VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Database System (Lab) (CO2014)

Assignment Report - Group CC06

A Hospital Database System

Advisor: Ph.D Đỗ Thành Thái
Students: Vũ Hoàng Tùng - 2252886
Phan Thanh Sơn - 2252718
Nguyễn Hồ Phi Ưng - 2252897

HO CHI MINH CITY, October 2024

Contents

Chapter 1

Conceptual design, logical design, database implementation on a DBMS, and exploring and utilizing technologies for developing applications on the chosen DBMS.

1.1 List of strong entities

A sufficient hospital database must cover key information of essential elements, such as patients, doctors. Those core components are represented by strong entities, and linking relationships describe how these entities interact. It is also what this chapter primarily focuses on: the attributes, interactions, and significance of each strong entity.

1.1.1 Patient Record

The Patient Record is a strong entity that holds all essential personal and medical information for each patient within the hospital system. This entity includes the following attributes:

- Record Number (Primary Key) – A unique identifier for each patient.
- First Name – The patient's first name.
- Last Name – The patient's last name.
- Gender – The gender of the patient. (e.g., Male, Female).
- Contact Information – The patient's phone number and email.
- Home Address – The patient's residential address(Street, District, City).

- Date of Birth – The patient's date of birth. (e.g., 18/07/2004).
- Current Medications – Medications the patient is currently prescribed.
- Emergency Contact – Contact information of a designated emergency contact (Phone number).

1.1.2 Employee

The Employee entity is a strong entity that captures all essential personal and professional information for each staff member within the hospital system, facilitating effective human resource management and operational efficiency. This entity includes the following attributes:

- Employee ID (Primary Key): A unique identifier for each employee (e.g., D001 for Doctor, N002 for Nurse).
- Name – The full name of the employee (e.g., John Smith).
- Gender – The gender of the employee
- Date of Birth – The date of birth of the employee.
- Job Type – The category of employment (e.g., Doctor, Nurse, Technician, Others).
- Experience: The amount of time that person has worked in his/her profession (e.g., 1 year).
- Salary – The employee's annual or hourly salary (e.g., $80,000/year or $40/hour).
- Contact Details – Phone number(s) and email address.
- Start Date – The date the employee began working at the hospital
- Department Number (Foreign Key) – A unique identifier for the department in which the employee works, linking to the Department entity

Inheritance and Specialization: The Employee entity serves as a superclass for various specialized subclasses, each with additional attributes:

1. Doctor:
    - Specialty: Medical specialty. E.g., Cardiology, Pediatrics, ...
    - Certificate: Diagnosis and treatment of patients, conducting medical procedures, and collaborating with other healthcare professionals.
2. Nurse:
    - Specialty: Medical Specialty. E.g: Diagnosis, Nursing
3. Technician:
    - Specialty: Technical Specialization. Eg: Electricity, Medical Facility, ...
    - Responsibility: Their individual tasks corresponding to the Specialty. For instance, those with the same Specialty of Medical Facility could maintain different kinds of items, or discard medical items.
4. Other:
    - Job - type: Receptionist, Janitor, ...
    - Responsibility: Their individual tasks corresponding to the Jobtype.

1.1.3 Department

The Department strong entity represents a specific unit that provides specialized medical services and manages healthcare professionals. This entity includes the following attributes:

- Department ID (Primary Key) – A unique identifier for each department.
- Department Name – The name of the department (e.g., Cardiology, Pediatrics).
- Location – The physical location of the department within the hospital (e.g., First Floor, East Wing), or in some rare case, outside the hospital.

1.1.4 Room

The Room entity represents rooms in the hospital and within departments, designed to serve various purpose, mainly accommodate patients during their stay in the hospital. It is not a weak entity of Department, since a Room is a separated unit, serving its own purpose, within or without the Department (some rooms like Meeting Room, Canteen do not belong to a certain Department)

- Room Number (Primary Key) – A unique identifier for each room.
- Department Number (Foreign Key) – A unique identifier linking the room to its respective department.
- Room Type – The type of room (e.g., Patient care, ICU).
- Room Name – The name of the room
- Status – Current condition of the room

1.1.5 Equipment

The Equipment entity keeps track of the equipment used in each surgery or diagnostic test, ensuring that the hospital maintains a comprehensive inventory of essential medical tools. This includes the following attributes:

- Equipment ID (Primary Key) – A unique identifier for each piece of equipment
- Name – The name of the equipment (e.g., MRI Machine, Surgical Scalpel).
- Type – The category of equipment (e.g., Medical, Diagnostic, or Surgery).
- Status – The current availability of the equipment (e.g., Available, Not Available, Malfunction).

1.1.6 Surgery

The Surgery entity records data related to surgical procedures performed on patients, capturing important details about each operation. Attributes:

- Surgery ID (Primary Key) – A unique identifier for each surgery.
- Type of Surgery – The specific type of surgery .

- Date of Surgery – The date when the surgery was conducted.
- Outcome – The result of the surgery (e.g., Successful, Complications).
- Complications – Any complications that arose during or after the surgery (e.g., Infection, Hemorrhage).

1.1.7 Diagnostic Test

The Diagnostic Test entity tracks various tests patients undergo to determine their medical conditions, including results and relevant details. This includes the following attributes:

- Test ID (Primary Key) – A unique identifier for each test.
- Test Name – The name of the diagnostic test (e.g., Blood Test, X-Ray).
- Description – A brief description of the test (e.g., Complete blood count).
- Date – The date when the test was performed.
- Results – The outcomes of the test (e.g., Negative, Positive).

1.1.8 Insurance

The Insurance entity handles information about patients' insurance policies, which can cover parts or all of a patient's medical expenses. The attributes are:

- Insurance ID (Primary Key) – A unique identifier for each insurance policy.
- Policy Number – The number of the insurance policy.
- Priority – Indicates which insurance will be applied first if a patient has multiple policies.
- Provider – The name of the insurance company (e.g., Health Insurance Co.).
- Status – The current status of the insurance policy (e.g., Active, Pending, Expired).
- Coverage Percentage – The percentage of the bill covered by insurance.
- Coverage Limit – The maximum amount the insurance will cover per year (or per life).

1.1.9 Billing

The Billing entity keeps track of the costs associated with patient care, treatments, surgeries, diagnostic tests, medications, and other services provided during a patient's stay or visit. It includes the below attributes:

- Billing ID (Primary Key) – A unique identifier for each billing record.
- Date Issued – The date when the billing was created.
- Initial Amount – The amount charged before applying insurance.
- Cover Amount – The amount covered by insurance.
- Final Amount – The amount the patient has to pay.
- Due Date – The date by which the payment is due.
- Status – The payment status of the billing.

1.1.10 Payment

The Payment Record entity tracks each payment made towards a billing record, detailing the amount paid and the method of payment. Attributes:

- Payment ID (Primary Key) – A unique identifier for each payment.
- Payment Date – The date when the payment was made.
- Method – The method of payment (e.g., Cash, Credit Card).
- Amount Paid – The amount paid in the transaction.
- Payment Receipt Number – A reference number for the payment receipt.
- Notes – Any additional notes regarding the payment (e.g., "Partial payment").

1.2 Relationships between strong entity types

1.2.1 Patient - Billing

The relationship between **Patient** and **Billing** links each patient to one or more billing records that are generated for their treatment.

- **Denoted by:** Have
- **Relationship's type:** Each patient may have multiple billings, while each billing record is associated with only one patient. Thus, this relationship is a **1:N** relationship.
- **Implementation:** An attribute named PatientID is added to the Billing entity as a Foreign Key referring to RecordID (Primary Key of Relation)

1.2.2 Patient - Insurance

The relationship between **Patient** and **Insurance** links each patient to one or more insurance policies that they hold.

- **Denoted by:** Have
- **Relationship's type:** Each patient may have multiple insurance policies, while each insurance is associated with only one patient through a priority system to determine which insurance is applied first in case of multiple policies. Thus, this relationship is **1:N** relationship.
- **Implementation:** An attribute named **PatientID** is added to the **Insurance** entity as a **Foreign Key** referring to **RecordID** (Primary Key of Patient entity).

1.2.3 Insurance - Billing

The relationship between **Insurance** and **Billing** links each billing record to one or more insurance policies that may cover part or all of the bill's amount.

- **Denoted by:** Cover

- **Relationship's type:** Each billing record can be covered by multiple insurance policies (as long as the coverage limit is not exceeded), and each insurance can be associated with multiple billing records. Thus, this relationship is an **N:M** relationship.
- **Implementation:** A junction table named Cover is introduced with a group of 2 Primary Keys **BillingID** (Foreign Key referring to **Billing** entity) and **InsuranceID** (Foreign Key referring to **Insurance** entity).

1.2.4 Payment - Billing

The relationship between **Billing** and **Payment** links each billing record to one or more payments made toward the required money.

- **Denoted by:** Pay
- **Relationship's type:** Each billing record may have multiple payments associated with it (e.g., partial payments), while each payment is associated with only one billing record. Thus, this relationship is a **1:N** relationship.
- **Implementation:** An attribute named **BillingID** is added to the Payment entity as a Foreign Key referring to **BillingID** (Primary Key of the Billing entity).

1.2.5 Doctor - Patient and Nurse - Patient

The relationship between Patient and Doctor or Nurse links each doctor or nurse to one or more patient that they need to treat.

- **Denoted by:** Assign
- **Relationship's type:** Each patient is assigned to **one doctor**, but each doctor can be assigned to multiple patients. Thus, this relationship is a **1:N**relationship.
- **Implementation:** Two junction tables named **Assign_Doc** and **Assign_Nurse** are introduced. Each table has a group of 2 Primary Keys:
  - **DoctorID** (Foreign Key referring to the **Doctor** entity)
  - **RecordID** (Foreign Key referring to the **Patient** entity).

1.2.6 Technician - Equipment

The relationship between **Technician** and **Equipment** links each technician to the equipment they are responsible for maintaining.

- **Denoted by:** Maintains
- **Relationship's type:** Each technician is responsible for maintaining multiple pieces of equipment, but each piece of equipment is assigned to only one technician. Thus, this relationship is a **1:N** relationship.
- **Implementation:** A new table is added with the following attribute:
  - Primary Key: Equipment ID, referring to Equipment Entity.
  - 2nd Foreign Key: TechID, referring to Technician entity.
  - Other attributes: Type and Date

1.2.7 Surgery - Doctor - Patient

The relationship among **Surgery**, **Doctor**, and **Patient** links each surgery to the doctor performing it and the patient undergoing it.

- **Denoted by:** Performe - Surgery
- **Relationship's type:** this relationship is a **ternary** relationship of 3 entities.
- **Implementation:** A junction table is introduced with the following group of 3 Primary Keys:
  - **SurgeryID** (Foreign Key referring to the **Surgery** entity)
  - **DoctorID** (Foreign Key referring to the **Doctor** entity)
  - **PatientID** (Foreign Key referring to the **Patient** entity)

1.2.8 Surgery - Equipment

The relationship between **Surgery** and **Equipment** links each surgery to the equipment used during the procedure.

- **Denoted by:** Use In Surgery
- **Relationship's type:** Each surgery can use many equipments, and each equipment can be used for many surgeries. Thus, this relationship is an **N:M** relationship.
- **Implementation:** A junction table is introduced with the following group of 2 Primary Keys:
  - **SurgeryID** (Foreign Key referring to the **Surgery** entity)
  - **EquipID** (Foreign Key referring to the **Equipment** entity)

1.2.9 Diagnostic Test - Nurse - Patient

The relationship among **Diagnostic Test**, **Nurse**, and **Patient** links each diagnostic test to the nurse performing it and the patient undergoing the test.

- **Denoted by:** Perform_Test
- **Relationship's type:** This is a **ternary** relationship of 3 entities.
- **Implementation:** A junction table named Perform_Test is introduced with the following group of **3 Primary Keys**:
  - **TestID** (Foreign Key referring to the **Diagnostic Test** entity)
  - **NurseID** (Foreign Key referring to the **Nurse** entity)
  - **PatientID** (Foreign Key referring to the **Patient** entity)

1.2.10 Diagnostic Test - Equipment

The relationship between **Diagnostic Test** and **Equipment** links each diagnostic test to the equipment used to perform it.

- **Denoted by:** Use_In_Test
- **Relationship's type:** Each diagnostic test can use many equipments, and each equipment can be used for many tests. Thus, this relationship is an **N:M** relationship.
- **Implementation:** A junction table named Use_In_Test is introduced with the following group of **2 Primary Keys**:
  - **TestID**: Foreign Key referring to the **Diagnostic Test** entity
  - **EquipID**: Foreign Key referring to the **Equipment** entity

1.2.11 Employee - Department

The relationship between **Department** and **Employee** links each employee to the department in which they work.

- **Denoted by:** Work_in
- **Relationship's type:** Each department can have multiple employees, but each employee belongs to only one department. Thus, this relationship is a **1:N**relationship.
- **Implementation:** An attribute named **Department_Number** is added to the **Employee** entity as a **Foreign Key** referring to **Department** entity

1.2.12 Manager - Department

The relationship between **Department** and **Employee** links each Manager to the department they manage.

- **Denoted by:** Manage
- **Relationship's type:** 1 manager manages 1 Department. Thus, this relationship is a **1:1**relationship.
- **Implementation:** An attribute named **Manager_ID** is added to the **Department** entity as a **Foreign Key** referring to **Employee** entity

1.2.13 Department - Room

The relationship between **Department** and **Room** links each room to the department containing it.

- **Denoted by:** Contains
- **Relationship's type:** Each department can contain multiple rooms, but each room is assigned to only one department. Thus, this relationship is a **1:N**relationship.
- **Implementation:** An attribute named **DepartmentID** is added to the **Room** entity as a **Foreign Key** referring to **DepartmentID** (Primary Key of the Department entity).

