctors, appointments, billing, and medical records. The schema consists of multiple tables, each serving a specific purpose, interconnected through foreign keys to maintain referential integrity. The following section details the structure and relationships defined in the database using SQL Data Definition Language (DDL) statements.

SQL Data Definition Language (DDL)

The DDL scripts provided below create the necessary tables and establish relationships among them to ensure the integrity and consistency of the data stored in the hospital management system.

Table Structures and Relationships

The database schema is designed to capture comprehensive information about patients, doctors, medical tests, prescriptions, and more. Below, we discuss the purpose of each table and the relationships between them.

Patient Table

The `Patient` table stores information about patients, including their citizen card number, name, gender, date of birth, contact information, address, and insurance status. This table serves as a central reference point for various other tables.

```sql
-- Patient table
CREATE TABLE Patient (
    CitizenCardNumber VARCHAR(255) PRIMARY KEY,
    Name VARCHAR(255),
    Gender CHAR(1),
    DateOfBirth DATE,
    ContactInformation VARCHAR(255),
    Address TEXT,
    Insurance BIT
);
```

Illness Table

The `Illness` table documents illnesses diagnosed in patients, including descriptions. It references the `Patient` table to associate each illness with a specific patient.

```sql
CREATE TABLE Illness (
    IllnessID INT PRIMARY KEY IDENTITY(1,1),
    PatientID VARCHAR(255),
    IllnessName VARCHAR(255),
    Description TEXT,
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
);
```

Doctor Table

The `Doctor` table maintains information about doctors, including their specializations and department affiliations. It references both the `Specializations` and `Department` tables.

```sql
-- Doctor table
CREATE TABLE Doctor (
    CitizenCardNumber VARCHAR(255) PRIMARY KEY,
    Name VARCHAR(255),
    Gender CHAR(1),
    DateOfBirth DATE,
    ContactInformation VARCHAR(255),
    Address TEXT,
    LicenseDetails VARCHAR(255),
    ShiftSchedule TEXT,
    DepartmentID INT NULL,
    Specialization INT,
    FOREIGN KEY (Specialization) REFERENCES Specializations(SpecializationID),
    FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);
```

Surgeries Table

The `Surgeries` table records surgeries performed, including details about the surgery type, patient, doctor, and operation room. It references the `Patient` and `Doctor` tables.

```sql
-- Surgeries table
CREATE TABLE Surgeries (
    SurgeryDate DATE,
    SurgeryID INT PRIMARY KEY IDENTITY(1,1),
    SurgeryType VARCHAR(255),
    PatientID VARCHAR(255),
    OperationRoom VARCHAR(255),
    DoctorID VARCHAR(255),
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber),
    FOREIGN KEY (DoctorID) REFERENCES Doctor(CitizenCardNumber)
);
```

Bill Table

The `Bill` table manages billing information, linking to patients, doctors, and appointments to generate and track payments. It references the `Patient`, `Doctor`, and `Appointment` tables.

```sql
-- Bill table
CREATE TABLE Bill (
    BillID INT PRIMARY KEY IDENTITY(1,1),
    PatientID VARCHAR(255),
    DoctorID VARCHAR(255),
    AppointmentID INT,
    PaymentStatus VARCHAR(100),
    BillDate DATE,
    TotalAmount DECIMAL(10, 2),
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber),
    FOREIGN KEY (DoctorID) REFERENCES Doctor(CitizenCardNumber),
    FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID)
);
```

MedicalRecords Table

The `MedicalRecords` table maintains records of medical interactions, referencing both doctors and patients to document various medical events and treatments.

```sql
-- Medical Records table
CREATE TABLE MedicalRecords (
    RecordID INT PRIMARY KEY IDENTITY(1,1),
    DoctorID VARCHAR(255),
    PatientID VARCHAR(255),
    RecordDate DATE,
    Description TEXT,
    FOREIGN KEY (DoctorID) REFERENCES Doctor(CitizenCardNumber),
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
);
```

Medications Table

The `Medications` table lists medications, their usage instructions, side effects, and manufacturers.

```sql
-- Medications table
CREATE TABLE Medications (
    MedicationID INT PRIMARY KEY identity(1,1),
    Description TEXT,
    MedicationName VARCHAR(255),
    UsageInstructions TEXT,
    SideEffects TEXT,
    Manufacturer VARCHAR(255)
);
```

