**HospitalDB Project Report**

**Hospital Management System**

**Database Name:**

HospitalDB

**Objective:** To develop a comprehensive Hospital Management System using SQL Server that handles patient data, staff and doctor management, appointments, billing, services, room admissions, and administrative controls. The system ensures data integrity, facilitates reporting, and supports automated tasks.

**Key Features:**

* Stores patient and doctor information, including personal and contact details.
* Manages hospital departments and room availability.
* Handles billing and insurance discounts.
* Tracks appointments and medical records.
* Supports staff assignment and shift scheduling.
* Implement views for reporting and aggregation.
* Includes stored procedures and functions for core operations.
* Utilizes triggers for data consistency.
* Enforces security through DCL (roles and permissions).
* Supports automated job scheduling with SQL Server Agent.

**Technologies Used:**

* Microsoft SQL Server
* T-SQL (Transact-SQL)
* SQL Server Agent (Jobs & Scheduling)
* BCP (Bulk Copy Program) for data export
* SQL Security (Roles, GRANT/REVOKE)
* Views, Triggers, Stored Procedures, Scalar Functions

**Scope:** This system simulates a real-world hospital management environment for educational and practical implementation of relational database concepts. It covers DDL, DML, DQL, DCL, TCL, and automation components.

**HospitalDB Project - Page 1: Entity Relationship and Normalization**

**ERD (Entity Relationship Diagram)**

I created this ERD to represent the relationships between key entities in the hospital system. This helps model real-world processes like patient appointments, treatments, room assignments, and billing.

**Key Design Choices:**

* **Appointments as a relation**: I added Appointments as a separate relation because patients can have many appointments with different doctors over time. This is a classic many-to-many relationship that needs to be managed properly.
* **Admit as a relation**: Admission involves linking a patient to a room during a specific period. That’s why I used a separate Admit table to handle the M:N relationship between Patients and Rooms with attributes like DateIn and DateOut.

**Normalization Process**

**1NF (First Normal Form)**

I ensured that all tables:

* Have atomic values (no multiple values in one field).
* Each field contains only one type of data.

This prevents repeating groups and ensures clarity in structure.

**2NF (Second Normal Form)**

I applied 2NF to remove partial dependencies. I ensured that:

* Every non-prime attribute is fully functionally dependent on the whole primary key.
* For example, in Admit, I separated room attributes into the Rooms table instead of repeating them for each admission.

👉 I added PhoneNumbers as a separate table to remove multi-valued attributes from Patients, which is required for 2NF.

**3NF (Third Normal Form)**

I completed 3NF by removing transitive dependencies:

* For instance, in the Doctors table, department info like DepartmentName and DepartmentNumber were moved to a separate Departments table.

This prevents redundancy and ensures that data is organized cleanly.

**Conclusion**

By following the normalization steps to 3NF, I made sure the database is:

* Free of redundancy
* Easy to maintain
* Logically structured and efficient

**Tables Design and Normalization to 3NF**

In this section, I explain how I designed the tables for the HospitalDB system and ensured that they followed normalization rules up to Third Normal Form (3NF). The goal was to eliminate data redundancy and improve data integrity.

**First Normal Form (1NF):** To achieve 1NF, I ensured that all tables have atomic values. Each column contains only one value, and each record is unique. For example:

* The Patients table contains atomic fields like Name, Email, DOB, and Gender.
* Repeating groups such as multiple phone numbers were handled by creating a separate table called PatientPhoneNumbers.

**Second Normal Form (2NF):** In 2NF, we eliminated partial dependencies. That means every non-key attribute is fully functionally dependent on the whole primary key. This mainly applies to tables with composite primary keys.

* I split the Patients and PatientPhoneNumbers tables to ensure phone numbers do not rely on only part of a key.
* I created separate Departments, Staff, and Doctors tables to isolate data that depended only partially on composite keys.

**Why we added 2NF explanation in the report:** I explained 2NF to show the step-by-step improvement in our database schema. Adding 2NF allows better control over functional dependencies and reduces redundancy that may exist in 1NF. Including it helps the reader understand how we removed partial dependencies before proceeding to full normalization.