1.3 Weak entities

1.3.1 Allergies

The **Allergies** entity represents the allergies of each patient and is considered a weak entity because its existence is dependent on the **Patient** entity. Its attributes are a composition of 2 Primary Key:

- **PatientID** (Foreign Key referring to the **Patient** entity)
- **Allergy** of Patients

1.3.2 Medical History

The **Medical History** entity represents the past medical conditions of each patient and is considered a weak entity because its existence is dependent on the **Patient** entity. It consists a composition of **2 Primary Keys**:

- **PatientID** (Foreign Key referring to the **Patient** entity)
- **Type**: Type of medical issue
- **Description**
- **Treatment** (Details of the treatments received)
- **Stage** : Severity of the issue

1.3.3 Bed

The Bed entity represents individual beds within hospital rooms and is considered a weak entity because its existence is dependent on the Room entity.

1.4 Key attributes of entities and their description

1.4.1 Patient record

- **Key attribute:** PatientID
- **Description**:A unique identifier assigned to each patient's medical record in the hospital system. This ID ensures that each patient's record is distinguishable from others, allowing accurate tracking of medical history, current medications, assigned doctors and nurses, and other related medical information.

1.4.2 Employee

- **Key attribute:** EmployeeID
- **Description**:A unique identifier assigned to each employee working in the hospital. This ID distinguishes each employee from others, ensuring proper management of roles, assignments, and responsibilities within the hospital system.

1.4.3 Department

- **Key attribute:** DepartmentID
- **Description**:A unique identifier assigned to each department within the hospital. This ID ensures that each department, such as Cardiology, Radiology, or Pediatrics, is uniquely distinguishable from others, allowing for proper organization, management, and assignment of resources, staff, and facilities within the hospital.

1.4.4 Room

- **Key attribute:** RoomID
- **Description**:A unique identifier assigned to each room in the hospital. This ID ensures that each room is distinguishable from all others, allowing for precise management and allocation of rooms within the facility.

1.4.5 Equipment

- **Key attribute:** EquipmentID
- **Description**:A unique identifier assigned to each piece of equipment used in the hospital. This ID ensures that each equipment item, such as MRI machines, ventilators, or surgical tools, is distinguishable from others, enabling accurate tracking, maintenance, and assignment to surgeries or diagnostic tests.

1.4.6 Surgery

- **Key attribute:** SurgeryID
- **Description**: unique identifier assigned to each surgical procedure performed in the hospital. This ID allows for precise tracking and documentation of each surgery, including details about the type of surgery, the date it was performed, and any associated complications or outcomes.

1.4.7 Diagnostic Test

- **Key attribute:** TestID
- **Description**: A unique identifier assigned to each diagnostic test conducted within the hospital. This ID ensures that every test, such as blood tests, X-rays, or MRIs, can be distinctly recognized and tracked in the medical records system.

1.4.8 Insurance

- **Key attribute:** InsuranceID
- **Description**: A unique identifier assigned to each insurance policy offered by the hospital. This ID enables the system to distinctly recognize and manage different insurance plans associated with patients.

1.4.9 Billing

- **Key attribute:** BillingID
- **Description**: A unique identifier assigned to each billing record generated for services rendered to a patient. This ID allows the hospital to track and manage individual billing statements, payments, and insurance claims accurately.

1.4.10 Payment

- **Key attribute:** ReceiptID
- **Description**: A unique identifier assigned to each payment transaction made by a patient or guarantor. This ID enables the hospital to accurately track and manage all payment activities, ensuring that each transaction can be referenced and audited as needed.

1.4.11 Allergies

- **Key attribute:** Allergy (partial key)
- **Description**: A descriptive label or name for the specific allergy that a patient has. This attribute serves as a partial key that, when combined with the RecordID from the Patient_Record entity, uniquely identifies the allergy record for a patient. This attribute is crucial for understanding patient sensitivities and managing their healthcare effectively, as it informs healthcare providers of any allergies that may impact treatment decisions and medication administration.

1.4.12 Medical History

- **Key attribute:** Type (partial key)
- **Description**: This partial key represents the category of the medical history entry. It distinguishes different medical history records for a single patient, working with RecordID to uniquely identify each entry in a patient's medical history.

1.5 Identify constraint

1.5.1 Uniqueness constraint

Each entity in our database all has the primary key to uniquely identify among instances of that entity. The key attribute of each entity types is listed in the previous section.

1.5.2 Cardinality & participation constraint

1.5.2.1 Patient - Billing

- **Cardinality constraint**: 1 - N (Billing at the N-side). As one patient can have many billing (he might have several surgeries as well as several tests, and each give him a bill). However, one bill can just aim at one patient. There is no such thing as two patient sharing one bill.

- **Participation constraint**: The Billing entity has the **total participation**, as each bill has to associate with at least and at most one patient. Whereas the patient do not have the **total participation**, as they can have no test and surgery at all (or they not yet have).

1.5.2.2 Patient - Insurance

- **Cardinality constraint**: 1 - N (Insurance at the N-side). As one patient can have many insurances. However, one insurance can just aim at one patient. There is no such thing as two insurance supporting two patient.
- **Participation constraint**:No side has the **total participation**, as each patient can have no insurance at all, and each insurance can support no patient.

1.5.2.3 Insurance - Billing

- **Cardinality constraint**: M - N. As one Insurance instance can cover many Billing instance, as long as the amount does not exceed the limit. Moreover, one Billing instance can also be covered by many Insurance instances, according to the Priority attribute.
- **Participation constraint**:No side has the **total participation**, as the patient might not buy any Insurance. In this case, the Billing entity does not have any corresponding Insurance instance to associate with. The same thing happens when the Patient has bought Insurance but there is no billing, as the Patient has not taken the surgeries or Test.

1.5.2.4 Payment - Billing

- **Cardinality constraint**: 1 - N (Payment is at the N-side). As one Billing can have many payments (the bill may be too large, it has to be splitted). However, one Payment can only aim at one Billing.
- **Participation constraint**: Only the Payment has the **total participation**, as one Payment must associate with at least and at most one Billing. On the other side, one Billing can have no Payment associated with it, because the Patient has not paid yet.

1.5.2.5 Doctor - Patient

- **Cardinality constraint**: 1 - N (Patient Record is at the N-side). One doctor can be assigned to many patients, while one patient can only be treated by one doctor.
- **Participation constraint**: No sides have the **total participation**. The doctor can be assigned to no patients, and the patient may have no doctors treated (only when he needs to do surgery, a doctor will be assigned and track their health).

1.5.2.6 Nurse - Patient

- **Cardinality constraint**: 1 - N (Patient Record is at the N-side). One nurse can be assigned to many patients, while one patient can only be treated by one nurse.
- **Participation constraint**: Only the Patient record entity has the **total participation**. As one patient must have one nurse to track his/her health and do diagnostic and medical test.

1.5.2.7 Technician - Equipment

- **Cardinality constraint**: M - N. One Equipment can be maintained by many technicians. On the other side, one technician can maintain many equipment.
- **Participation constraint**: No sides has the **total participation**. As the technicians can be assigned no equipment. Moreover, under some specific conditions, the equipment is not maintained by any technician.

1.5.2.8 Surgery - Doctor - Patient

- **Cardinality constraint**: M - N - 1. (The 1-side is the Patient). As for a combination of (Doctor, Surgery) instance, there is just one Patient can associate. On the other side, for a combination of (Doctor, Patient) instance, many Surgeries can associate, because this patient may need to undergo several Surgeries. With the combination of (Patient, Surgery), many Doctor can associate, because there can be many Doctor in a Surgery room.

1.5.2.9 Surgery - Equipment

- **Cardinality constraint**: M - N. One equipment can be used in many surgeries, as long as it is cleaned and repaired carefully. On the other side, one Surgery can involve many equipments.
- **Participation constraint**: Only the Surgery side has the **total participation**, because a surgery must have equipments, otherwise the Doctors cannot have the essential tools.

1.5.2.10 Diagnostic Test - Nurse - Patient

- **Cardinality constraint**: M - N - 1. (The 1-side is the Patient). As for a combination of (Nurse, Test) instance, there is just one Patient can associate. On the other side, for a combination of (Nurse, Patient) instance, many Tests can associate, because this patient may need to undergo several Tests. With the combination of (Patient, Test), many Nurses can associate, because some tests may be hard for one nurses to perform.

1.5.2.11 Diagnostic Test - Equipment

- **Cardinality constraint**: M - N. One equipment can be used in many diagnostic test, as long as it is cleaned and repaired carefully. On the other side, one Test can involve many equipments.
- **Participation constraint**: Only the Diagnostic Test side has the **total participation**, because a test must have equipments, otherwise the Nurses cannot have the essential tools.

1.5.2.12 Employee - Department

- **Cardinality constraint**: 1 - N (The Employee is at the N-side). A department can have many employees, but one employee can only work for one department.
- **Participation constraint**: Both side have the **total participation**. As a Department must have at least one employee, and an Employee must work in at least and at most 1 Department.

1.5.2.13 Manager - Department

- **Cardinality constraint**: 1 - 1. One employee can only be the manager of a Department. On the other side, one Department just need 1 employee to be the Manager.
- **Participation constraint**: Only the side of Department has the **total participation**. Because each Department must have one manager. On the other side, not every employee can be the manager.

1.5.2.14 Room - Department

- **Cardinality constraint**: 1 - N (Room is at the N-side). One department can have many rooms. On the other side, one room can just be contained in one department.
- **Participation constraint**: Only the side of the Department has the **total participation**. A department must have at least one room. However a room can belong to no Department (one example is the emergency room).

1.5.2.15 Room - Bed (Identity relationship)

- **Cardinality constraint**: 1 - N (Bed is at the N-side). One room can have many beds. On the other side, a bed is only contained in one room.
- **Participation constraint**: Only the side of the Bed has the **total participation**. We can have a room with no beds for patients. (ex:Administrative Office). However, a bed must be in some room.

1.5.2.16 Patient - Allergies (Identity relationship)

- **Cardinality constraint**: 1 - N (Allergies is at the N-side). One patient can have many Allergies history. However, an Allergies history of a patient can only associate with that patient.
- **Participation constraint**: Only the side of the Allergies has the **total participation**, as this is the identity relationship.

1.5.2.17 Patient - Medical History (Identity relationship)

- **Cardinality constraint**: 1 - N (Medical History is at the N-side). One patient can have many Medical history. However, a Medical history of a patient can only associate with that patient.
- **Participation constraint**: Only the side of the Medical history has the **total participation**, as this is the identity relationship.

1.6 Entity-relationship diagram (ERD)

Below is the picture of our hospital database's ERD. If you want to view it more detail, click here. This will go to the draw.io file where the diagram is located. Be sure to go to the "diagram" sheet.

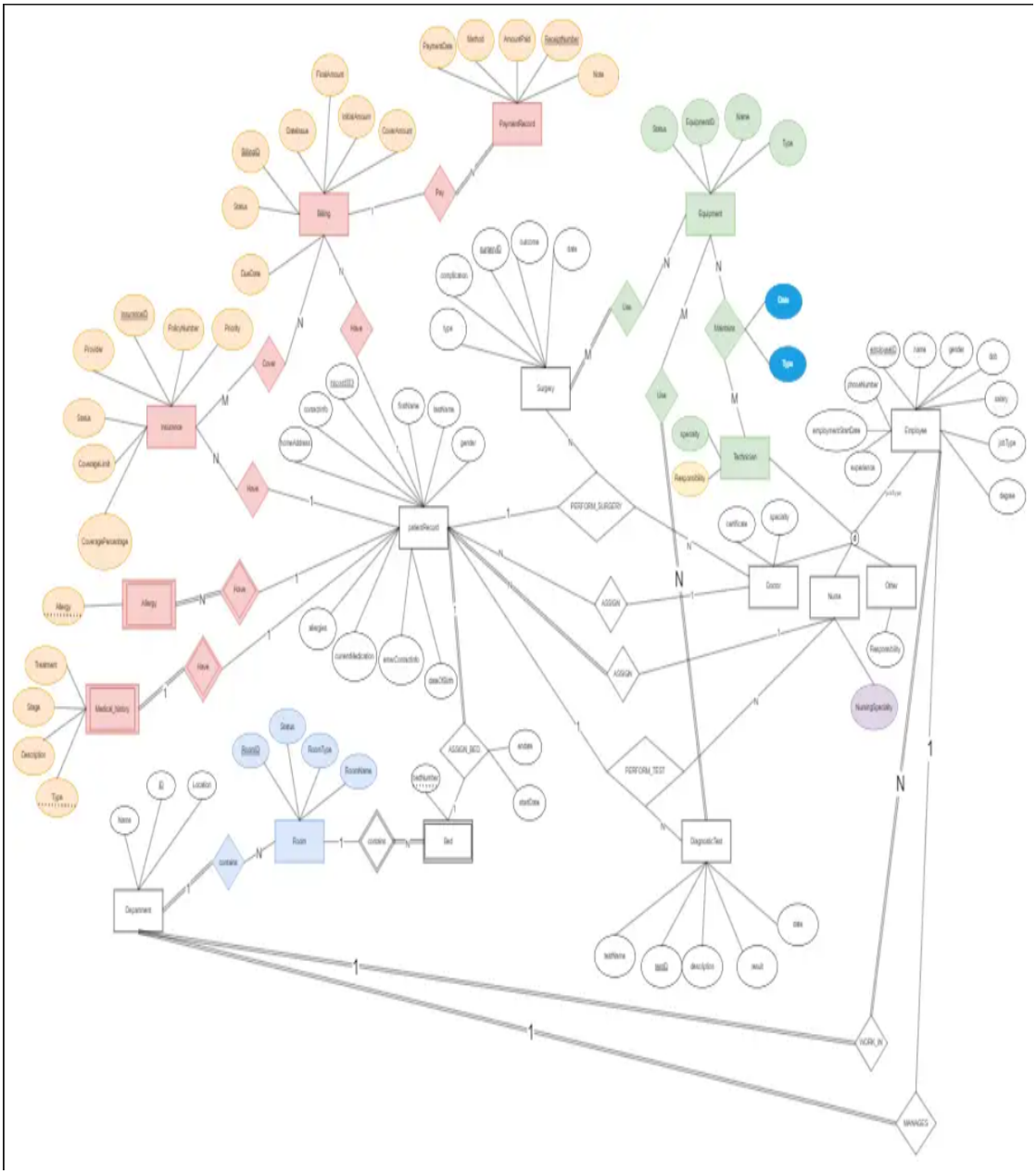


Figure 1.1: ERD for the hospital database

1.7 Mapping to relational schema specifying constraint

Below is the picture of our hospital database's relational schema. If you want to view it more detail, click here. This will go to the draw.io file where the schema is located. Be sure to go to the "mapping" sheet.