Specializations Table

The `Specializations` table records different medical specializations and their corresponding consultation costs. This table is linked to the `Doctor` table to specify each doctor's area of expertise.

```sql
CREATE TABLE Specializations (
    SpecializationID INT PRIMARY KEY IDENTITY(1,1),
    Specialization VARCHAR(255),
    ConsultationCost DECIMAL(10,2)
);
```

Department Table

The `Department` table captures information about various hospital departments. It is referenced by both the `Doctor` and `Nurse` tables to associate medical staff with their respective departments.

```sql
-- Department table
CREATE TABLE Department (
    DepartmentID INT PRIMARY KEY IDENTITY(1,1),
    DepartmentName VARCHAR(255)
);
```

Nurse Table

The `Nurse` table contains information about nurses, including their license details and shift schedules. It links to the `Department` table to indicate the department each nurse is assigned to.

```sql
-- Nurse table
CREATE TABLE Nurse (
    CitizenCardNumber VARCHAR(255) PRIMARY KEY,
    Name VARCHAR(255),
    Gender CHAR(1),
    DateOfBirth DATE,
    ContactInformation VARCHAR(255),
    Address TEXT,
    LicenseDetails VARCHAR(255),
    ShiftSchedule TEXT,
    DepartmentID INT NULL,
    FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);
```

Appointment Table

The `Appointment` table tracks appointments between patients and doctors. It references both the `Patient` and `Doctor` tables and captures details about the appointment's date, time, and status.

```sql
-- Appointment table
CREATE TABLE Appointment (
    AppointmentID INT PRIMARY KEY IDENTITY(1,1),
    DoctorID VARCHAR(255),
    PatientID VARCHAR(255),
    AppointmentDateTime DATETIME,
    Reason TEXT,
    Status VARCHAR(100),
    FOREIGN KEY (DoctorID) REFERENCES Doctor(CitizenCardNumber),
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
);
```

MedicalTest Table

The `MedicalTest` table stores information about medical tests, including test types, results, and associated patients and doctors. It references the `Patient` and `Doctor` tables.

```sql
-- Medical test table
CREATE TABLE MedicalTest (
    TestID INT PRIMARY KEY IDENTITY(1,1),
    PatientID VARCHAR(255),
    DoctorID VARCHAR(255),
    TestType VARCHAR(255),
    TestDate DATE,
    Results TEXT,
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber),
    FOREIGN KEY (DoctorID) REFERENCES Doctor(CitizenCardNumber)
);
```

Prescriptions Table

The `Prescriptions` table manages prescription details, linking patients, doctors, appointments, and medications. It references the `Patient`, `Doctor`, `Appointment`, and `Medications` tables.

```sql
-- Prescriptions table
CREATE TABLE Prescriptions (
    PrescriptionID INT PRIMARY KEY IDENTITY(1,1),
    PatientID VARCHAR(255),
    DoctorID VARCHAR(255),
    AppointmentID INT,
    PrescriptionDate DATE,
    DosageInstructions TEXT,
    MedicationID INT,
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber),
    FOREIGN KEY (DoctorID) REFERENCES Doctor(CitizenCardNumber),
    FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID),
    FOREIGN KEY (MedicationID) REFERENCES Medications(MedicationID)
);
```

Referential Integrity and Constraints

To ensure data consistency and integrity, the schema includes several foreign key constraints. These constraints enforce relationships between tables, ensuring that data remains consistent and valid across the database. For example, deleting a patient will cascade delete related records in the `Illness`, `Surgeries`, `Appointment`, `Bill`, `MedicalTest`, `MedicalRecords`, and `Prescriptions` tables.

```sql
ALTER TABLE Illness
ADD CONSTRAINT FK_Illness_Patient
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
    ON DELETE CASCADE;

ALTER TABLE Surgeries
ADD CONSTRAINT FK_Surgeries_Patient
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
    ON DELETE CASCADE;

ALTER TABLE Appointment
ADD CONSTRAINT FK_Appointment_Patient
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
    ON DELETE CASCADE;

ALTER TABLE Bill
ADD CONSTRAINT FK_Bill_Patient
    FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID)
    ON DELETE CASCADE;

ALTER TABLE MedicalTest
ADD CONSTRAINT FK_MedicalTest_Patient
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
    ON DELETE CASCADE;

ALTER TABLE MedicalRecords
ADD CONSTRAINT FK_MedicalRecords_Patient
    FOREIGN KEY (PatientID) REFERENCES Patient(CitizenCardNumber)
    ON DELETE CASCADE;

ALTER TABLE Prescriptions
ADD CONSTRAINT FK_Prescriptions_Patient
     FOREIGN KEY (AppointmentID) REFERENCES Appointment(AppointmentID)
    ON DELETE CASCADE;
```

Stored Procedures

The hospital management system's database includes a series of stored procedures to facilitate efficient and consistent execution of common tasks. These stored procedures enhance the database's functionality by encapsulating complex operations into reusable SQL code. The following section describes each stored procedure and its purpose.