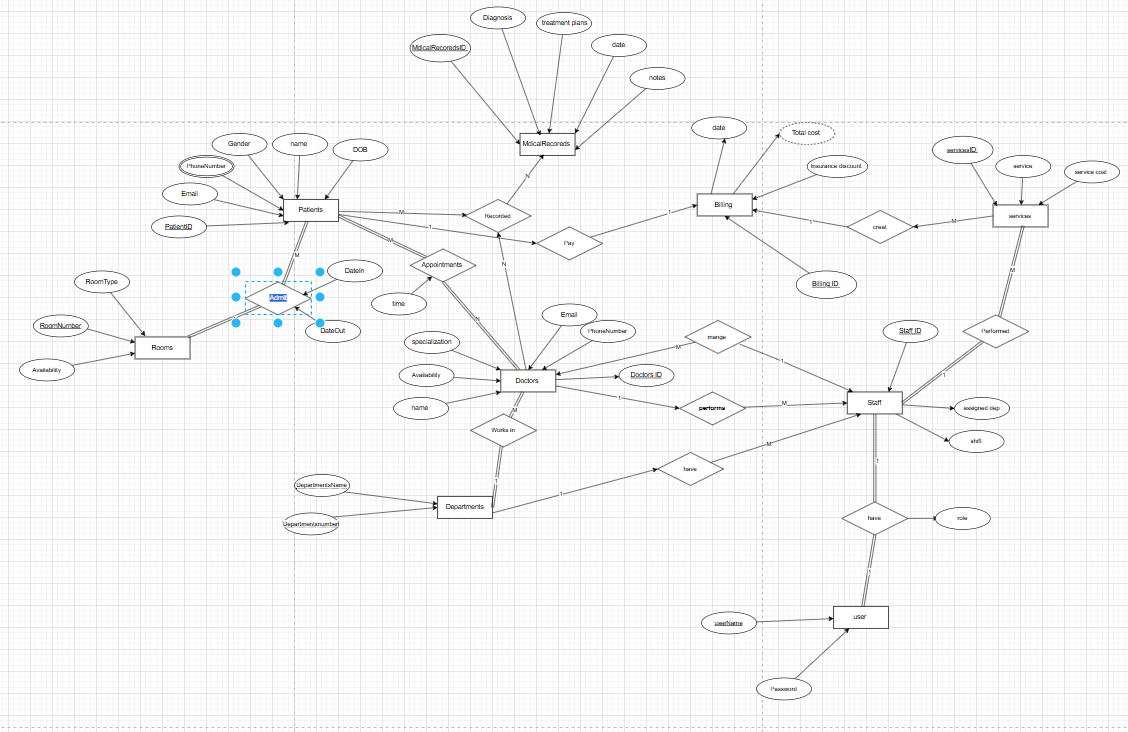
**Third Normal Form (3NF):** To achieve 3NF, I removed transitive dependencies. This means that non-key attributes are not allowed to depend on other non-key attributes.

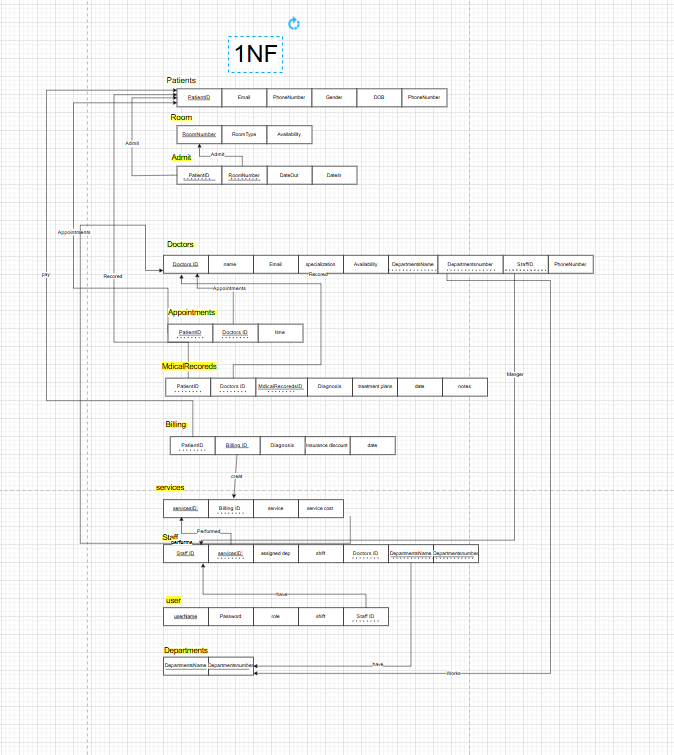
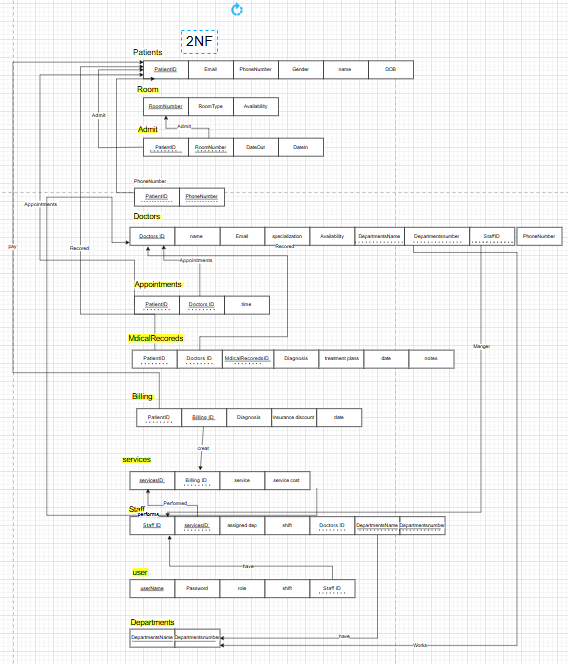
* For example, in the Doctors table, I avoided storing DepartmentName repeatedly by using a foreign key to the Departments table.
* Similarly, the Appointments table links Patients and Doctors using foreign keys without repeating their attributes.