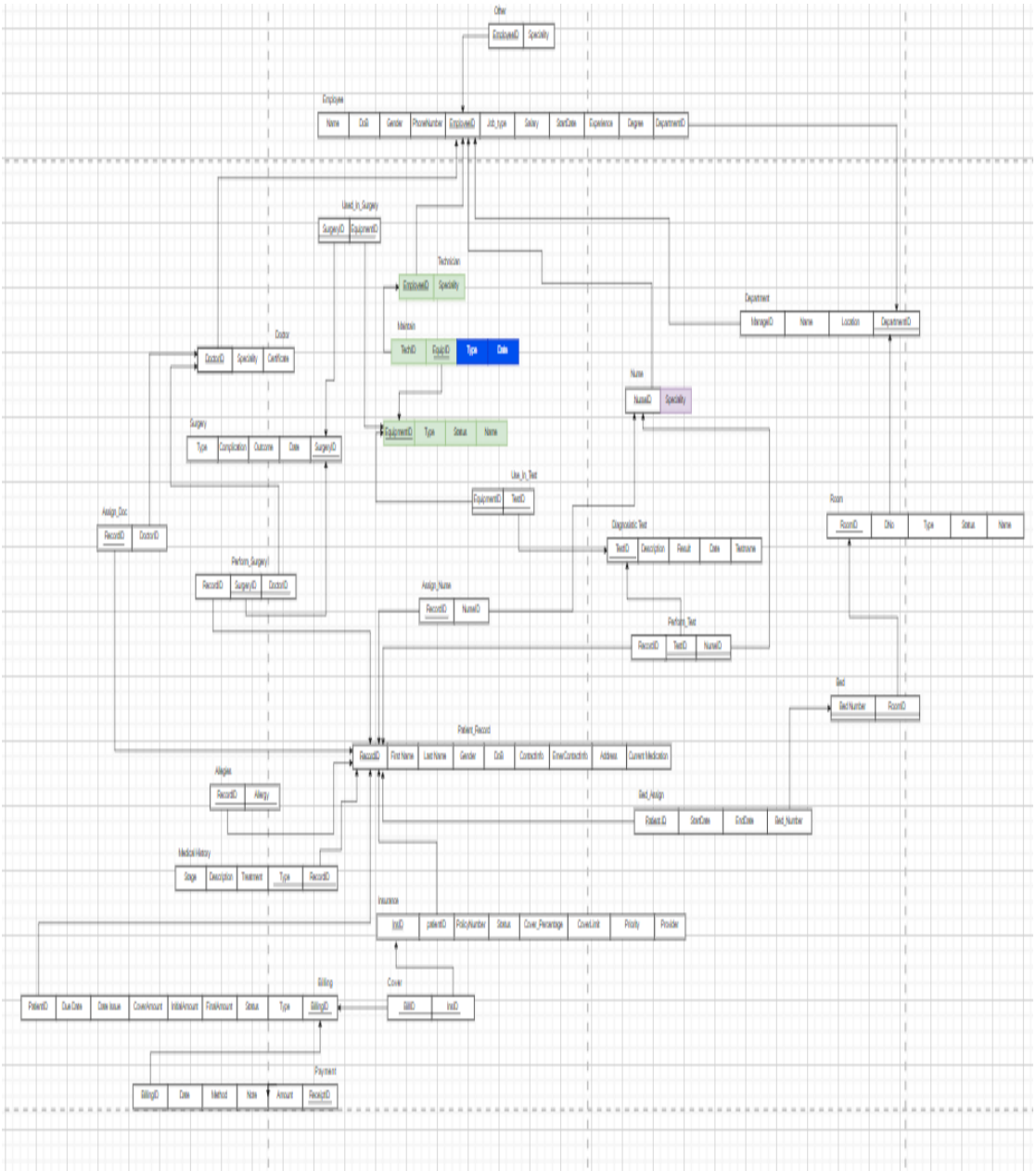


Figure 1.2: Relational schema for the hospital database

1.8 Specific Definitions of the Constraints

Entities	Attribute	Description / Constraint
EMPLOYEE	EmployeeID	PRIMARY KEY, IDENTITY(1,1), uniquely identifies each employee
	Name	NOT NULL, every employee must have a name
	Dob	NOT NULL, date of birth of the employee
	Gender	CHECK (Gender IN ('M', 'F', 'O')), limits gender values
	PhoneNumber	NOT NULL, UNIQUE, ensures no two employees share the same number
	DepartmentID	FOREIGN KEY references DEPARTMENT(DepartmentNumber), ON DELETE NO ACTION, ON UPDATE NO ACTION
DEPARTMENT	DepartmentNumber	PRIMARY KEY, IDENTITY(1,1), unique department identifier
	ManageID	FOREIGN KEY references EMPLOYEE(EmployeeID), ON DELETE SET NULL, ON UPDATE CASCADE
TECHNICIAN	EmployeeID	PRIMARY KEY, FOREIGN KEY references EMPLOYEE(EmployeeID)
	Specialty	Defines technician specialty
DOCTOR	DoctorID	PRIMARY KEY, FOREIGN KEY references EMPLOYEE(EmployeeID)
	Specialty	Doctor's area of expertise
	Certification	Doctor's certifications
OTHER_EMPLOYEE	EmployeeID	PRIMARY KEY, FOREIGN KEY references EMPLOYEE(EmployeeID)
	Specialty	Additional employees' specialization
NURSE	NurseID	PRIMARY KEY, FOREIGN KEY references EMPLOYEE(EmployeeID)
	Specialty	Nurse's specialization
ROOM	RoomID	PRIMARY KEY
	Dno	FOREIGN KEY references DEPARTMENT(DepartmentNumber), ON DELETE NO ACTION, ON UPDATE NO ACTION
BED	BedNumber, RoomID	Composite PRIMARY KEY (BedNumber, RoomID), FOREIGN KEY references ROOM(RoomID), ON DELETE CASCADE

Entities	Attribute	Description / Constraint
SURGERY	SurgeryID	PRIMARY KEY
EQUIPMENT	EquipmentID	PRIMARY KEY
MAINTAIN	TechID, EquipID	Composite PRIMARY KEY (TechID, EquipID), FOREIGN KEY references TECHNICIAN(EmployeeID), FOREIGN KEY references EQUIPMENT(EquipmentID)
USED_IN_SURGERY	SurgeryID, EquipmentID	Composite PRIMARY KEY (SurgeryID, EquipmentID), FOREIGN KEY references SURGERY(SurgeryID), FOREIGN KEY references EQUIPMENT(EquipmentID)
DIAGNOSTIC_TEST	TestID	PRIMARY KEY
USED_IN_TEST	TestID, EquipmentID	Composite PRIMARY KEY (TestID, EquipmentID), FOREIGN KEY references DIAGNOSTIC_TEST(TestID), FOREIGN KEY references EQUIPMENT(EquipmentID)
PATIENT_RECORD	RecordID	PRIMARY KEY
ASSIGN_NURSE	RecordID, NurseID	PRIMARY KEY, FOREIGN KEY references NURSE(NurseID), FOREIGN KEY references PATIENT_RECORD(RecordID)
ASSIGN_DOCTOR	RecordID, DoctorID	PRIMARY KEY, FOREIGN KEY references DOCTOR(DoctorID), FOREIGN KEY references PATIENT_RECORD(RecordID)
PERFORM_SURGERY	SurgeryID, DoctorID	Composite PRIMARY KEY (SurgeryID, DoctorID), FOREIGN KEY references SURGERY(SurgeryID), FOREIGN KEY references DOCTOR(DoctorID)
PERFORM_TEST	TestID, NurseID	Composite PRIMARY KEY (TestID, NurseID), FOREIGN KEY references DIAGNOSTIC_TEST(TestID), FOREIGN KEY references NURSE(NurseID)
ASSIGN_BED	PatientID	PRIMARY KEY, FOREIGN KEY references PATIENT_RECORD(RecordID), FOREIGN KEY (BedNumber, RoomID) references BED
ALLERGY	RecordID, Allergy	Composite PRIMARY KEY (RecordID, Allergy), FOREIGN KEY references PATIENT_RECORD(RecordID)
MEDICAL_HISTORY	RecordID, TypeName	Composite PRIMARY KEY (RecordID, TypeName), FOREIGN KEY references PATIENT_RECORD(RecordID)

Entities	Attribute	Description / Constraint
BILLING	BillingID	PRIMARY KEY
INSURANCE	InsuranceID	PRIMARY KEY
HAVE_INSURANCE	InsuranceID	PRIMARY KEY, FOREIGN KEY references INSURANCE(InsuranceID), FOREIGN KEY references PATIENT_RECORD(RecordID)
COVER	BillingID, InsuranceID	Composite PRIMARY KEY (BillingID, InsuranceID), FOREIGN KEY references BILLING(BillingID), FOREIGN KEY references INSURANCE(InsuranceID)
PAYMENT	ReceiptID	PRIMARY KEY, FOREIGN KEY references BILLING(BillingID)

1.9 Database Management system

Currently, there are many DBMS available as opens sources such as Oracle, MS SQL Server, MySQL, PostgreSQL, etc. While each offer various features, this Hospital Database will be implemented using **MS SQL Server** . Choosing MS SQL Server offers several practical benefits, especially for managing sensitive and essential healthcare data. Here are some reasons why it's a solid choice:

1. **Strong Security**: MS SQL Server provides features that protect data, such as encryption and advanced security settings. This is crucial in healthcare, where protecting patient data is a top priority.
2. **Reliability and Availability**: MS SQL Server includes options like backup systems and failover capabilities, which ensure the database remains accessible even if issues arise. This reliability is essential for hospitals, which operate around the clock.
3. **Efficient Performance**: MS SQL Server's indexing and query optimization features allow it to handle complex searches and large amounts of data smoothly. This helps retrieve patient information quickly, even as records grow over time.
4. **Scalability**: MS SQL Server can easily grow with the hospital's needs. It works well with both small and large databases, making it a good choice as the hospital expands its services.
5. **Ease of Use and Management**: MS SQL Server integrates well with other Microsoft tools, such as Excel and Power BI, making it easy for staff to work with the data. Its management tools are also user-friendly, which helps database administrators manage and maintain the system efficiently.
6. **Extensive Support and Resources**: Microsoft provides excellent support, resources, and documentation, which can help the hospital's IT team troubleshoot and maintain the database system with ease.

In short, MS SQL Server is secure, reliable, and adaptable, making it a suitable choice for managing healthcare data effectively. However, other than those basic features, MS SQL Server offers other unique features that not a lot of other DBMS have to offer.

Stand-Out Features of MS SQL Server

1. **Seamless Integration with Microsoft Ecosystem**: MS SQL Server's close integration with Microsoft tools like Azure, Power BI, Excel, and SharePoint makes it an ideal choice for businesses already using Microsoft products. This integration allows data to flow easily across platforms for real-time analytics, visualization, and reporting.

2. **Enhanced Security Compared to Other DBMSs**: MS SQL Server stands out with its robust security offerings, such as Transparent Data Encryption and Always Encrypted, which are often more comprehensive than what is standard in many other DBMSs. These features provide data privacy and protection.

3. **Built-in BI Tools**: Unlike many DBMSs, SQL Server includes Business Intelligence tools directly within the system. SQL Server Reporting Services (SSRS), SQL Server Integration Services (SSIS), and SQL Server Analysis Services (SSAS) allow businesses to generate insights and make data-driven decisions without requiring external BI solutions.

4. **Efficient and Simplified Database Management**: With SQL Server Management Studio (SSMS), MS SQL Server offers one of the most accessible and intuitive management interfaces available. SSMS's tools for backup, recovery, and system monitoring make it a strong choice for efficient, centralized database administration.

5. **Streamlined Licensing and Cost Management**: MS SQL Server's free Express edition yet still provides necessary features for the Hospital Database.