Patient Management Procedures

- **AddPatient:** This procedure inserts a new patient record into the `Patient` table. It takes parameters such as `CitizenCardNumber`, `Name`, `Gender`, `DateOfBirth`, `ContactInformation`, `Address`, and `Insurance`.
- **DeletePatient:** This procedure deletes a patient record based on the provided `CitizenCardNumber`. This ensures that all related records are also deleted due to cascade delete constraints.
- **UpdatePatientContact:** This procedure updates a patient's contact information, address, and insurance status using the `CitizenCardNumber` as the identifier.

```sql
CREATE PROCEDURE AddPatient
    @CitizenCardNumber VARCHAR(255),
    @Name VARCHAR(255),
    @Gender CHAR(1),
    @DateOfBirth DATE,
    @ContactInformation VARCHAR(255),
    @Address TEXT,
    @Insurance BIT
AS
BEGIN
    INSERT INTO Patient (CitizenCardNumber, Name, Gender, DateOfBirth,
   ContactInformation, Address, Insurance)
    VALUES (@CitizenCardNumber, @Name, @Gender, @DateOfBirth, @ContactInformation,
   @Address, @Insurance);
END
GO
```

```sql
CREATE PROCEDURE DeletePatient
    @CitizenCardNumber VARCHAR(255)
AS
BEGIN
    DELETE FROM Patient WHERE CitizenCardNumber = @CitizenCardNumber;
END
GO
```

```sql
CREATE PROCEDURE UpdatePatientContact
    @CitizenCardNumber VARCHAR(255),
    @NewContactInformation VARCHAR(255),
    @NewAddress TEXT,
    @NewInsurance BIT
AS
BEGIN
    UPDATE Patient
    SET ContactInformation = @NewContactInformation,
        Address = @NewAddress,
        Insurance = @NewInsurance
    WHERE CitizenCardNumber = @CitizenCardNumber;
END
GO
```

Appointment Management Procedures

- **ScheduleAppointment:** This procedure schedules a new appointment between a doctor and a patient, specifying the appointment date, reason, and status.
- **UpdateAppointment:** This procedure updates the date and status of an existing appointment based on the `AppointmentID`.
- **CancelAppointment:** This procedure deletes an appointment using the `AppointmentID`.

```sql
CREATE PROCEDURE ScheduleAppointment
    @DoctorID VARCHAR(255),
    @PatientID VARCHAR(255),
    @AppointmentDateTime DATETIME,
    @Reason TEXT,
    @Status VARCHAR(100)
AS
BEGIN
    INSERT INTO Appointment (DoctorID, PatientID, AppointmentDateTime, Reason,
    Status)
    VALUES (@DoctorID, @PatientID, @AppointmentDateTime, @Reason, @Status);
END

GO
```

```sql
CREATE PROCEDURE UpdateAppointment
    @AppointmentID INT,
    @NewAppointmentDateTime DATETIME,
    @NewStatus VARCHAR(100)
AS
BEGIN
    -- Update the appointment details in the Appointment table
    UPDATE Appointment
    SET AppointmentDateTime = @NewAppointmentDateTime,
        Status = @NewStatus
    WHERE AppointmentID = @AppointmentID;
END
GO
```

```sql
CREATE PROCEDURE CancelAppointment
    @AppointmentID INT
AS
BEGIN
    -- Delete the appointment from the Appointment table
    DELETE FROM Appointment
    WHERE AppointmentID = @AppointmentID;
END
GO
```