**Why we added Appointments as a relation:** We included the Appointments relation to manage the many-to-many relationship between Patients and Doctors. One doctor can have many appointments with different patients, and one patient can visit many doctors. The Appointments table includes fields like PatientID, DoctorsID, and Time to track each unique visit.

**Why we added Admit as a relation:** The Admit relation connects Patients to Rooms. A patient may be admitted to a room during a hospital stay, so we needed to track the DateIn, DateOut, and RoomNumber. This relation manages hospital stays and ensures room usage is logged properly. It also helps control room availability.

**Conclusion:** With these normalization steps and relational designs, we ensured that our HospitalDB system avoids redundancy, supports accurate queries, and makes future updates and extensions more efficient.

THE ERD:

 THE Maping 1 NF : THE Maping 2 NF :

**Explanation of Relationships and Design Decisions**

In this section, I will explain why certain relationships and tables were created in the HospitalDB schema and how they contribute to the overall data integrity and functional requirements of the hospital system.

**1. Appointments (Relationship Table)**

* **Why it's a relationship:** The Appointments table links Patients and Doctors. One patient can have many appointments with different doctors, and one doctor can treat many patients.
* **Why it's needed:** It allows us to track the date and time of each visit, enforce many-to-many relationships, and support queries like "which doctor treated which patient and when."
* **Normalization:** It holds only foreign keys and atomic time data, ensuring compliance with 3NF.

**2. Admit (Relationship Table)**

* **Why it's a relationship:** It connects Patients with Rooms to represent admission events. One patient may be admitted multiple times, and rooms may be reused.
* **Why it's needed:** Tracks hospitalization history, room usage, and stay duration using attributes DateIn and DateOut.
* **Normalization:** Fully normalized; no repeating groups or partial dependencies.

**3. Staff Table**

* **Why it's separate from Doctors/Users:**
  + Doctors are staff members, but we might also have other roles like nurses or technicians in future updates.
  + It allows detailed assignment of Shift, AssignedDep, and ServiceID to staff members.
* **Why it's needed:** Facilitates access control and role management through the Users table.
* **Normalization:** It holds single-valued, non-redundant fields, adhering to 3NF.

**4. Services Table**

* **Why it's linked to Billing:** Each billing event may involve one or more services. Services store details like ServiceName and ServiceCost.
* **Why it's needed:** Enables itemized billing and cost analysis per patient.
* **Normalization:** Atomic values only, no derived or redundant data.

**5. Users Table**

* **Why it's linked to Staff:** Users inherit access rights from their Staff ID. This enables login control and role-based functionality.
* **Why it's needed:** Supports login system and role-based access for Doctors and Admins.
* **Why we removed Shift:** Shift is already maintained in Staff and not needed in Users, avoiding redundancy (2NF compliance).

**6. Departments Table**

* **Why it's a separate entity:** Departments are reused across Doctors and Staff, and have distinct attributes like DepartmentName and DepartmentNumber.
* **Why it's needed:** To centralize department data and avoid redundancy in Doctors/Staff.
* **Normalization:** Already in 3NF.

**Summary of Design Benefits**

* **Modularity:** Each table handles one responsibility.
* **Scalability:** New roles or services can be added easily.
* **Query efficiency:** Data is normalized, so joins are accurate and performance-friendly.
* **Error prevention:** Foreign keys enforce referential integrity (e.g., billing cannot exist for a non-existing patient).

**Page 4: Data Insertion & Integrity**

In this phase, we inserted real data into the HospitalDB system to simulate a fully functioning environment for testing queries and reports. Data was carefully structured and inserted into the tables in a logical order to preserve referential integrity. Below is the explanation of the data insertion strategy and practices.