In short, thanks to those significant traits and its well-known reliability, MS SQL Server is undoubtedly an excellent choice for this Database

1.10 Define relational schema in DBMS. Define user groups and permission.

1.10.1 Define relational schema in DBMS

We implemented the relational schema above to the DBMS. We use the DDL language to create tables. Below is the code that initialize the HospitalDb:

```
    CREATE DATABASE HospitalDB; -- run this command first
    -- then run these command later.
    USE HospitalDB;


IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'EMPLOYEE'
)
BEGIN
 CREATE TABLE EMPLOYEE (
  EmployeeID INT PRIMARY KEY IDENTITY(1,1),
  Name VARCHAR(100) NOT NULL,
  Dob DATE NOT NULL,
  Gender CHAR(1) CHECK (Gender IN ('M', 'F', 'O')),
```

```
  PhoneNumber VARCHAR(15) NOT NULL UNIQUE,
  Job_type VARCHAR(50) NOT NULL,
  Salary DECIMAL(10, 2) NOT NULL,
  startDate DATE NOT NULL,
  Experience INT,
  Degree VARCHAR(100),
 );
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'DEPARTMENT'
)
BEGIN
 CREATE TABLE DEPARTMENT (
  DepartmentNumber INT PRIMARY KEY IDENTITY(1,1),
  ManageID INT,
  Name VARCHAR(100) NOT NULL,
  Location VARCHAR(100),
  CONSTRAINT FK_Department_ManageID
   FOREIGN KEY (ManageID) REFERENCES EMPLOYEE(EmployeeID)
   ON DELETE SET NULL
   ON UPDATE CASCADE
 );
END

ALTER TABLE EMPLOYEE
ADD DepartmentID INT


ALTER TABLE EMPLOYEE
ADD CONSTRAINT FK_Employees_Department
FOREIGN KEY (DepartmentID) REFERENCES DEPARTMENT(DepartmentNumber)
ON DELETE NO ACTION
ON UPDATE NO ACTION;


IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'TECHNICIAN'
)
BEGIN
 CREATE TABLE TECHNICIAN (
    EmployeeID INT PRIMARY KEY,
    Specialty NVARCHAR(100),
    CONSTRAINT FK_Technician_Employee
        FOREIGN KEY (EmployeeID) REFERENCES EMPLOYEE(EmployeeID)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
```

```
 );
END

GO
IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'DOCTOR'
)
BEGIN
 CREATE TABLE DOCTOR (
    DoctorID INT PRIMARY KEY,
    Specialty NVARCHAR(100),
    Certifica NVARCHAR(100),
    CONSTRAINT FK_Doctor_Employee
        FOREIGN KEY (DoctorID) REFERENCES EMPLOYEE(EmployeeID)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
 );
END

GO

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'OTHER_EMPLOYEE'
)
BEGIN
 CREATE TABLE OTHER_EMPLOYEE (
    EmployeeID INT PRIMARY KEY,
    Specialty VARCHAR(100),
    FOREIGN KEY (EmployeeID) REFERENCES EMPLOYEE(EmployeeID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'NURSE'
)
BEGIN
 CREATE TABLE NURSE (
    NurseID INT PRIMARY KEY,
    Specialty NVARCHAR(100),
    CONSTRAINT FK_Nurse_Employee
        FOREIGN KEY (NurseID) REFERENCES EMPLOYEE(EmployeeID)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
 );
END
```

```
GO
IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'ROOM'
)
BEGIN
 CREATE TABLE ROOM (
    RoomID INT PRIMARY KEY,
    Dno INT,
    TypeRoom NVARCHAR(100),
    StatusRoom NVARCHAR(100),
    CONSTRAINT FK_Room_Department
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(DepartmentNumber)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'BED'
)
BEGIN
 CREATE TABLE BED (
    BedNumber INT NOT NULL,
    RoomID INT NOT NULL,
    PRIMARY KEY (BedNumber, RoomID),
    CONSTRAINT FK_Bed_Room
        FOREIGN KEY (RoomID) REFERENCES ROOM(RoomID)
        ON DELETE CASCADE
        ON UPDATE NO ACTION
 );
END


IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'SURGERY'
)
BEGIN
 CREATE TABLE SURGERY (
    SurgeryID INT PRIMARY KEY,
    TypeSurgery NVARCHAR(100),
    Complication NVARCHAR(100),
    Outcome NVARCHAR(100),
    DateSurgery DATE
);
END
```

```
IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'EQUIPMENT'
)
BEGIN
 CREATE TABLE EQUIPMENT (
    EquipmentID INT PRIMARY KEY,
    Type NVARCHAR(100) NOT NULL,
    Status NVARCHAR(50) NOT NULL,
    Name NVARCHAR(100) NOT NULL
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'MAINTAIN'
)
BEGIN
 CREATE TABLE MAINTAIN (
    TechID INT,
    EquipID INT,
    TypeMaintain NVARCHAR(100) NOT NULL,
    DateMaintain DATE NOT NULL,
    PRIMARY KEY (TechID, EquipID),
    FOREIGN KEY (TechID) REFERENCES TECHNICIAN(EmployeeID),
    FOREIGN KEY (EquipID) REFERENCES EQUIPMENT(EquipmentID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'USED_IN_SURGERY'
)
BEGIN
 CREATE TABLE USED_IN_SURGERY (
    SurgeryID INT,
    EquipmentID INT,
    PRIMARY KEY (SurgeryID, EquipmentID),
    FOREIGN KEY (SurgeryID) REFERENCES SURGERY(SurgeryID),
    FOREIGN KEY (EquipmentID) REFERENCES EQUIPMENT(EquipmentID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
```

```
    WHERE name = 'DIAGNOSTIC_TEST'
)
BEGIN
 CREATE TABLE DIAGNOSTIC_TEST (
    TestID INT PRIMARY KEY,
    TestName NVARCHAR(100) NOT NULL,
    TestDescription NVARCHAR(255),
    TestDate DATE NOT NULL,
    TestResult NVARCHAR(100)
);

END



IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'USED_IN_TEST'
)
BEGIN
 CREATE TABLE USED_IN_TEST (
    TestID INT,
    EquipmentID INT,
    PRIMARY KEY (TestID, EquipmentID),
    FOREIGN KEY (TestID) REFERENCES DIAGNOSTIC_TEST(TestID),
    FOREIGN KEY (EquipmentID) REFERENCES EQUIPMENT(EquipmentID)
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'PATIENT_RECORD'
)
BEGIN
 CREATE TABLE PATIENT_RECORD (
    RecordID INT PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    Gender NVARCHAR(10) NOT NULL,
    ContactInfo NVARCHAR(255) NOT NULL,
    EmerContactInfo NVARCHAR(255),
    Address NVARCHAR(255),
    CurrentMedication NVARCHAR(255)
);

END


IF NOT EXISTS (
```

```
    SELECT * FROM sys.tables
    WHERE name = 'ASSIGN_NURSE'
)
BEGIN
 CREATE TABLE ASSIGN_NURSE (
    NurseID INT,
    RecordID INT,
    PRIMARY KEY (RecordID),
    FOREIGN KEY (NurseID) REFERENCES NURSE(NurseID),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'ASSIGN_DOCTOR'
)
BEGIN
 CREATE TABLE ASSIGN_DOCTOR (
    DoctorID INT,
    RecordID INT,
    PRIMARY KEY (RecordID),
    FOREIGN KEY (DoctorID) REFERENCES DOCTOR(DoctorID),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'PERFORM_SURGERY'
)
BEGIN
 CREATE TABLE PERFORM_SURGERY (
    SurgeryID INT,
    DoctorID INT,
    RecordID INT,

    PRIMARY KEY (SurgeryID, DoctorID),
    FOREIGN KEY (SurgeryID) REFERENCES SURGERY(SurgeryID),
    FOREIGN KEY (DoctorID) REFERENCES DOCTOR(DoctorID),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
```

```
    WHERE name = 'PERFORM_TEST'
)
BEGIN
 CREATE TABLE PERFORM_TEST (
    TestID INT,
    NurseID INT,
    RecordID INT,
    PRIMARY KEY (TestID, NurseID),
    FOREIGN KEY (TestID) REFERENCES DIAGNOSTIC_TEST(TestID),
    FOREIGN KEY (NurseID) REFERENCES NURSE(NurseID),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID)
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'PERFORM_TEST'
)
BEGIN
 CREATE TABLE PERFORM_TEST (
    RecordID INT,
    TestID INT,
    NurseID INT,
    PRIMARY KEY (RecordID, TestID, NurseID),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID),
    FOREIGN KEY (TestID) REFERENCES DIAGNOSTIC_TEST(TestID),
    FOREIGN KEY (NurseID) REFERENCES NURSE(NurseID)
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'ASSIGN_BED'
)
BEGIN
 CREATE TABLE ASSIGN_BED (
    PatientID INT,
    StartDate DATE,
    EndDate DATE,
    BedNumber INT,
    RoomID INT,
    PRIMARY KEY (PatientID),
    FOREIGN KEY (PatientID) REFERENCES PATIENT_RECORD(RecordID),
    FOREIGN KEY (BedNumber, RoomID) REFERENCES BED(BedNumber, RoomID)
);

END

IF NOT EXISTS (
```

```
    SELECT * FROM sys.tables
    WHERE name = 'ALLERGY'
)
BEGIN
 CREATE TABLE ALLERGY (
    RecordID INT,
    Allergy VARCHAR(255),
    PRIMARY KEY (RecordID, Allergy),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID)
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'MEDICAL_HISTORY'
)
BEGIN
 CREATE TABLE MEDICAL_HISTORY (
    RecordID INT,
    TypeName VARCHAR(255),
    Treatment VARCHAR(255),
    DescriptionDetail TEXT,
    Stage VARCHAR(50),
    PRIMARY KEY (RecordID, TypeName),
    FOREIGN KEY (RecordID) REFERENCES PATIENT_RECORD(RecordID)
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'BILLING'
)
BEGIN
 CREATE TABLE BILLING (
    BillingID INT PRIMARY KEY,
    TypeBill VARCHAR(255),
    StatusBill VARCHAR(50),
    InitialAmount DECIMAL(18, 2),
    CoverAmount DECIMAL(18, 2),
    FinalAmount DECIMAL(18, 2),
    DateIssued DATE,
    DueDate DATE
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'INSURANCE'
)
```

```
BEGIN
 CREATE TABLE INSURANCE (
    InsuranceID INT PRIMARY KEY,
    Provider VARCHAR(255),
    PolicyNumber VARCHAR(255),
    StatusInsurance VARCHAR(50),
    CoverPercentage DECIMAL(5, 2),
    CoverLimit DECIMAL(18, 2),
    InsurancePriority INT
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'HAVE_INSURANCE'
)
BEGIN
 CREATE TABLE HAVE_INSURANCE (
    InsuranceID INT PRIMARY KEY,
    PatientID INT,
    FOREIGN KEY (InsuranceID) REFERENCES INSURANCE(InsuranceID),
    FOREIGN KEY (PatientID) REFERENCES PATIENT_RECORD(RecordID)
);

END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'COVER'
)
BEGIN
 CREATE TABLE COVER (
    BillingID INT,
    InsuranceID INT,
    PRIMARY KEY (BillingID, InsuranceID),
    FOREIGN KEY (BillingID) REFERENCES BILLING(BillingID),
    FOREIGN KEY (InsuranceID) REFERENCES INSURANCE(InsuranceID)
);
END

IF NOT EXISTS (
    SELECT * FROM sys.tables
    WHERE name = 'PAYMENT'
)
BEGIN
 CREATE TABLE PAYMENT (
    ReceiptID INT PRIMARY KEY,
    Amount DECIMAL(18, 2),
    Note VARCHAR(255),
```

```
    Method VARCHAR(50),
    DatePay DATE,
    BillingID INT,
    FOREIGN KEY (BillingID) REFERENCES BILLING(BillingID)
);


END
```

1.10.2 Define user groups and permission

We decided to separate the users of our HospitalDb application into 7 groups, which are:

- AdminRole. This role is assigned for the administrator of the hospitalDb, usually in the Administrative Office.Users with AdminRole can grant all actions including SELECT, INSERT, UPDATE, DELETE on all tables of the relational schema.
- DoctorRole. This role is assign for employees who are doctors .Users with DoctorRole can:
  - Grant SELECT on tables: ALLERGY, ASSIGN_DOCTOR, DOCTOR, EMPLOYEE, MEDICAL_HISTORY, PATIENT_RECORD, PERFORM_SURGERY, USED_IN_SURGERY, DIAGNOSTIC_TEST, PERFORM_TEST, ROOM, BED, USED_IN_SURGERY, USED_IN_TEST
  - Grant INSERT, UPDATE, DELETE ON: MEDICAL_HISTORY, ALLERGY
- NurseRole. This role is assigned for employees who are nurses. Users with NurseRole can:
  - grant SELECT on tables: ALLERGY, ASSIGN_NURSE, DIAGNOSTIC_TEST, EMPLOYEE, MEDICAL_HISTORY, NURSE, PATIENT_RECORD, PERFORM_SURGERY, PERFORM_TEST, ROOM, BED, SURGERY, USED_IN_SURGERY, USED_IN_TEST - INSERT, UPDATE, DELETE: ALLERGY, MEDICAL_HISTORY
  - grant INSERT, DELETE, UPDATE on tables: EQUIPMENT, EMPLOYEE, MAINTAIN, TECHNICIAN.
- TechnicianRole. This role is assigned for employees who are technicians. Users with TechnicianRole can:
  - grant SELECT on tables: EQUIPMENT, EMPLOYEE, MAINTAIN, TECHNICIAN.
- RelativeRole. This role is assigned for people who are relatives to the patients. They can tracks the surgeries and tests that the patient undergoes, as well as the health status of the patient. Users with RelativeRole can:
  - grant SELECT on tables: ALLERGY, ASSIGN_BED, ASSIGN_NURSE, ASSGIN_DOCTOR, BILLING, COVER, DIAGNOSTIC_TEST, DOCTOR, EMPLOYEE, HAVE_INSURANCE, INSURANCE, MEDICAL_HISTORY, NURSE, PATIENT_RECORD, PAYMENT, PERFORM_SURGERY, PERFORM_TEST,BED, ROOM, SURGERY.
- ReceptionistRole.This role is assigned for employees who are receptionist. Users with ReceptionistRole can:

  - grant SELECT on tables: BILLING, COVER, DEPARTMENT, HAVE_INSURANCE, INSURANCE, PAYMENT, PATIENT_RECORD
  - grant INSERT, DELETE, UPDATE on tables: BILLING, COVER, DEPARTMENT, HAVE_INSURANCE, INSURANCE, PAYMENT

- ManagerDepartmentRole.This role is assigned for employees who are the manager of a department. Users with ManagerDepartmentROle can:
  - grante SELECT, INSERT, DELETE, UPDATE on tables: DEPARTMENT, ROOM, BED