Medical Records and Tests Procedures

- **RecordMedicalTest:** This procedure records a new medical test, including the test type, date, and results, linked to a specific patient and doctor.
- **RecordTreatment:** This procedure inserts a new medical record describing a treatment or medical interaction between a doctor and a patient.

```sql
CREATE PROCEDURE RecordMedicalTest
    @PatientID VARCHAR(255),
    @DoctorID VARCHAR(255),
    @TestType VARCHAR(255),
    @TestDate DATE,
    @Results TEXT
AS
BEGIN
    INSERT INTO MedicalTest (PatientID, DoctorID, TestType, TestDate, Results)
    VALUES (@PatientID, @DoctorID, @TestType, @TestDate, @Results);
END
GO
```

```sql
-- Record treatment outcome
CREATE PROCEDURE RecordTreatment
    @DoctorID VARCHAR(255),
    @PatientID VARCHAR(255),
    @Description TEXT,
    @RecordDate DATE
AS
BEGIN
    INSERT INTO MedicalRecords (DoctorID, PatientID, Description, RecordDate)
    VALUES (@DoctorID, @PatientID, @Description, @RecordDate);
END
GO
```

Surgery Management Procedures

- **CompleteSurgery:** This procedure records details of a completed surgery, including the date, patient, doctor, surgery type, and operation room.

```sql
CREATE PROCEDURE CompleteSurgery
    @SurgeryDate DATE,
    @PatientID VARCHAR(255),
    @DoctorID VARCHAR(255),
    @SurgeryType VARCHAR(255),
    @OperationRoom VARCHAR(255)
AS
BEGIN
    INSERT INTO Surgeries (SurgeryDate, PatientID, DoctorID, SurgeryType,
   OperationRoom)
    VALUES (@SurgeryDate, @PatientID, @DoctorID, @SurgeryType, @OperationRoom);
END
GO
```

Billing and Payment Procedures

- **ProcessPayment:** This procedure updates the payment status of a bill based on the `BillID`.
- **UpdateBillPaymentStatus:** This procedure marks a specific bill as paid using the `BillID`.

```sql
CREATE PROCEDURE ProcessPayment
    @BillID INT,
    @PaymentStatus VARCHAR(100)
AS
BEGIN
    UPDATE Bill
    SET PaymentStatus = @PaymentStatus
    WHERE BillID = @BillID;
END
GO
```

```sql
CREATE PROCEDURE UpdateBillPaymentStatus
    @BillID INT
AS
BEGIN
    -- Update the payment status of the specified bill
    UPDATE Bill
    SET PaymentStatus = 'Paid'
    WHERE BillID = @BillID;
END
GO
```

Prescription Management Procedures

- **PrescribeMedication:** This procedure creates a new prescription for a patient, detailing the medication, dosage instructions, and related appointment.
- **ListPrescriptionsByPatient:** This procedure lists all prescriptions for a specific patient, including details about the medication and dosage instructions.

```sql
-- Prescribe medication

CREATE PROCEDURE PrescribeMedication
    @PatientID VARCHAR(255),
    @DoctorID VARCHAR(255),
    @AppointmentID INT,
    @MedicationID INT,
    @PrescriptionDate DATE,
    @DosageInstructions TEXT
AS
BEGIN
    INSERT INTO Prescriptions (PatientID, DoctorID, AppointmentID, MedicationID,
   PrescriptionDate, DosageInstructions)
    VALUES (@PatientID, @DoctorID, @AppointmentID, @MedicationID, @PrescriptionDate
   , @DosageInstructions);
END
GO
```

```sql
CREATE PROCEDURE ListPrescriptionsByPatient
    @PatientID VARCHAR(255)
AS
BEGIN
    SET NOCOUNT ON;

    -- Query to list all prescriptions for a specific patient
    SELECT
        p.PrescriptionID,
        p.PatientID,
        p.DoctorID,
        p.AppointmentID,
        p.PrescriptionDate,
        p.DosageInstructions,
        p.MedicationID,
        m.MedicationName,
        m.Description AS MedicationDescription,
        m.SideEffects
    FROM
        Prescriptions p
        INNER JOIN Medications m ON p.MedicationID = m.MedicationID
    WHERE
        p.PatientID = @PatientID
    ORDER BY
        p.PrescriptionDate DESC;
END
GO
```