**Order of Data Insertion:**

1. **Departments**: Base table without dependencies.
2. **Patients**: Basic information for patients.
3. **PatientPhoneNumbers**: Depends on Patients.
4. **Room**: Independent room availability data.
5. **Staff**: Required before linking Doctors and Users.
6. **Doctors**: Links to Departments and Staff.
7. **Billing**: Linked to Patients.
8. **Services**: Linked to Billing.
9. **Users**: Linked to Staff.
10. **Appointments**: Links Patients with Doctors.
11. **MedicalRecords**: Needs Doctors and Patients.
12. **Admit**: Links patients to room assignments.

**Data Integrity Practices:**

* **Primary Keys**: Set for all tables to ensure unique identification.
* **Foreign Keys**: Defined and enforced to maintain referential integrity.
* **Data Types & Lengths**: Carefully selected to match realistic hospital operations (e.g., VARCHAR for emails, BIT for availability).
* **Constraints**: Applied to prevent data entry errors (e.g., no duplicate IDs, correct FK references).

**Sample Entries:**

* **Departments**: 20 department entries from "Cardiology" to "Rehabilitation".
* **Patients**: 20 real-like patients with full names, emails, DOBs, and gender.
* **Doctors**: 20 specialists each mapped to their department and staff ID.

**DQL Queries with Explanations**

This section includes practical SQL queries that are used to test the understanding and implementation of Data Query Language (DQL) commands. Each query is followed by an explanation of its purpose and what it demonstrates.

**1. List all patients who visited a certain doctor (e.g., Dr. Ahmed Ali)**

SELECT P.PatientID, P.Name AS PatientName, D.Name AS DoctorName

FROM Appointments A

JOIN Patients P ON A.PatientID = P.PatientID

JOIN Doctors D ON A.DoctorsID = D.DoctorsID

WHERE D.Name = 'Dr. Ahmed Ali';

*Purpose:* Tests understanding of filtering results using WHERE and joining related entities.  
*Use Case:* Track patients treated by a specific doctor.

**2. Count of appointments per department**

SELECT D.DepartmentName, COUNT(\*) AS AppointmentCount

FROM Appointments A

JOIN Doctors D ON A.DoctorsID = D.DoctorsID

GROUP BY D.DepartmentName;

*Purpose:* Tests grouping and counting.  
*Use Case:* Helps identify departments with the highest patient visits.

**3. Retrieve doctors who have more than 5 appointments in a month**

SELECT D.DoctorsID, D.Name AS DoctorName, COUNT(\*) AS TotalAppointments

FROM Appointments A

JOIN Doctors D ON A.DoctorsID = D.DoctorsID

WHERE MONTH(A.Time) = 6 AND YEAR(A.Time) = 2024

GROUP BY D.DoctorsID, D.Name

HAVING COUNT(\*) > 5;

*Purpose:* Tests GROUP BY, HAVING, and date filters.  
*Use Case:* Identify busy doctors to balance workload.

**4. JOIN across 3–4 tables: Patient, Doctor, Department, MedicalRecords**

SELECT P.Name AS PatientName, D.Name AS DoctorName, D.DepartmentName, M.Diagnosis

FROM MedicalRecords M

JOIN Patients P ON M.PatientID = P.PatientID

JOIN Doctors D ON M.DoctorsID = D.DoctorsID;

*Purpose:* Validates ability to combine data from multiple sources.  
*Use Case:* Provide a complete view of diagnosis.

**5. Use GROUP BY, HAVING, and aggregate: Average service cost per department**

SELECT D.DepartmentName, AVG(S.ServiceCost) AS AvgCost

FROM Services S

JOIN Billing B ON S.BillingID = B.BillingID

JOIN Patients P ON B.PatientID = P.PatientID

JOIN MedicalRecords M ON M.PatientID = P.PatientID

JOIN Doctors D ON M.DoctorsID = D.DoctorsID

GROUP BY D.DepartmentName

HAVING AVG(S.ServiceCost) > 300;

*Purpose:* Demonstrates nested joins with aggregation and filtering.  
*Use Case:* Monitor cost efficiency per department.

**6. Use EXISTS: List patients who have medical records**

SELECT P.Name

FROM Patients P

WHERE EXISTS (

SELECT 1 FROM MedicalRecords M

WHERE M.PatientID = P.PatientID

);

*Purpose:* Tests correlated subqueries with EXISTS.  
*Use Case:* Quickly find documented patients.

**7. Use NOT EXISTS: List patients who never had an appointment**

SELECT P.Name

FROM Patients P

WHERE NOT EXISTS (

SELECT 1 FROM Appointments A

WHERE A.PatientID = P.PatientID

);

*Purpose:* Tests understanding of exclusion logic.  
*Use Case:* Identify patients who need engagement.