Below is the SQL code to define user groups and user permissions in MS SQL server:

```
    USE HospitalDB;
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'AdminRole')
BEGIN
    CREATE ROLE AdminRole;
END

GRANT SELECT, INSERT, UPDATE, DELETE ON SCHEMA::dbo TO AdminRole;
---- //////////
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'DoctorRole')
BEGIN
    CREATE ROLE DoctorRole;
END

GRANT SELECT ON ALLERGY TO DoctorRole;
GRANT SELECT ON ASSIGN_DOCTOR TO DoctorRole;
GRANT SELECT ON DOCTOR TO DoctorRole;
GRANT SELECT ON EMPLOYEE TO DoctorRole;
GRANT SELECT ON MEDICAL_HISTORY TO DoctorRole;
GRANT SELECT ON PATIENT_RECORD TO DoctorRole;
GRANT SELECT ON PERFORM_SURGERY TO DoctorRole;
GRANT SELECT ON DIAGNOSTIC_TEST TO DoctorRole;
GRANT SELECT ON PERFORM_TEST TO DoctorRole;
GRANT SELECT ON ROOM TO DoctorRole;
GRANT SELECT ON BED TO DoctorRole;
GRANT SELECT ON USED_IN_SURGERY TO DoctorRole;
GRANT SELECT ON USED_IN_TEST TO DoctorRole;
GRANT INSERT, DELETE, UPDATE ON MEDICAL_HISTORY TO DoctorRole;
GRANT INSERT, DELETE, UPDATE ON ALLERGY TO DoctorRole;
---- //////////
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'NurseRole')
BEGIN
    CREATE ROLE NurseRole;
END

-- Grant SELECT permission on the specified tables to NurseRole
GRANT SELECT ON ALLERGY TO NurseRole;
GRANT SELECT ON ASSIGN_NURSE TO NurseRole;
```

```
GRANT SELECT ON DIAGNOSTIC_TEST TO NurseRole;
GRANT SELECT ON EMPLOYEE TO NurseRole;
GRANT SELECT ON MEDICAL_HISTORY TO NurseRole;
GRANT SELECT ON NURSE TO NurseRole;
GRANT SELECT ON PATIENT_RECORD TO NurseRole;
GRANT SELECT ON PERFORM_SURGERY TO NurseRole;
GRANT SELECT ON PERFORM_TEST TO NurseRole;
GRANT SELECT ON ROOM TO NurseRole;
GRANT SELECT ON BED TO NurseRole;
GRANT SELECT ON SURGERY TO NurseRole;
GRANT SELECT ON USED_IN_SURGERY TO NurseRole;
GRANT SELECT ON USED_IN_TEST TO NurseRole;
GRANT INSERT, UPDATE, DELETE ON ALLERGY TO NurseRole;
GRANT INSERT, UPDATE, DELETE ON MEDICAL_HISTORY TO NurseRole;
---- //////////
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'TechnicianRole')
BEGIN
    CREATE ROLE TechnicianRole;
END
GRANT SELECT ON EQUIPMENT TO TechnicianRole;
GRANT SELECT ON EMPLOYEE TO TechnicianRole;
GRANT SELECT ON MAINTAIN TO TechnicianRole;
GRANT SELECT ON TECHNICIAN TO TechnicianRole;
---- //////////
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'RelativeRole')
BEGIN
    CREATE ROLE RelativeRole;
END
GRANT SELECT ON ALLERGY TO RelativeRole;
GRANT SELECT ON ASSIGN_BED TO RelativeRole;
GRANT SELECT ON ASSIGN_NURSE TO RelativeRole;
GRANT SELECT ON ASSIGN_DOCTOR TO RelativeRole;
GRANT SELECT ON BILLING TO RelativeRole;
GRANT SELECT ON COVER TO RelativeRole;
GRANT SELECT ON DIAGNOSTIC_TEST TO RelativeRole;
GRANT SELECT ON DOCTOR TO RelativeRole;
GRANT SELECT ON EMPLOYEE TO RelativeRole;
GRANT SELECT ON HAVE_INSURANCE TO RelativeRole;
GRANT SELECT ON INSURANCE TO RelativeRole;
GRANT SELECT ON MEDICAL_HISTORY TO RelativeRole;
GRANT SELECT ON NURSE TO RelativeRole;
GRANT SELECT ON PATIENT_RECORD TO RelativeRole;
GRANT SELECT ON PAYMENT TO RelativeRole;
GRANT SELECT ON PERFORM_SURGERY TO RelativeRole;
GRANT SELECT ON PERFORM_TEST TO RelativeRole;
GRANT SELECT ON BED TO RelativeRole;
GRANT SELECT ON ROOM TO RelativeRole;
GRANT SELECT ON SURGERY TO RelativeRole;
```

```
---- //////////
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'ReceptionistRole')
BEGIN
    CREATE ROLE ReceptionistRole;
END
GRANT SELECT ON BILLING TO ReceptionistRole;
GRANT SELECT ON COVER TO ReceptionistRole;
GRANT SELECT ON DEPARTMENT TO ReceptionistRole;
GRANT SELECT ON HAVE_INSURANCE TO ReceptionistRole;
GRANT SELECT ON INSURANCE TO ReceptionistRole;
GRANT SELECT ON PAYMENT TO ReceptionistRole;
GRANT SELECT ON PATIENT_RECORD TO ReceptionistRole;

GRANT INSERT, DELETE, UPDATE ON BILLING TO ReceptionistRole;
GRANT INSERT, DELETE, UPDATE ON COVER TO ReceptionistRole;
GRANT INSERT, DELETE, UPDATE ON HAVE_INSURANCE TO ReceptionistRole;
GRANT INSERT, DELETE, UPDATE ON INSURANCE TO ReceptionistRole;
GRANT INSERT, DELETE, UPDATE ON PAYMENT TO ReceptionistRole;


---- //////////
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'ManagerDepartmentRole')
BEGIN
    CREATE ROLE ManagerDepartmentRole;
END
GRANT SELECT, INSERT, DELETE, UPDATE ON DEPARTMENT TO ManagerDepartmentRole;
GRANT SELECT, INSERT, DELETE, UPDATE ON ROOM TO ManagerDepartmentRole;
GRANT SELECT, INSERT, DELETE, UPDATE ON BED TO ManagerDepartmentRole;
```

1.11 Tools for Developing the database application (Front-end and Back-end

Using .NET for developing applications that interact with MS SQL Server databases offers numerous advantages, along with a suite of powerful tools to facilitate development. Here's an overview of the reasons to choose .NET for this purpose and the relevant tools you can leverage.

Why Use .NET for MS SQL Database Applications

1. **Seamless Integration**: .NET provides seamless integration with MS SQL Server. The Entity Framework (EF) and ADO.NET allow developers to easily connect to the database, execute queries, and manage data with minimal effort.

2. **Strong Language Support**: languages like C# and VB.NET supports type safety, object-oriented programming, and rich language features including libraries and documents.

3. **Robust Framework**: The .NET framework offers a vast class library, simplifying tasks such as data access, security, and application configuration.

4. **Cross-Platform Development**: With the newer version of .NET Core supporting Windows, Linux, and macOS, it is advantageous for deploying applications in diverse environments.

5. **Scalability and Performance**: .NET applications can scale efficiently, allowing developers to build applications that can handle increased loads seamlessly. Additionally, the asynchronous programming model in .NET enhances performance, especially for I/O-bound operations like database access.

6. **Security Features**: .NET offers built-in security features, including authentication, authorization, and data protection, helping to safeguard applications against common vulnerabilities.

7. **Community and Support**: The .NET ecosystem has a large community, providing access to extensive resources, documentation, and forums for troubleshooting and knowledge sharing.

Chapter 2

Complete the database from the schema and securing the database, and develop an application program using the finalized database on a DBMS.

2.1 Triggers and Functions

2.1.1 Introduction

This document describes SQL triggers and stored procedures designed for the `HospitalDB` database, which supports various automation processes within a hospital management system.

- **SQL Triggers**: Two triggers are implemented to automate billing processes and payment updates. These triggers ensure billing records are accurately updated when insurance coverage or payments are processed.
  - The `InsertCover` trigger calculates and applies insurance coverage amounts after a new billing record is inserted.
  - The `UpdatePay` trigger adjusts the final billing amount when a payment is recorded.
- **SQL Stored Procedures**: A set of stored procedures enable efficient assignment and reassignment of hospital resources. These procedures facilitate updating the assignments of doctors, nurses, beds, rooms, and employee details.
  - Procedures such as `change_doctor`, `change_nurse`, and `change_bed` handle the reassignment of medical staff and patient accommodations.
  - Procedures like `update_ex`, `change_tech`, `change_room`, and `change_depart` update employee experience, maintenance records, room assignments, and department transfers.

2.1.2 Trigger 1: InsertCover

The `InsertCover` trigger is executed after a new record is inserted into the `BILLING` table. This trigger calculates insurance coverage for a patient's bill based on available insurance policies, their limits, and coverage percentages.

2.1.2.1 Trigger Code

```
USE HospitalDB;

GO
CREATE OR ALTER TRIGGER InsertCover
ON BILLING
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @BillingID INT, @InsuranceID INT, @InitialAmount DECIMAL(18, 2),
    @FinalAmount DECIMAL(18, 2), @CoverAmount DECIMAL(18, 2),
    @CoverLimit DECIMAL(18, 2), @CoverPercentage DECIMAL(5, 2),
    @TotalCover DECIMAL(18, 2);

    SELECT
        @BillingID = BillingID,
        @InitialAmount = InitialAmount,
        @FinalAmount = InitialAmount,
        @TotalCover = CoverAmount
    FROM INSERTED;

    DECLARE BillCursor CURSOR FOR
    SELECT INS.InsuranceID, CoverLimit, CoverPercentage
    FROM HAVE_INSURANCE HI
    INNER JOIN INSURANCE INS ON HI.InsuranceID = INS.InsuranceID
    INNER JOIN INSERTED I ON HI.PatientID = I.PatientID
    WHERE CoverLimit > 0
    ORDER BY INS.InsurancePriority ASC;

    OPEN BillCursor;
    FETCH NEXT FROM BillCursor
    INTO @InsuranceID, @CoverLimit, @CoverPercentage;

    WHILE @@FETCH_STATUS = 0 AND @FinalAmount > 0
    BEGIN
        -- Calculate cover
        SET @CoverAmount = @InitialAmount * @CoverPercentage / 100;
        SET @CoverAmount = CASE
                           WHEN @CoverAmount > @CoverLimit THEN @CoverLimit
                           ELSE @CoverAmount
                           END;

        IF @CoverAmount > @FinalAmount
        BEGIN
```

```
            SET @CoverAmount = @FinalAmount;
        END

        SET @CoverLimit = @CoverLimit - @CoverAmount;
        SET @FinalAmount = @FinalAmount - @CoverAmount;

        INSERT INTO COVER (BillingID, InsuranceID) VALUES (@BillingID, @InsuranceID);

        SET @TotalCover = @TotalCover + @CoverAmount;

        -- Update insurance cover limit
        UPDATE INSURANCE
        SET CoverLimit = CoverLimit - @CoverAmount
        WHERE @InsuranceID = InsuranceID;

        FETCH NEXT FROM BillCursor
        INTO @InsuranceID, @CoverLimit, @CoverPercentage;
    END

    CLOSE BillCursor;
    DEALLOCATE BillCursor;

    -- Update billing table with final amounts
    UPDATE BILLING
    SET CoverAmount = CoverAmount + @TotalCover,
        FinalAmount = @FinalAmount
    WHERE BillingID = @BillingID;
END;
GO
```

Listing 2.1: `InsertCover` Trigger

2.1.2.2 Explanation

- The trigger captures billing information from the `INSERTED` pseudo-table.
- It uses a cursor to loop through available insurance policies, ordered by priority.
- For each insurance policy, the cover amount is calculated based on the percentage of coverage and the available limit.
- The final billing amount and the cover amount are updated accordingly.

2.1.3 Trigger 2: UpdatePay

The `UpdatePay` trigger is executed after a new record is inserted into the `PAYMENT` table. It reduces the final billing amount by the payment amount.

2.1.3.1 Trigger Code

```
USE HospitalDB;

GO
```

```sql
CREATE OR ALTER TRIGGER UpdatePay
ON PAYMENT
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    -- Update the FinalAmount in the BILLING table
    UPDATE B
    SET B.FinalAmount = B.FinalAmount - I.Amount
    FROM BILLING B
    INNER JOIN INSERTED I
        ON B.BillingID = I.BillingID;

    -- Debug: Check if rows were updated
    IF @@ROWCOUNT = 0
    BEGIN
        RAISERROR('No matching BillingID found in the BILLING table.', 16, 1);
    END
    ELSE
    BEGIN
        PRINT 'Billing record updated successfully.';
    END
END;
GO
```

Listing 2.2: `UpdatePay` Trigger

2.1.3.2 Explanation

- When a payment is inserted, the trigger updates the corresponding billing record by reducing the `FinalAmount` by the payment amount.
- If no matching `BillingID` is found, an error is raised.
- A success message is printed when the update is successful.

2.1.4 Procedure: `change_doctor`

This procedure updates the doctor assigned to a patient record.

2.1.4.1 Procedure Code

```sql
USE HospitalDB;

GO
CREATE OR ALTER PROCEDURE change_doctor
    @RecordID INT,
    @DoctorID INT
AS
BEGIN
    SET NOCOUNT ON;

```

```
    BEGIN TRY
        UPDATE Assign_Doctor
        SET DoctorID = @DoctorID
        WHERE RecordID = @RecordID;

        IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR('Record not found or doctor does not exist.', 16, 1);
        END
        ELSE
            PRINT 'Doctor assigned to patient record updated successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR('An error occurred: %s', 16, 1, @ErrorMessage);
    END CATCH
END;
GO
```

Listing 2.3: `change_doctor` Procedure

2.1.4.2 Explanation

The `change_doctor` procedure updates the `DoctorID` in the `Assign_Doctor` table for a given `RecordID`. Key steps:

- If no record matches the `RecordID`, a custom error is raised.
- On success, a confirmation message is printed.
- Robust error handling ensures meaningful error messages in case of failure.

2.1.5 Procedure: `change_bed`

This procedure changes the bed assigned to a patient.