Query Procedures

- **GetPatientsByIllness:** This procedure retrieves a list of patients diagnosed with a specific illness, including their contact details and insurance status.
- **ListAllIllnesses:** This procedure lists all distinct illnesses recorded in the database.
- **GetAppointmentsByCitizenCardNumber:** This procedure retrieves all appointments associated with a specific citizen card number, whether the individual is a doctor or a patient.

```sql
CREATE PROCEDURE GetPatientsByIllness
    @IllnessName VARCHAR(255)
AS
BEGIN
    SELECT
    p.CitizenCardNumber,
    p.Name,
    p.Gender,
    dbo.GetAge(p.DateOfBirth) AS Age,
    p.ContactInformation,
    p.Address,
    p.Insurance
    FROM Patient p
    JOIN Illness i ON p.CitizenCardNumber = i.PatientID
    WHERE i.IllnessName = @IllnessName
    ORDER BY p.Name;
END
GO
```

```sql
CREATE PROCEDURE ListAllIllnesses
AS
BEGIN
    -- Selects all illness records
    SELECT DISTINCT IllnessName
    FROM Illness;
END
GO
```

```sql
CREATE PROCEDURE GetAppointmentsByCitizenCardNumber
    @CitizenCardNumber VARCHAR(255)
AS
BEGIN
    SELECT
        AppointmentID,
        DoctorID,
        PatientID,
        AppointmentDateTime,
        Reason,
        Status
    FROM
        Appointment
    WHERE
        DoctorID = @CitizenCardNumber OR
        PatientID = @CitizenCardNumber
    ORDER BY
        AppointmentDateTime;
END
GO
```

Database Triggers

Triggers in the hospital management system's database automate specific actions in response to events on particular tables. This section outlines the triggers implemented and their functionalities, enhancing data integrity, automating billing processes, and maintaining logs of critical changes.

Trigger: LogPatientUpdates

```sql
CREATE TRIGGER LogPatientUpdates
ON Patient
AFTER UPDATE
AS
BEGIN
    IF UPDATE(ContactInformation)
    BEGIN
        INSERT INTO PatientUpdateLog(PatientID, OldContact, NewContact, UpdateDate)
        SELECT i.CitizenCardNumber, d.ContactInformation, i.ContactInformation,
    GETDATE()
        FROM inserted i
        JOIN deleted d ON i.CitizenCardNumber = d.CitizenCardNumber;
    END
END;
GO
```

This trigger logs changes to patient contact information. Whenever the `ContactInformation` field in the `Patient` table is updated, this trigger captures the old and new contact information along with the update date, and inserts this data into the `PatientUpdateLog` table. This ensures an audit trail of changes to patient contact details, facilitating better monitoring and traceability.

Trigger: CheckAppointmentConflict

```sql
CREATE TRIGGER CheckAppointmentConflict
ON Appointment
INSTEAD OF INSERT
AS
BEGIN
    IF EXISTS (
        SELECT 1 FROM Appointment
        WHERE DoctorID IN (SELECT DoctorID FROM inserted)
        AND AppointmentDateTime IN (SELECT AppointmentDateTime FROM inserted)
    )
    BEGIN
        RAISERROR ('This doctor has another appointment at the same time', 16, 1);
    END
    ELSE
    BEGIN
        INSERT INTO Appointment (AppointmentDateTime, DoctorID, PatientID, Reason,
    Status)
        SELECT AppointmentDateTime, DoctorID, PatientID, Reason, Status FROM
    inserted;
    END
END;
GO
```

This trigger prevents scheduling conflicts for doctors' appointments. When a new appointment is inserted, it checks if the doctor already has an appointment at the same date and time. If a conflict is detected, it raises an error and does not insert the new appointment. If no conflict exists, the appointment is inserted as intended. This ensures that no doctor is double-booked, maintaining the integrity of the appointment schedule.

Trigger: AutoGenerateBill

```sql
CREATE TRIGGER AutoGenerateBill
ON Appointment
AFTER INSERT
AS
BEGIN
    DECLARE @Fee DECIMAL(10,2);
    DECLARE @Specialization INT;

    SELECT @Specialization = d.Specialization
    FROM Doctor d
    JOIN inserted i ON i.DoctorID = d.CitizenCardNumber;

    SELECT @Fee = dbo.CalculateConsultationFee(i.PatientID, @Specialization)
    FROM inserted i;

    INSERT INTO Bill(PatientID, DoctorID, AppointmentID, PaymentStatus, BillDate,
    TotalAmount)
    SELECT PatientID, DoctorID, AppointmentID, 'Pending', GETDATE(), @Fee
    FROM inserted;
END;
GO
```

This trigger automatically generates a bill after a new appointment is created. Upon insertion of a new appointment, it determines the consultation fee based on the doctor's specialization and the patient's details using the `CalculateConsultationFee` function. It then inserts a new record into the `Bill` table with the appointment details and the calculated fee, setting the payment status to 'Pending'. This automation ensures timely billing and reduces manual errors in the billing process.