**8. Total services and cost per patient**

SELECT P.Name AS PatientName, COUNT(S.ServiceID) AS ServicesReceived, SUM(S.ServiceCost) AS TotalCost

FROM Patients P

JOIN Billing B ON P.PatientID = B.PatientID

JOIN Services S ON B.BillingID = S.BillingID

GROUP BY P.Name;

*Purpose:* Validates usage of aggregation with joins.  
*Use Case:* Analyze billing records.

**9. Find doctors who treated patients for 'Cancer'**

SELECT DISTINCT D.Name AS DoctorName

FROM MedicalRecords M

JOIN Doctors D ON M.DoctorsID = D.DoctorsID

WHERE CAST(M.Diagnosis AS VARCHAR(100)) = 'Cancer';

*Purpose:* Demonstrates data type casting and filtering.  
*Use Case:* Track cancer specialists.

**10. Number of admissions per room**

SELECT R.RoomNumber, COUNT(\*) AS AdmissionsCount

FROM Admit A

JOIN Room R ON A.RoomNumber = R.RoomNumber

GROUP BY R.RoomNumber;

*Purpose:* Measures room usage with grouping.  
*Use Case:* Optimize room allocation.

**11. List doctors who are available (Availability = 1) and have future appointments**

SELECT DISTINCT D.Name

FROM Doctors D

JOIN Appointments A ON D.DoctorsID = A.DoctorsID

WHERE D.Availability = 1 AND A.Time > GETDATE();

*Purpose:* Combines filtering with temporal condition.  
*Use Case:* Scheduling efficiency.

**12. Top 3 departments with most appointments**

SELECT TOP 3 D.DepartmentName, COUNT(\*) AS AppointmentsCount

FROM Appointments A

JOIN Doctors D ON A.DoctorsID = D.DoctorsID

GROUP BY D.DepartmentName

ORDER BY AppointmentsCount DESC;

*Purpose:* Combines TOP, GROUP BY, ORDER BY.  
*Use Case:* Identify peak departments.

**Functions & Stored Procedures**

This page demonstrates the scalar function and stored procedures implemented in the HospitalDB system. These help automate core operations, simplify recurring tasks, and ensure integrity and modularity of logic.

**1. Scalar Function: Calculate Patient Age**

CREATE FUNCTION dbo.GetPatientAge(@DOB DATE)

RETURNS INT

AS

BEGIN

RETURN DATEDIFF(YEAR, @DOB, GETDATE())

END;

**Purpose:** Computes the current age of a patient based on their Date of Birth.

**2. Procedure: Admit a Patient**

CREATE PROCEDURE AdmitPatient

@PatientID INT,

@DateIn DATE,

@DateOut DATE,

@RoomNumber INT

AS

BEGIN

INSERT INTO Admit (PatientID, DateIn, DateOut, RoomNumber)

VALUES (@PatientID, @DateIn, @DateOut, @RoomNumber);

UPDATE Room

SET Availability = 0

WHERE RoomNumber = @RoomNumber;

END;

**Purpose:** Inserts a record into the Admit table and updates room availability to 0 (occupied).

**3. Procedure: Generate Invoice**

CREATE PROCEDURE GenerateInvoice

@PatientID INT,

@Diagnosis VARCHAR(100),

@InsuranceDiscount DECIMAL(5,2),

@Date DATE

AS

BEGIN

INSERT INTO Billing (PatientID, Diagnosis, InsuranceDiscount, Date)

VALUES (@PatientID, @Diagnosis, @InsuranceDiscount, @Date);

END;

**Purpose:** Adds billing information based on a patient's diagnosis and insurance.

**4. Procedure: Assign Doctor to Department and Shift**

CREATE PROCEDURE AssignDoctorToDepartment

@StaffID INT,

@DoctorsID INT,

@DepartmentName VARCHAR(100),

@DepartmentNumber INT,

@Shift VARCHAR(50)

AS

BEGIN

UPDATE Staff

SET DoctorsID = @DoctorsID,

DepartmentName = @DepartmentName,

DepartmentNumber = @DepartmentNumber,

Shift = @Shift

WHERE StaffID = @StaffID;

END;

**Purpose:** Dynamically updates the department, doctor, and shift details of a staff member.

**5. Extra Procedure: Mark Room Available**

CREATE PROCEDURE MarkRoomAvailable

@RoomNumber INT

AS

BEGIN

UPDATE Room

SET Availability = 1

WHERE RoomNumber = @RoomNumber;

END;

**Purpose:** Makes a room available again after discharge.

**6. Extra Procedure: Delete Patient and Related Data**

CREATE PROCEDURE DeletePatientData

@PatientID INT

AS

BEGIN

DELETE FROM Billing WHERE PatientID = @PatientID;