2.1.5.1 Procedure Code

```
USE HospitalDB;

GO
CREATE OR ALTER PROCEDURE change_bed
    @PatientID INT,
    @BedNumber INT
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        UPDATE ASSIGN_BED
        SET BedNumber = @BedNumber
        WHERE PatientID = @PatientID;

        IF @@ROWCOUNT = 0
```

```
        BEGIN
            RAISERROR('Record not found or bed does not exist.', 16, 1);
        END
        ELSE
            PRINT 'Bed assignment updated successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR('An error occurred: %s', 16, 1, @ErrorMessage);
    END CATCH
END;
GO
```

Listing 2.4: `change_bed` Procedure

2.1.5.2 Explanation

The `change_bed` procedure updates the `BedNumber` in the `ASSIGN_BED` table for a specific `PatientID`. Key steps:

- Verifies if a patient record exists using `@@ROWCOUNT`.
- Raises a custom error if no record is found.
- Implements robust error handling for meaningful error messages.

2.1.6 Procedure: `change_nurse`

This procedure updates the nurse assigned to a patient record.

2.1.6.1 Procedure Code

```
USE HospitalDB;

GO
CREATE OR ALTER PROCEDURE change_nurse
    @RecordID INT,
    @NurseID INT
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        UPDATE ASSIGN_NURSE
        SET NurseID = @NurseID
        WHERE RecordID = @RecordID;

        IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR('Record not found or nurse does not exist.', 16, 1);
        END
        ELSE
            PRINT 'Nurse assigned to patient record updated successfully.';
    END TRY
```

```
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR('An error occurred: %s', 16, 1, @ErrorMessage);
    END CATCH
END;
GO
```

Listing 2.5: `change_nurse` Procedure

2.1.6.2 Explanation

The `change_nurse` procedure updates the `NurseID` in the `ASSIGN_NURSE` table for a given `RecordID`. Key points:

- Checks if the record exists and raises a custom error if not.
- On success, prints a confirmation message.
- Uses error handling to manage unexpected issues.

2.1.7 Procedure: `update_ex`

This procedure updates an employee's years of experience.

2.1.7.1 Procedure Code

```
USE HospitalDB;

GO
CREATE OR ALTER PROCEDURE update_ex
    @ID INT,
    @year INT
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        UPDATE EMPLOYEE
        SET Experience = @year
        WHERE EmployeeID = @ID;

        IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR('Employee not found.', 16, 1);
        END
        ELSE
            PRINT 'Experience updated successfully.';
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR('An error occurred: %s', 16, 1, @ErrorMessage);
    END CATCH
END;
GO
```

Listing 2.6: `update_ex` Procedure

2.1.7.2 Explanation

The `update_ex` procedure updates the `Experience` field in the `EMPLOYEE` table for a specified `EmployeeID`. Key steps:

- Raises a custom error if the employee does not exist.
- Provides a success message upon completion.
- Implements error handling for detailed failure messages.

2.1.8 Procedure: `insert_cover`

This procedure inserts a record into the `COVER` table if the given billing and insurance details are valid. The function need to confirm a Patient has those insurance through HAVE_INSURANCE table before confirming cover for bill.

2.1.8.1 Procedure Code

```
GO
CREATE OR ALTER PROCEDURE insert_cover
    @BillingID INT,
    @InsuranceID INT
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY

        IF EXISTS(
            SELECT HI.InsuranceID
            FROM HAVE_INSURANCE HI
            INNER JOIN BILLING B ON HI.PatientID = B.PatientID
            WHERE B.BillingID = @BillingID
              AND HI.InsuranceID = @InsuranceID
        )
        BEGIN
            INSERT INTO COVER (BillingID, InsuranceID)
            VALUES (@BillingID, @InsuranceID);

            PRINT 'COVER INSERTED SUCCESSFULLY.';
        END
        ELSE
        BEGIN
            RAISERROR('ERROR IN COVER. PLEASE CHECK THE INFO AGAIN.', 16, 1);
        END
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR('An error occurred: %s', 16, 1, @ErrorMessage);
        RETURN;
```

```
    END CATCH
END;
GO
```

Listing 2.7: `insert_cover` Procedure

2.1.8.2 Explanation

The `insert_cover` procedure adds a new record to the `COVER` table if the provided `BillingID` and `InsuranceID` are valid. Key steps:

- Verifies that the insurance is associated with the patient linked to the billing record using a `IF EXISTS` query.
- If valid, inserts the `BillingID` and `InsuranceID` into the `COVER` table and print a success message.
- If invalid, raises a custom error indicating incorrect information.
- Implements error handling using `BEGIN TRY...END TRY` and `BEGIN CATCH` to manage SQL errors and provide descriptive error messages.

2.2 Change the relational schema to BCNF form

In this section, for the relations which are not in BCNF, we will point out and fix it, and for the other relations, we will give proof that they are in BCNF. The relations are sorted alphabetically. To summarize, there is just one relation in our 25 relations that need to be reformed. That relation is BILLING. Other relation remains the same.

2.2.1 Allergy

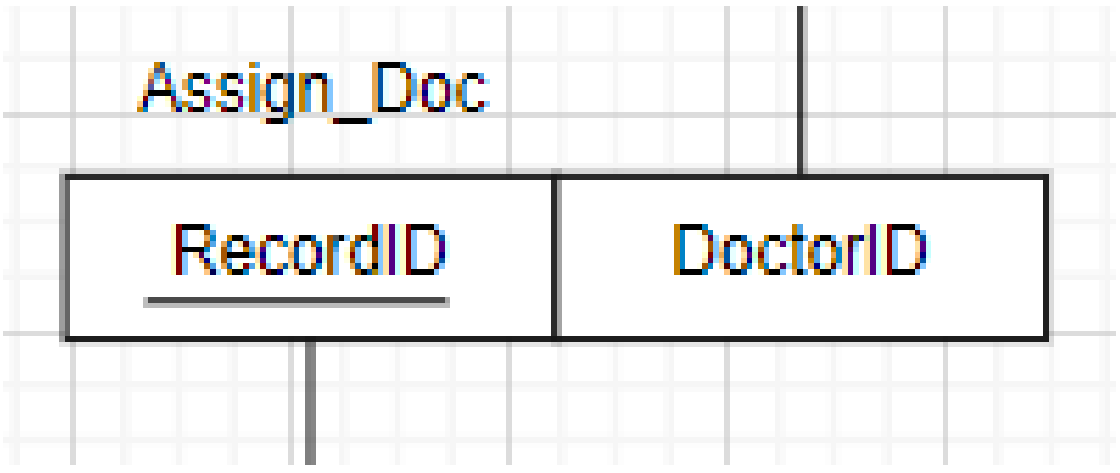


Figure 2.1: relation ALLERGY

This relation is formerly a weak entity in the Entity-Relationship diagram. It has the `RecordID` column as an identifier for a patient record, and a partial key `Allergy` to identify allergy instances in the same patient.

All attributes of this relation contain only atomic values. Each column contains values of a single type, and each row is uniquely identifiable by a primary key. Therefore, this `ALLERGY` relation is in **1NF**.

There is a composite functional dependency here, because the primary key of this relation is the combination { `RecordID`, `Allergy` }. However, there is no partial functional dependency, as `RecordID` or `Allergy` alone does not determine the values of the other attributes. Hence, this relation is in **2NF**.

Furthermore, there are no transitive functional dependencies such that the determinant is a non-key attribute. Thus, this relation is in **3NF**.

Finally, because there is just one functional dependency:

$$\{\texttt{RecordID}, \texttt{Allergy}\} \rightarrow \{\texttt{RecordID}, \texttt{Allergy}\}$$

and the composite key { `RecordID`, `Allergy` } is a candidate key, this relation is in **BCNF**.

2.2.2 ASSIGN_BED

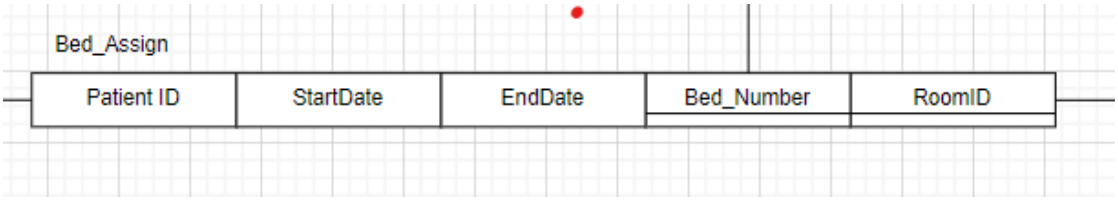


Figure 2.2: relation ASSIGN$_BED$

This relation describes the relationship between the `BED` and `PATIENT_RECORD` relations. It has the `PatientID` as a foreign key to the `PATIENT_RECORD` relation, and the `StartDate` and `EndDate` columns (of type `DATE`) specify the time period during which the patient is assigned to the bed. The `BedNumber` and `RoomID` columns together form a composite primary key for this relation.

All attributes of this relation contain only atomic values, each column contains values of a single type, and each row is uniquely identifiable by a primary key. Therefore, this `ASSIGN_BED` relation is in **1NF**.

The primary key of this relation is a composite key. Clearly, there is a composite functional dependency here:

$$\{\texttt{BedNumber}, \texttt{RoomID}\} \rightarrow \{\texttt{PatientID}, \texttt{StartDate}, \texttt{EndDate}\}$$

However, `BedNumber` alone cannot determine any of the other attributes, as `BedNumber` can be the same for different persons (it is a partial key in its weak entity). Similarly, `RoomID` alone cannot determine any attributes, since there can be many beds and patients in a single room. Therefore, the `PatientID`, `StartDate`, and `EndDate` values are not uniquely determined by `RoomID` alone. Thus, this `ASSIGN_BED` relation is in **2NF**.

The `PatientID` cannot act as the determinant of any functional dependencies in this relation, as it can be `NULL` for some beds, indicating that some beds are not assigned to a patient yet.

Similarly, `StartDate` and `EndDate` cannot be determinants, as more than one patient may be assigned to beds on a given day. Hence, this relation is in **3NF**.

Finally, there is only one functional dependency in this relation:

$$\{\texttt{BedNumber}, \texttt{RoomID}\} \rightarrow \{\texttt{PatientID}, \texttt{StartDate}, \texttt{EndDate}\}$$

and the composite key { `BedNumber`, `RoomID` } is a candidate key. Therefore, this relation is in **BCNF**.

2.2.3 ASSIGN_DOCTOR

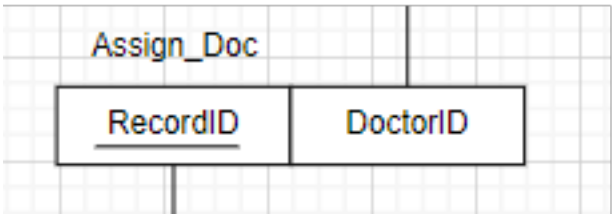


Figure 2.3: relation ASSIGN$_DOC$

The `ASSIGN_DOCTOR` relation describes the assignment of doctors to patient records. It has the following columns:

- `DoctorID` (Foreign Key to a `DOCTOR` relation, which can be `NULL`)
- `RecordID` (Primary Key and Foreign Key to a `PATIENT_RECORD` relation, not `NULL`)

All attributes in this relation (`DoctorID` and `RecordID`) contain only atomic values, and each column holds a single type of data (`INT`). Each row is uniquely identifiable by the primary key, which is `RecordID`. Therefore, the `ASSIGN_DOCTOR` relation is in **1NF**.

The primary key of this relation is `RecordID`, which uniquely identifies each row in the table. Since there is only a single attribute (`RecordID`) in the primary key, there are no partial dependencies possible. Therefore, the `ASSIGN_DOCTOR` relation is in **2NF**.

Since `RecordID` is the primary key, any functional dependency must have `RecordID` as its determinant. There is no transitive dependency here, as all non-key attributes (in this case, `DoctorID`) depend directly on `RecordID`. Therefore, the `ASSIGN_DOCTOR` relation is in **3NF**.

In this relation, we only have one functional dependency:

$$\texttt{RecordID} \rightarrow \texttt{DoctorID}$$

Since `RecordID` is a candidate key (and in fact the primary key) for this relation, it satisfies the requirement that every determinant is a candidate key. (We cannot have the functional dependency `DoctorID` $\rightarrow$ `RecordID`, as a doctor can be assigned to multiple patients.) Therefore, the `ASSIGN_DOCTOR` relation is in **3NF**.

Since there is only one functional dependency, and the determinant (`RecordID`) is a candidate key, we conclude that the `ASSIGN_DOCTOR` relation is in **BCNF**.

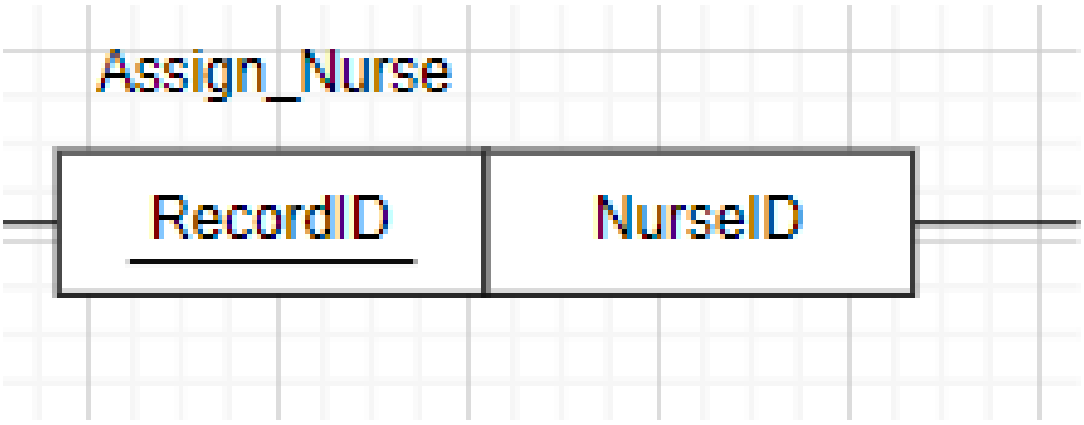


Figure 2.4: relation ASSIGN_NURSE

2.2.4 ASSIGN_NURSE

The `ASSIGN_NURSE` relation describes the assignment of nurses to patient records. It has the following columns:

- `NurseID` (Foreign Key to a `NURSE` relation, which can be `NULL`)
- `RecordID` (Primary Key and Foreign Key to a `PATIENT_RECORD` relation, not `NULL`)

All attributes in this relation (`NurseID` and `RecordID`) contain only atomic values, and each column holds a single type of data (`INT`). Each row is uniquely identifiable by the primary key, which is `RecordID`. Therefore, the `ASSIGN_NURSE` relation is in **1NF**.