Database Functions

This section describes the user-defined functions (UDFs) implemented in the hospital management system's database. These functions provide essential calculations and checks, enhancing the system's ability to process data efficiently and accurately.

Function: GetAge

```sql
CREATE FUNCTION GetAge(@dob DATE)
RETURNS INT
AS
BEGIN
    -- Calculate age based on the date of birth and the current date
    DECLARE @age INT;
    SET @age = DATEDIFF(year, @dob, GETDATE());

    -- Adjust for cases where this year's birthday has not yet occurred
    IF (MONTH(@dob) > MONTH(GETDATE())) OR (MONTH(@dob) = MONTH(GETDATE()) AND DAY(@dob) > DAY(GETDATE()))
        SET @age = @age - 1;

    RETURN @age;
END;
GO
```

The `GetAge` function calculates a patient's age based on their date of birth. This function computes the difference in years between the current date and the date of birth. Additionally, it adjusts for cases where the current year's birthday has not yet occurred, ensuring an accurate age calculation.

Function: CalculateConsultationFee

```sql
CREATE FUNCTION CalculateConsultationFee
(
    @CitizenCardNumber VARCHAR(255),
    @Specialization VARCHAR(255)
)
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @BaseCost DECIMAL(10,2);
    DECLARE @DiscountFactor DECIMAL(10,2) = 0.60;  -- 40% discount, so pay 60%
    DECLARE @TotalCost DECIMAL(10,2);
    DECLARE @HasInsurance BIT;

    -- Determine if the patient has insurance
    SET @HasInsurance = dbo.CheckPatientInsurance(@CitizenCardNumber);

    -- Get base cost for the specialization
    SELECT @BaseCost = ConsultationCost
    FROM Specializations
    WHERE SpecializationID = @Specialization;

    -- Handle the case where no cost is found for the specialization
    IF @BaseCost IS NULL
        SET @BaseCost = 0;  -- Consider setting a default cost or handling this with an error

    -- Calculate total cost based on insurance
    IF @HasInsurance = 1
        SET @TotalCost = @BaseCost * @DiscountFactor;
    ELSE
        SET @TotalCost = @BaseCost;

    RETURN @TotalCost;
END;
GO
```

The `CalculateConsultationFee` function calculates the consultation fee for a patient based on their insurance status and the doctor's specialization. It first checks if the patient has insurance using the `CheckPatientInsurance` function. Then, it retrieves the base cost of the consultation from the `Specializations` table. If the patient has insurance, a discount is applied to the base cost. The function returns the total cost, ensuring that billing processes are accurate and consistent.

Function: CheckPatientInsurance

```
CREATE FUNCTION CheckPatientInsurance
(
    @CitizenCardNumber VARCHAR(255)
)
RETURNS BIT
AS
BEGIN
    DECLARE @HasInsurance BIT;

    -- Get insurance status directly from the Insurance column
    SELECT @HasInsurance = Insurance
    FROM Patient
    WHERE CitizenCardNumber = @CitizenCardNumber;

    -- Handle case where no such patient exists
    IF @HasInsurance IS NULL
        SET @HasInsurance = 0;

    RETURN @HasInsurance;
END;
GO
```

The `CheckPatientInsurance` function determines whether a patient has insurance coverage. It queries the `Patient` table to retrieve the insurance status based on the patient's citizen card number. If the patient exists and has insurance, the function returns 1 (true). If the patient does not exist or does not have insurance, it returns 0 (false). This function is crucial for calculating costs and determining eligibility for discounted rates.

Database Indexing

In order to enhance the performance and efficiency of our database operations, we have implemented several indices. Indices play a crucial role in speeding up the retrieval of records by reducing the amount of data that needs to be scanned. Below is a detailed description of the indices created for various tables in our hospital management system.

Patient Table Indices

```sql
-- Useful for queries filtering patients by age / date of birth and Name
CREATE INDEX idx_patient_name ON Patient(Name);
CREATE INDEX idx_patient_dateofbirth ON Patient(DateOfBirth);
```

The `idx_patient_name` and `idx_patient_dateofbirth` indices are created to facilitate quick lookups of patients based on their names and dates of birth. These indices are particularly useful for queries that filter patients by age or search for patients by name, improving the speed of these operations.