DELETE FROM Admit WHERE PatientID = @PatientID;

DELETE FROM Appointments WHERE PatientID = @PatientID;

DELETE FROM MedicalRecords WHERE PatientID = @PatientID;

DELETE FROM Patients WHERE PatientID = @PatientID;

END;

**Purpose:** Ensures full cleanup of patient data including bills, appointments, and admissions.

**Triggers**

**What are Triggers in SQL?**  
A **trigger** in SQL is a special type of stored procedure that automatically executes or fires when certain events occur in the database. These events could be:

* AFTER INSERT – After a new record is inserted
* AFTER UPDATE – After a record is updated
* AFTER DELETE – After a record is deleted
* INSTEAD OF – Replaces the original action with custom logic

Triggers are used to:

* Enforce business rules
* Automatically maintain data consistency
* Log actions or history
* Prevent unwanted changes

**Implemented Triggers in HospitalDB:**

**1. AFTER INSERT on Appointments → Auto insert into MedicalRecords**

CREATE TRIGGER trg\_AutoInsertMedicalRecord

ON Appointments

AFTER INSERT

AS

BEGIN

INSERT INTO MedicalRecords (PatientID, DoctorsID, MedicalRecordsID, Diagnosis, TreatmentPlans, Date, Notes)

SELECT

i.PatientID,

i.DoctorsID,

(SELECT ISNULL(MAX(MedicalRecordsID), 0) + 1 FROM MedicalRecords),

'To be updated',

'To be updated',

GETDATE(),

'Auto-generated from appointment'

FROM inserted i;

END;

**Purpose:** Automatically creates a blank medical record whenever a new appointment is added.

**2. INSTEAD OF DELETE on Patients → Prevent deletion if bills exist**

CREATE TRIGGER trg\_PreventDeletePatientWithBills

ON Patients

INSTEAD OF DELETE

AS

BEGIN

IF EXISTS (

SELECT 1 FROM Billing B

JOIN deleted d ON B.PatientID = d.PatientID

)

BEGIN

RAISERROR('Cannot delete patient with existing billing records.', 16, 1);

ROLLBACK TRANSACTION;

END

ELSE

BEGIN

DELETE FROM Patients WHERE PatientID IN (SELECT PatientID FROM deleted);

END

END;

**Purpose:** Prevents deletion of patients if they have associated billing records to preserve financial data integrity.

**3. AFTER INSERT/UPDATE on Admit → Prevent assigning same room to multiple patients**

CREATE TRIGGER trg\_NoDoubleOccupancy

ON Admit

AFTER INSERT, UPDATE

AS

BEGIN

IF EXISTS (

SELECT RoomNumber

FROM Admit

GROUP BY RoomNumber, DateIn, DateOut

HAVING COUNT(\*) > 1

)

BEGIN

RAISERROR('Room conflict detected: two patients in the same room.', 16, 1);

ROLLBACK TRANSACTION;

END

END;

**Purpose:** Ensures that no two patients are admitted into the same room at the same time.

**4. EXTRA – Prevent appointments in the past**

CREATE TRIGGER trg\_NoPastAppointments

ON Appointments

AFTER INSERT

AS

BEGIN

IF EXISTS (

SELECT 1 FROM inserted WHERE Time < GETDATE()

)

BEGIN

RAISERROR('Cannot schedule appointment in the past.', 16, 1);

ROLLBACK TRANSACTION;

END

END;

**Purpose:** Prevents users from adding appointments with a past date/time.

**5. EXTRA – Automatically insert a default service after billing**

CREATE TRIGGER trg\_DefaultServiceOnBilling

ON Billing

AFTER INSERT

AS

BEGIN

INSERT INTO Services (BillingID, Service, ServiceCost)

SELECT i.BillingID, 'General Consultation', 100.00

FROM inserted i;

END;

**Purpose:** Automatically adds a default service entry (like consultation) every time a billing record is created.

**SQL Security, Transactions, Views, and SQL Jobs**

**SQL Security (DCL)**

Security ensures only authorized users can access/modify data.

**1. Roles and Permissions:**

CREATE ROLE DoctorUser;

CREATE ROLE AdminUser;

GRANT SELECT ON Patients TO DoctorUser;

GRANT SELECT ON Appointments TO DoctorUser;

GRANT INSERT, UPDATE ON DATABASE::HospitalDB TO AdminUser;

REVOKE DELETE ON Doctors TO PUBLIC;

DENY UPDATE ON Patients TO DoctorUser;

GRANT CREATE PROCEDURE TO AdminUser;

**Explanation:**

* DoctorUser: Can only view Patients and Appointments, cannot modify.
* AdminUser: Can insert and update across the HospitalDB database.
* Restricts delete access and update from DoctorUser.