The primary key of this relation is `RecordID`, which uniquely identifies each row in the table. Since there is only a single attribute (`RecordID`) in the primary key, there are no partial dependencies possible. Therefore, the `ASSIGN_NURSE` relation is in **2NF**.

Since `RecordID` is the primary key, any functional dependency must have `RecordID` as its determinant. There is no transitive dependency here, as all non-key attributes (in this case, `NurseID`) depend directly on `RecordID`. Therefore, the `ASSIGN_NURSE` relation is in **3NF**.

In this relation, we only have one functional dependency:

$$\texttt{RecordID} \rightarrow \texttt{NurseID}$$

Since `RecordID` is a candidate key (and in fact the primary key) for this relation, it satisfies the requirement that every determinant is a candidate key. (We cannot have the functional dependency $\texttt{NurseID} \rightarrow \texttt{RecordID}$, as a nurse can be assigned to multiple patients.) Therefore, the `ASSIGN_NURSE` relation is in **3NF**.

Since there is only one functional dependency, and the determinant (`RecordID`) is a candidate key, we conclude that the `ASSIGN_NURSE` relation is in **BCNF**.

2.2.5 BED

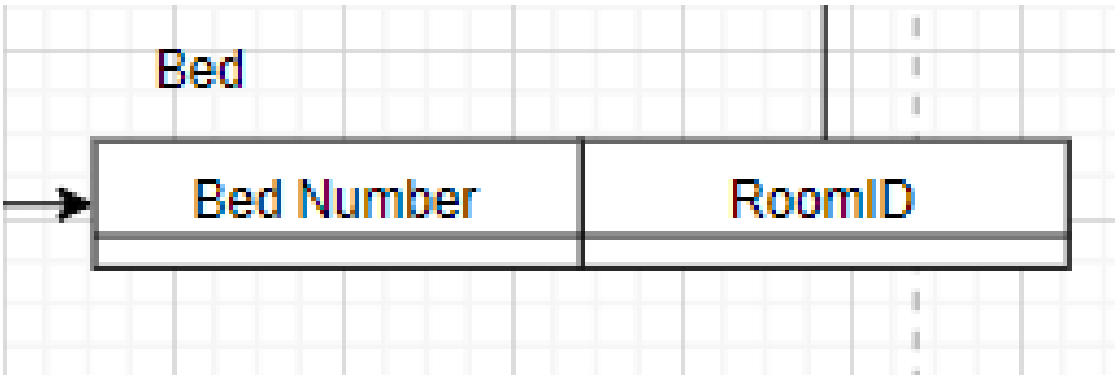


Figure 2.5: relation BED

The `BED` relation describes the beds within a room, where each bed is uniquely identified by a combination of `BedNumber` and `RoomID`. It has the following columns:

- `BedNumber` (part of the primary key for uniquely identifying each bed within a room)
- `RoomID` (part of the composite primary key and a foreign key referencing the `ROOM` relation, indicating which room the bed belongs to)

All attributes in this relation (`BedNumber` and `RoomID`) contain only atomic values, and each column holds a single type of data (`INT`). Each row is uniquely identifiable by the composite primary key { `BedNumber`, `RoomID` }. Therefore, the `BED` relation is in **1NF**.

The primary key of this relation is a composite key { `BedNumber`, `RoomID` }, which uniquely identifies each bed in a specific room. Since there are no non-key attributes in the `BED` relation (all attributes are part of the primary key), there are no partial dependencies. Therefore, the `BED` relation is in **2NF**.

Since there are no non-key attributes, there are no transitive dependencies in the `BED` relation. Therefore, the `BED` relation is in **3NF**.

The only functional dependency in this relation is:

$\{\texttt{BedNumber}, \texttt{RoomID}\} \rightarrow \{\texttt{BedNumber}, \texttt{RoomID}\}$ (trivially, the primary key determines itself)

Since the composite key { `BedNumber`, `RoomID` } is a candidate key for this relation, and there are no other functional dependencies, the condition for **BCNF** is satisfied. Since there is only

one functional dependency, and the determinant { `BedNumber`, `RoomID` } is a candidate key, we conclude that the `BED` relation is in **BCNF**.

2.2.6 BILLING

In this `BILLING` relation, we notice that there is a violation of **BCNF**. Specifically, there is a functional dependency:

$$\{\texttt{InitialAmount}, \texttt{CoverAmount}\} \rightarrow \texttt{FinalAmount}$$

where `FinalAmount` is calculated as `FinalAmount = InitialAmount - CoverAmount`. Here, `InitialAmount` and `CoverAmount` are not candidate keys of this relation. As a result, we decided to decompose the relation.

We create a new relation that holds the amounts, called `BILLING_CALCULATION`, with the following attributes:

- `InitialAmount DECIMAL(10, 2) NOT NULL`
- `CoverAmount DECIMAL(10, 2) NOT NULL`
- `FinalAmount DECIMAL(10, 2) NOT NULL`

The primary key of this new relation is the composite key { `InitialAmount`, `CoverAmount` }. Additionally, we remove the column `FinalAmount` from the `BILLING` relation to eliminate the BCNF violation. This is the relational schema after fixing:

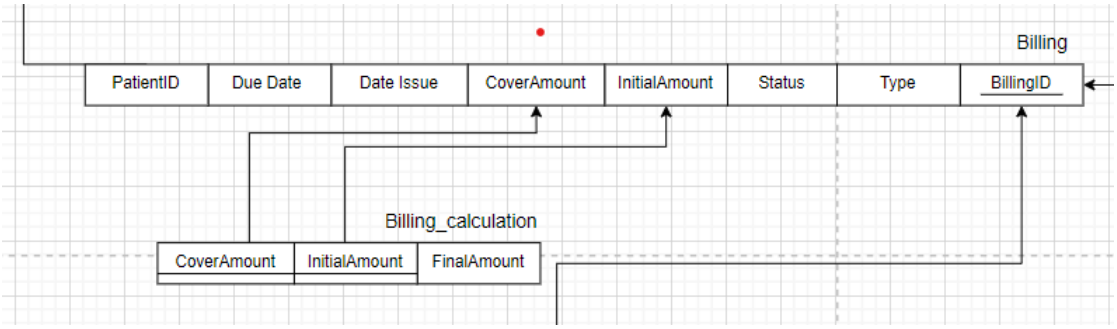


Figure 2.6: relation BILLING

2.2.7 COVER

The `COVER` relation describes the many-to-many relationship between `BILLING` and `INSURANCE`. Each `BILLING` record can be associated with multiple `INSURANCE` records, and each `INSURANCE` record can cover multiple `BILLING` records. It has the following columns:

- `BillingID` (Primary Key and Foreign Key referencing the `BILLING` relation, not `NULL`)
- `InsuranceID` (Primary Key and Foreign Key referencing the `INSURANCE` relation, not `NULL`)

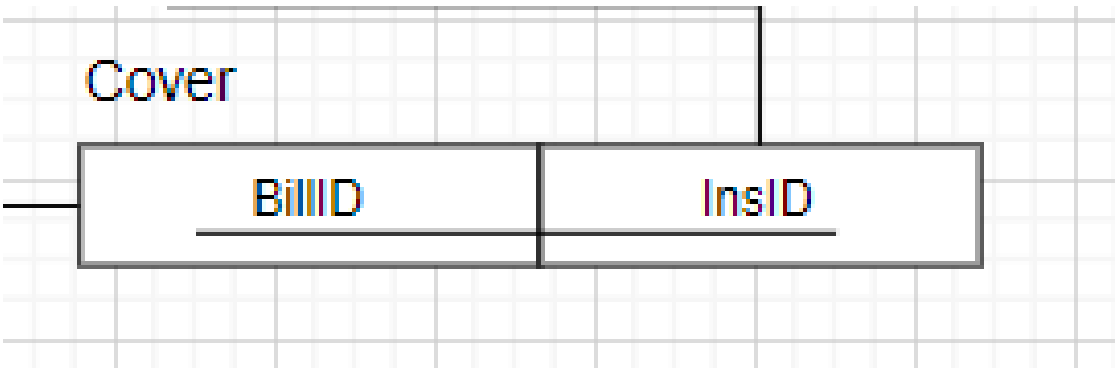


Figure 2.7: relation COVER

All attributes in this relation (`BillingID` and `InsuranceID`) contain only atomic values, and each column holds a single type of data (`INT`). Each row is uniquely identifiable by the composite primary key { `BillingID`, `InsuranceID` }. Therefore, the `COVER` relation is in **1NF**.

The primary key of this relation is a composite key { `BillingID`, `InsuranceID` }, which uniquely identifies each coverage record in the relationship. Since there are no non-key attributes in the `COVER` relation (all attributes are part of the primary key), there are no partial dependencies. Therefore, the `COVER` relation is in **2NF**.

Since there are no non-key attributes, there are no transitive dependencies in the `COVER` relation. Therefore, the `COVER` relation is in **3NF**.

The only functional dependency in this relation is:

$\{\texttt{BillingID}, \texttt{InsuranceID}\} \rightarrow \{\texttt{BillingID}, \texttt{InsuranceID}\}$ (trivially, the primary key determines itself)

Since the composite key { `BillingID`, `InsuranceID` } is a candidate key for this relation, and there are no other functional dependencies, the condition for **BCNF** is satisfied. Since there is only one functional dependency, and the determinant { `BillingID`, `InsuranceID` } is a candidate key, we conclude that the `COVER` relation is in **BCNF**.

2.2.8 DEPARTMENT

The `DEPARTMENT` relation stores information about departments. It has the following columns:

- `DepartmentNumber` (Primary Key, `int`, `NOT NULL`)
- `ManageID` (Foreign Key referencing another relation, likely `EMPLOYEE`, `int`, can be `NULL`)
- `Name` (`varchar(100)`, `NOT NULL`)

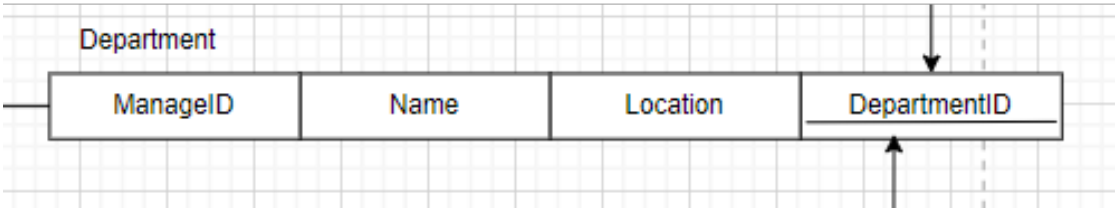


Figure 2.8: relation DEPARTMENT

- `Location` (`varchar(100)`, can be `NULL`)

All attributes in this relation (`DepartmentNumber`, `ManageID`, `Name`, and `Location`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `DepartmentNumber`. Therefore, the `DEPARTMENT` relation is in **1NF**.

The primary key of this relation is a single attribute, `DepartmentNumber`, which uniquely identifies each department. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `DEPARTMENT` relation is in **2NF**.

`DepartmentNumber` is the primary key, and all other attributes (`ManageID`, `Name`, and `Location`) depend directly on `DepartmentNumber`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For instance, `Name` and `Location` are directly related to `DepartmentNumber`, not to each other or to `ManageID`. Therefore, the `DEPARTMENT` relation is in **3NF**.

There are two functional dependencies (FDs) here. The first one is the trivial FD:

$$\texttt{DepartmentNumber} \rightarrow \{\texttt{ManageID}, \texttt{Name}, \texttt{Location}\}$$

where `DepartmentNumber` is clearly the candidate key. The second FD is:

$$\texttt{Name} \rightarrow \{\texttt{DepartmentNumber}, \texttt{ManageID}, \texttt{Location}\}$$

As different departments have different names, `Name` is also a candidate key in this relation (since there are no two departments with the same name). This does not violate the rule of **BCNF**. Hence, this relation is in **BCNF** form.

2.2.9 DIAGNOSTIC_TEST

The `DIAGNOSTIC_TEST` relation stores information about diagnostic tests. It has the following columns:

- `TestID` (Primary Key, `int`, `NOT NULL`)
- `TestName` (`nvarchar(100)`, `NOT NULL`)
- `TestDescription` (`nvarchar(255)`, can be `NULL`)

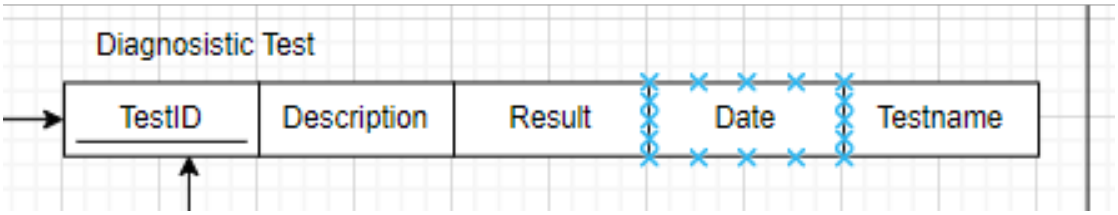


Figure 2.9: relation DIAGNOSTIC_TEST

- `TestDate` (`date`, `NOT NULL`)
- `TestResult` (`nvarchar(100)`, can be `NULL`)

All attributes in this relation (`TestID`, `TestName`, `TestDescription`, `TestDate`, and `TestResult`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `TestID`. Therefore, the `DIAGNOSTIC_TEST` relation is in **1NF**.