Illness Table Index

```sql
-- Facilitates quick lookups of illnesses associated with specific patients
CREATE INDEX idx_illness_patientid ON Illness(PatientID);
```

The `idx_illness_patientid` index helps in quickly retrieving illnesses associated with specific patients. This index is essential for queries that join the `Illness` table with the `Patient` table to fetch all illnesses for a given patient.

Nurse Table Index

```sql
-- Helps in quickly finding all nurses in a specific department
CREATE INDEX idx_nurse_departmentid ON Nurse(DepartmentID);
```

The `idx_nurse_departmentid` index is designed to speed up queries that filter nurses based on their department. This is useful for administrative tasks and for generating reports about staff distribution across departments.

Doctor Table Index

```sql
-- Useful for queries involving doctors in specific departments
CREATE INDEX idx_doctor_departmentid ON Doctor(DepartmentID);
```

The `idx_doctor_departmentid` index enhances the performance of queries that involve doctors in specific departments. This index is beneficial for managing doctor assignments and analyzing departmental resources.

Surgeries Table Indices

```sql
-- Enhances performance of queries involving surgeries specific to a doctor or
    patient
CREATE INDEX idx_surgeries_patientid ON Surgeries(PatientID);
CREATE INDEX idx_surgeries_doctorid ON Surgeries(DoctorID);
```

The indices `idx_surgeries_patientid` and `idx_surgeries_doctorid` are created to enhance the performance of queries that involve surgeries related to specific doctors or patients. These indices are critical for retrieving surgery histories and for surgical department management.

Appointment Table Indices

```sql
-- Helps in fetching appointments by doctor, patient, or date
CREATE INDEX idx_appointment_doctorid ON Appointment(DoctorID);
CREATE INDEX idx_appointment_patientid ON Appointment(PatientID);
CREATE INDEX idx_appointment_datetime ON Appointment(AppointmentDateTime);
```

The `idx_appointment_doctorid`, `idx_appointment_patientid`, and `idx_appointment_datetime` indices facilitate quick retrieval of appointments based on the doctor, patient, or appointment date and time. These indices are essential for scheduling and managing appointments efficiently.

Bill Table Indices

```sql
-- Useful for retrieving billing information by patient, doctor, or specific
    appointments
CREATE INDEX idx_bill_patientid ON Bill(PatientID);
CREATE INDEX idx_bill_doctorid ON Bill(DoctorID);
CREATE INDEX idx_bill_appointmentid ON Bill(AppointmentID);
CREATE INDEX idx_bill_paymentstatus ON Bill(PaymentStatus);
```

The indices `idx_bill_patientid`, `idx_bill_doctorid`, `idx_bill_appointmentid`, and `idx_bill_paymentstatus` are designed to improve the performance of queries related to billing information. These indices are useful for quickly fetching bills associated with specific patients, doctors, appointments, or payment statuses.

Medical Test Table Indices

```sql
-- Facilitates faster retrieval of test results related to specific doctors or
    patients
CREATE INDEX idx_medicaltest_patientid ON MedicalTest(PatientID);
CREATE INDEX idx_medicaltest_doctorid ON MedicalTest(DoctorID);
```

The `idx_medicaltest_patientid` and `idx_medicaltest_doctorid` indices facilitate faster retrieval of medical test results related to specific doctors or patients. These indices are critical for medical diagnostics and for tracking patient test histories.

Prescriptions Table Indices

```sql
-- Assists in quickly locating prescriptions tied to specific appointments,
    medications, doctors, or patients
CREATE INDEX idx_prescriptions_patientid ON Prescriptions(PatientID);
CREATE INDEX idx_prescriptions_doctorid ON Prescriptions(DoctorID);
CREATE INDEX idx_prescriptions_appointmentid ON Prescriptions(AppointmentID);
CREATE INDEX idx_prescriptions_medicationid ON Prescriptions(MedicationID);
```

The indices `idx_prescriptions_patientid`, `idx_prescriptions_doctorid`, `idx_prescriptions_appointmentid`, and `idx_prescriptions_medicationid` assist in quickly locating prescriptions tied to specific appointments, medications, doctors, or patients. These indices are vital for pharmacy operations, medication management, and ensuring accurate prescription records.