**Transaction Control Language (TCL)**

Ensures all related changes are completed together, or not at all (rollback on failure).

BEGIN TRANSACTION;

BEGIN TRY

INSERT INTO Admit (PatientID, DateIn, DateOut, RoomNumber)

VALUES (21, '2025-06-25', '2025-06-30', 1);

UPDATE Room SET Availability = 0 WHERE RoomNumber = 1;

INSERT INTO Billing (BillingID, PatientID, Diagnosis, InsuranceDiscount, Date)

VALUES (21, 21, 'Infection', 0.10, GETDATE());

INSERT INTO Services (ServiceID, BillingID, Service, ServiceCost)

VALUES (21, 21, 'Antibiotics Therapy', 250.00);

INSERT INTO MedicalRecords (PatientID, DoctorsID, MedicalRecordsID, Diagnosis, TreatmentPlans, Date, Notes)

VALUES (21, 5, 21, 'Infection', 'Antibiotics', GETDATE(), 'Auto-log from transaction');

COMMIT;

PRINT 'Transaction completed successfully.';

END TRY

BEGIN CATCH

ROLLBACK;

PRINT 'Transaction failed. Rolled back due to error: ' + ERROR\_MESSAGE();

END CATCH;

**Explanation:**

* Simulates admitting a patient and related updates.
* If one step fails, all changes are rolled back.

**Views**

**1. vw\_DoctorSchedule – Upcoming appointments:**

CREATE VIEW vw\_DoctorSchedule AS

SELECT D.DoctorsID, D.Name AS DoctorName, A.PatientID, A.Time AS AppointmentTime

FROM Doctors D

JOIN Appointments A ON D.DoctorsID = A.DoctorsID

WHERE A.Time > GETDATE();

**2. vw\_PatientSummary – Latest patient visit:**

CREATE VIEW vw\_PatientSummary AS

SELECT P.PatientID, P.Name AS PatientName, MAX(A.Time) AS LastVisit

FROM Patients P

LEFT JOIN Appointments A ON P.PatientID = A.PatientID

GROUP BY P.PatientID, P.Name;

**3. vw\_DepartmentStats – Department doctor and patient count:**

CREATE VIEW vw\_DepartmentStats AS

SELECT D.DepartmentName,

COUNT(DISTINCT Doc.DoctorsID) AS TotalDoctors,

COUNT(DISTINCT A.PatientID) AS TotalPatients

FROM Departments D

LEFT JOIN Staff S ON D.DepartmentNumber = S.DepartmentNumber

LEFT JOIN Doctors Doc ON S.DoctorsID = Doc.DoctorsID

LEFT JOIN Appointments A ON Doc.DoctorsID = A.DoctorsID

GROUP BY D.DepartmentName;

**4. vw\_ServiceCosts – Total service cost per patient:**

CREATE VIEW vw\_ServiceCosts AS

SELECT B.PatientID, SUM(S.ServiceCost) AS TotalServiceCost

FROM Billing B

JOIN Services S ON B.BillingID = S.BillingID

GROUP BY B.PatientID;

**5. vw\_UpcomingAdmissions – Admissions in the future:**

CREATE VIEW vw\_UpcomingAdmissions AS

SELECT A.PatientID, A.RoomNumber, A.DateIn, A.DateOut

FROM Admit A

WHERE A.DateIn > GETDATE();

**Explanation:**

* Views provide reusable, simplified data queries for reporting, monitoring, and summaries.

**SQL Jobs (SQL Server Agent)**

Automate backup, reporting, and alerts with scheduled jobs.

**1. Doctor Daily Schedule Report**

CREATE TABLE DoctorDailyScheduleLog (...);

CREATE PROCEDURE InsertDailyDoctorSchedule AS BEGIN ... END;

**Job Code:**

* Creates job, adds step to run procedure, schedules it at 7:00 AM daily.

**2. Backup Job**

BACKUP DATABASE HospitalDB TO DISK = 'C:\Backups\HospitalDB.bak' ...;

**Explanation:**

* Ensures database is backed up daily at 2:00 AM.

**3. Alert: Doctor with more than 5 appointments**

IF EXISTS (...) BEGIN EXEC sp\_send\_dbmail ... END;

**Explanation:**

* Sends email to ahmad.k@hospital.com if any doctor exceeds 5 daily appointments.