Database Views

In a hospital management system, creating views can significantly enhance data accessibility and organization. Views are virtual tables representing the result of a stored query, allowing users to access a specific subset of the database without exposing the underlying table structures. Below is an explanation of the views created in the SQL script:

Patient Views

```sql
CREATE VIEW ViewPatientDetails
AS
SELECT p.CitizenCardNumber, p.Name, p.Gender, dbo.GetAge(p.DateOfBirth) AS Age,
       p.ContactInformation, p.Address, p.Insurance
FROM Patient p;
GO
```

This view consolidates patient information, including their citizen card number, name, gender, age (calculated using the GetAge function), contact information, address, and insurance status. It simplifies querying patient details by providing a comprehensive view of essential patient information.

View Patient Illness Details

```sql
CREATE VIEW ViewPatientIllnessDetails
AS
SELECT
    p.CitizenCardNumber, p.Name, p.Gender, dbo.GetAge(p.DateOfBirth) AS Age,
    p.ContactInformation, p.Address, p.Insurance,
    i.IllnessName, i.Description AS IllnessDescription
FROM
    Patient p
JOIN
    Illness i ON p.CitizenCardNumber = i.PatientID;
GO
```

This view extends the ViewPatientDetails by joining the Patient and Illness tables. It provides detailed information about patients along with their associated illnesses, including the illness name and description. This is useful for medical professionals needing a quick overview of a patient's medical history.

Appointment Views

```sql
CREATE VIEW vw_UpcomingAppointments
AS
SELECT a.AppointmentDateTime, a.Reason, p.Name AS PatientName, d.Name AS DoctorName
FROM Appointment a
JOIN Patient p ON a.PatientID = p.CitizenCardNumber
JOIN Doctor d ON a.DoctorID = d.CitizenCardNumber
WHERE a.AppointmentDateTime > GETDATE();
GO
```

This view provides a list of upcoming appointments, displaying the appointment date and time, reason for the appointment, and the names of the patient and doctor. By filtering appointments to those scheduled after the current date and time, this view helps manage and keep track of future appointments effectively.

Billing Views

```sql
CREATE VIEW ViewUnpaidBills
AS
SELECT
    b.BillID,
    p.CitizenCardNumber AS PatientID,
    d.CitizenCardNumber AS DoctorID,
    b.AppointmentID,
    b.PaymentStatus,
    b.TotalAmount,
    b.BillDate
FROM
    Bill b
JOIN
    Patient p ON b.PatientID = p.CitizenCardNumber
JOIN
    Doctor d ON b.DoctorID = d.CitizenCardNumber
WHERE
    b.PaymentStatus = 'Unpaid' OR b.PaymentStatus = 'Pending';
GO
```

This view focuses on unpaid bills, joining the Bill, Patient, and Doctor tables to provide a comprehensive look at outstanding payments. It includes bill ID, patient and doctor IDs, appointment ID, payment status, total amount, and bill date. This view is crucial for the finance department to monitor and manage pending and unpaid bills.

Conclusion

In conclusion, the hospital management system developed in this project addresses critical operational needs within healthcare facilities. It does so by integrating key functionalities such as patient and doctor management, appointment scheduling, and the management of billing and prescriptions.

The system leverages SQL to create a structured and effective database environment. Through the use of data definition language (DDL) scripts, the database structure is laid out with tables for patients, doctors, appointments, and other necessary elements. These tables are interconnected with foreign key constraints to maintain data integrity and relational logic.

Stored procedures play a vital role in the system, automating complex tasks like patient registration, appointment scheduling, and billing generation. For example, procedures such as *AddPatient* and *ScheduleAppointment* encapsulate the SQL commands needed to insert data into multiple tables while maintaining transactional integrity. This modular approach not only makes the code more manageable but also enhances security by limiting direct access to the underlying data structures.

Triggers are another SQL feature used in this system to ensure consistency and automate responses to database events. For instance, triggers such as *AutoGenerateBill* are set to run after new appointments are inserted into the database, automatically creating associated billing records. This reduces manual entry errors and ensures that bills are generated in real-time, directly after appointments are scheduled.

The system also includes comprehensive views and indexing strategies that optimize data retrieval and performance. Views such as *ViewPatientDetails* and *ViewUnpaidBills* provide simplified access to complex queries, allowing for efficient data handling and reporting. Meanwhile, indexes are strategically placed on frequently accessed fields to speed up query response times, essential for maintaining performance as the database grows.

Overall, the implemented SQL elements significantly contribute to the system's functionality and efficiency, supporting high-volume and complex operations typical in hospital environments.