**4. Export Billing Weekly**

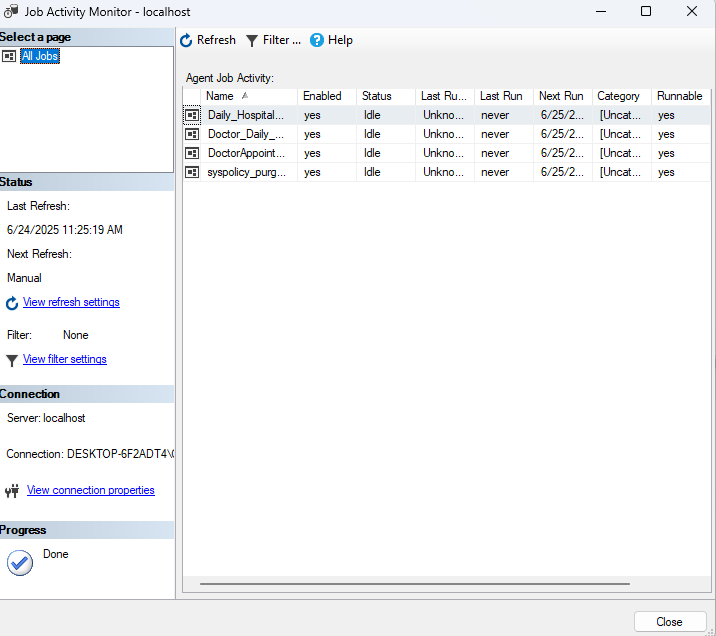
EXEC xp\_cmdshell 'mkdir C:\Exports';

...

bcp "SELECT \* FROM HospitalDB.dbo.Billing" queryout "C:\Exports\BillingSummary.csv" -c -t, -T -S localhost

**Explanation:**

* Exports weekly billing data every Sunday at 9:00 AM as .csv file using bcp command.

****

**Error Section that I get in implementation**

**1. Error: Data types of text and varchar are incompatible in the equal to operator**

**Query Attempted:**

SELECT DISTINCT D.Name AS DoctorName

FROM MedicalRecords M

JOIN Doctors D ON M.DoctorsID = D.DoctorsID

WHERE M.Diagnosis = 'Cancer';

**Error Message:**

Msg 402, Level 16, State 1, Line 65

The data types of text and varchar are incompatible in the equal to operator.

**Solution:**  
Cast M.Diagnosis to VARCHAR type before comparing.

WHERE CAST(M.Diagnosis AS VARCHAR(100)) = 'Cancer';

**Reason:** text type must be cast to varchar to compare using =.

**2. Error: CREATE PROCEDURE must be the only statement in the batch**

**Query Attempted:**

CREATE PROCEDURE AdmitPatient ...

**Error Message:**

Incorrect syntax: 'CREATE PROCEDURE' must be the only statement in the batch.

**Solution:**  
Use GO before and after the CREATE PROCEDURE block.

GO

CREATE PROCEDURE ...

GO

**Reason:** SQL Server needs procedure creation to be isolated in its own batch.

**3. Error: Schedule does not exist / Schedule name conflict**

**Command Used:**

EXEC msdb.dbo.sp\_attach\_schedule @schedule\_name = 'Daily\_7AM\_Schedule';

**Error Message:**

The specified @schedule\_name ('Daily\_7AM\_Schedule') does not exist.

**Or:**

There are two or more schedules named 'Daily\_11AM\_Schedule'...

**Solution:**

* Ensure you **create** the schedule before attaching it.
* Use **unique names** for each schedule.
* If duplicates exist, use @schedule\_id instead of @schedule\_name.

**4. Error: xp\_cmdshell or Agent XPs disabled**

**Action:** Trying to create a SQL job using PowerShell or BCP.  
**Error Message:**

SQL Server Agent (Agent XPs) is disabled

**Solution:**  
Run:

EXEC sp\_configure 'show advanced options', 1;

RECONFIGURE;

EXEC sp\_configure 'Agent XPs', 1;

RECONFIGURE;

**5. Error: The schedule is invalid - freq\_interval must be at least 1**

**Error Message:**

The schedule for this job is invalid (reason: @freq\_interval must be at least 1).

**Solution:**  
Add @freq\_interval = 1 when using @freq\_type = 4 (Daily).

@freq\_type = 4,

@freq\_interval = 1,

**Reason:** Even daily jobs need interval (days between jobs).

**6. Error: Cannot delete patient with billing**

**Trigger Functionality:**

INSTEAD OF DELETE ON Patients

**Behavior:**  
Prevents deletion of patients if there are related billing records.  
**Error Message:**

RAISERROR('Cannot delete patient with existing billing records.', 16, 1);

**Solution:**  
Add INSTEAD OF DELETE trigger to handle logic safely.

**7. Error: SQL Server Agent is not running**

**Behavior:** Jobs won't execute.  
**Solution:**

* Start SQL Server Agent manually or use configuration.
* Ensure SQL Server Agent is enabled in SSMS.