The primary key of this relation is a single attribute, `TestID`, which uniquely identifies each diagnostic test. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `DIAGNOSTIC_TEST` relation is in **2NF**.

`TestID` is the primary key, and all other attributes (`TestName`, `TestDescription`, `TestDate`, and `TestResult`) depend directly on `TestID`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `TestName`, `TestDescription`, `TestDate`, and `TestResult` are directly related to `TestID` and not to each other. Therefore, the `DIAGNOSTIC_TEST` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\texttt{TestID} \rightarrow \{\texttt{TestName}, \texttt{TestDescription}, \texttt{TestDate}, \texttt{TestResult}\}$$

Since `TestID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `TestID` as the determinant, and `TestID` is the candidate key, we conclude that the `DIAGNOSTIC_TEST` relation is in **BCNF**.

2.2.10 EMPLOYEE

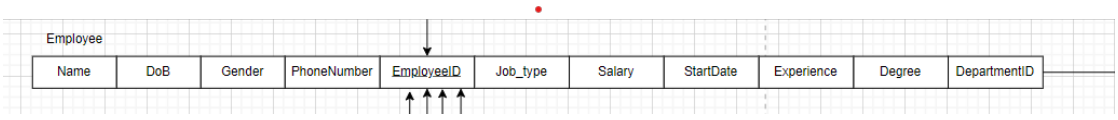


Figure 2.10: relation EMPLOYEE

The `EMPLOYEE` relation stores information about employees. It has the following columns:

- `EmployeeID` (Primary Key, `int`, `NOT NULL`)

- `Name` (`varchar(100)`, `NOT NULL`)
- `Dob` (Date of Birth, `date`, `NOT NULL`)
- `Gender` (`char(1)`, can be `NULL`)
- `PhoneNumber` (`varchar(15)`, `NOT NULL`)
- `Job_type` (`varchar(50)`, `NOT NULL`)
- `Salary` (`decimal(10,2)`, `NOT NULL`)
- `startDate` (`date`, `NOT NULL`)
- `Experience` (`int`, `NOT NULL`)
- `Degree` (`varchar(100)`, can be `NULL`)
- `DepartmentID` (Foreign Key referencing `DEPARTMENT` relation, `int`, `NOT NULL`)

All attributes in this relation (`EmployeeID`, `Name`, `Dob`, `Gender`, `PhoneNumber`, `Job_type`, `Salary`, `startDate`, `Experience`, `Degree`, and `DepartmentID`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `EmployeeID`. Therefore, the `EMPLOYEE` relation is in **1NF**.

The primary key of this relation is a single attribute, `EmployeeID`, which uniquely identifies each employee. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `EMPLOYEE` relation is in **2NF**.

There is another functional dependency (FD) other than the trivial FD in this relation:

$$\texttt{PhoneNumber} \rightarrow \{\text{all of the remaining attributes in the relation}\}$$

Since the phone number is unique for each employee, it is also a candidate key. This does not violate the rules of **3NF**. Hence, this relation is in **3NF**.

Since there are two functional dependencies in this relation:

$$\texttt{EmployeeID} \rightarrow \{\text{all of the remaining attributes in the relation}\}$$

$$\texttt{PhoneNumber} \rightarrow \{\text{all of the remaining attributes in the relation}\}$$

and both `EmployeeID` and `PhoneNumber` are candidate keys, this does not violate the rules of **BCNF**. Hence, the `EMPLOYEE` relation is in **BCNF**.

2.2.11 EQUIPMENT

The `EQUIPMENT` relation stores information about equipment. It has the following columns:

- `EquipmentID` (Primary Key, `int`, `NOT NULL`)
- `Type` (`nvarchar(100)`, `NOT NULL`)
- `Status` (`nvarchar(50)`, `NOT NULL`)
- `Name` (`nvarchar(100)`, `NOT NULL`)

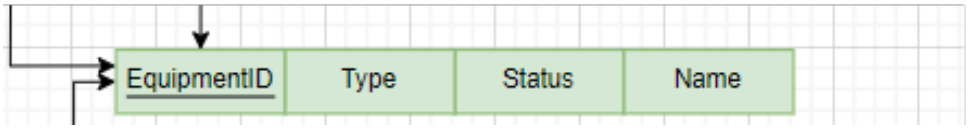


Figure 2.11: relation EQUIPMENT

All attributes in this relation (`EquipmentID`, `Type`, `Status`, and `Name`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `EquipmentID`. Therefore, the `EQUIPMENT` relation is in **1NF**.

The primary key of this relation is a single attribute, `EquipmentID`, which uniquely identifies each piece of equipment. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `EQUIPMENT` relation is in **2NF**.

`EquipmentID` is the primary key, and all other attributes (`Type`, `Status`, and `Name`) depend directly on `EquipmentID`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `Type`, `Status`, and `Name` are directly related to `EquipmentID` and not to each other. Therefore, the `EQUIPMENT` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\texttt{EquipmentID} \rightarrow \{\texttt{Type}, \texttt{Status}, \texttt{Name}\}$$

Since `EquipmentID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `EquipmentID` as the determinant, and `EquipmentID` is the candidate key, we conclude that the `EQUIPMENT` relation is in **BCNF**.

2.2.12 HAVE_INSURANCE

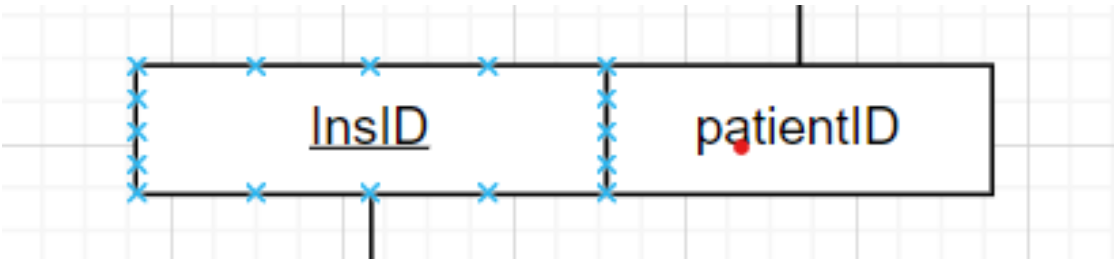


Figure 2.12: relation HAVE_INSURANCE

The `HAVE_INSURANCE` relation describes the relationship between `INSURANCE` and `PATIENT_RECORD`. It has the following columns:

- `InsuranceID` (Primary Key and Foreign Key referencing the `INSURANCE` relation, `int`, `NOT NULL`)

- `PatientID` (Foreign Key referencing the `PATIENT_RECORD` relation, `int`, can be `NULL`)

All attributes in this relation (`InsuranceID` and `PatientID`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `InsuranceID`. Therefore, the `HAVE_INSURANCE` relation is in **1NF**.

The primary key of this relation is a single attribute, `InsuranceID`, which uniquely identifies each entry in the table. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `HAVE_INSURANCE` relation is in **2NF**.

`InsuranceID` is the primary key, and the only other attribute (`PatientID`) depends directly on `InsuranceID`. There are no transitive dependencies in this relation, as `PatientID` depends only on `InsuranceID`. Therefore, the `HAVE_INSURANCE` relation is in **3NF**.

The only functional dependency in this relation is:

$$\texttt{InsuranceID} \rightarrow \texttt{PatientID}$$

Since `InsuranceID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `InsuranceID` as the determinant, and `InsuranceID` is the candidate key, we conclude that the `HAVE_INSURANCE` relation is in **BCNF**.

2.2.13 INSURANCE

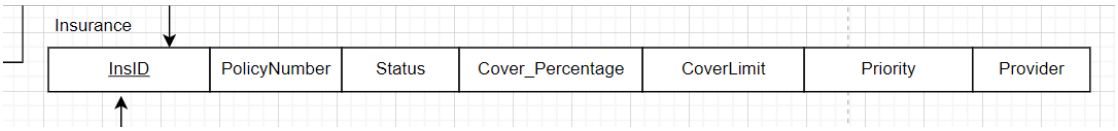


Figure 2.13: relation INSURANCE

The `INSURANCE` relation stores information about insurance policies. It has the following columns:

- `InsuranceID` (Primary Key, `int`, `NOT NULL`)
- `Provider` (`varchar(255)`, can be `NULL`)
- `PolicyNumber` (`varchar(255)`, can be `NULL`)
- `StatusInsurance` (`varchar(50)`, can be `NULL`)
- `CoverPercentage` (`decimal(5,2)`, can be `NULL`)
- `CoverLimit` (`decimal(18,2)`, can be `NULL`)
- `InsurancePriority` (`int`, can be `NULL`)

All attributes in this relation (`InsuranceID`, `Provider`, `PolicyNumber`, `StatusInsurance`, `CoverPercentage`, `CoverLimit`, and `InsurancePriority`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `InsuranceID`. Therefore, the `INSURANCE` relation is in **1NF**.

The primary key of this relation is a single attribute, `InsuranceID`, which uniquely identifies each insurance policy. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `INSURANCE` relation is in **2NF**.

`InsuranceID` is the primary key, and all other attributes (`Provider`, `PolicyNumber`, `StatusInsurance`, `CoverPercentage`, `CoverLimit`, and `InsurancePriority`) depend directly on `InsuranceID`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `Provider`, `PolicyNumber`, `StatusInsurance`, and `CoverPercentage` are directly related to `InsuranceID` and not to each other. Therefore, the `INSURANCE` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\texttt{InsuranceID} \rightarrow \{\texttt{Provider}, \texttt{PolicyNumber}, \texttt{StatusInsurance}, \texttt{CoverPercentage}, \texttt{CoverLimit}, \texttt{InsurancePriority}\}$$

Since `InsuranceID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `InsuranceID` as the determinant, and `InsuranceID` is the candidate key, we conclude that the `INSURANCE` relation is in **BCNF**.

2.2.14 MAINTAIN

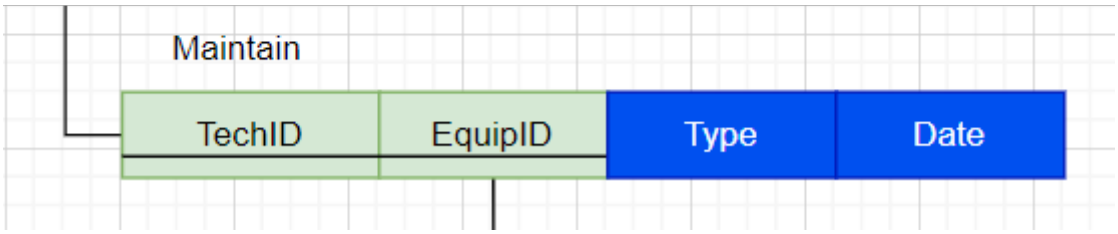


Figure 2.14: relation MAINTAIN

The `MAINTAIN` relation describes the maintenance records between `TECHNICIAN` and `EQUIPMENT`. It has the following columns:

- `TechID` (Primary Key, Foreign Key referencing the `TECHNICIAN` relation, `int`, `NOT NULL`)
- `EquipID` (Primary Key, Foreign Key referencing the `EQUIPMENT` relation, `int`, `NOT NULL`)

- `TypeMaintain` (`nvarchar(100)`, `NOT NULL`)
- `DateMaintain` (`date`, `NOT NULL`)

All attributes in this relation (`TechID`, `EquipID`, `TypeMaintain`, and `DateMaintain`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the composite primary key { `TechID`, `EquipID` }. Therefore, the `MAINTAIN` relation is in **1NF**.

The primary key of this relation is a composite key { `TechID`, `EquipID` }, which uniquely identifies each maintenance record. Since there are no partial dependencies (dependencies on only part of a composite key), all non-key attributes (`TypeMaintain` and `DateMaintain`) depend on the entire primary key { `TechID`, `EquipID` }. Therefore, the `MAINTAIN` relation is in **2NF**.

{ `TechID`, `EquipID` } is the primary key, and all other attributes (`TypeMaintain` and `DateMaintain`) depend directly on { `TechID`, `EquipID` }. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `TypeMaintain` and `DateMaintain` are directly related to { `TechID`, `EquipID` } and not to each other. Therefore, the `MAINTAIN` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\{\texttt{TechID}, \texttt{EquipID}\} \rightarrow \{\texttt{TypeMaintain}, \texttt{DateMaintain}\}$$

Since { `TechID`, `EquipID` } is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have { `TechID`, `EquipID` } as the determinant, and { `TechID`, `EquipID` } is the candidate key, we conclude that the `MAINTAIN` relation is in **BCNF**.

2.2.15 MEDICAL_HISTORY

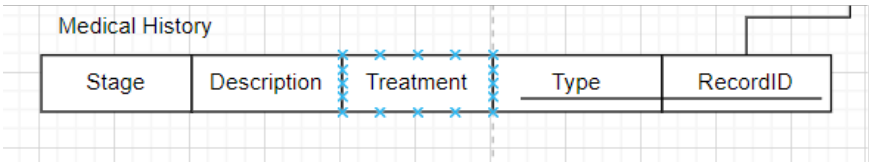


Figure 2.15: relation MEDICAL_HISTORY

The `MEDICAL_HISTORY` relation describes medical history entries related to a specific patient record. It has the following columns:

- `RecordID` (Primary Key, Foreign Key referencing the `PATIENT_RECORD` relation, `int`, `NOT NULL`)
- `TypeName` (Primary Key, `varchar(255)`, `NOT NULL`)
- `Treatment` (`varchar(255)`, can be `NULL`)

- `DescriptionDetail` (`text`, can be `NULL`)
- `Stage` (`varchar(50)`, can be `NULL`)

All attributes in this relation (`RecordID`, `TypeName`, `Treatment`, `DescriptionDetail`, and `Stage`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the composite primary key { `RecordID`, `TypeName` }. Therefore, the `MEDICAL_HISTORY` relation is in **1NF**.

The primary key of this relation is a composite key { `RecordID`, `TypeName` }, which uniquely identifies each medical history entry for a patient. Since there are no partial dependencies (dependencies on only part of a composite key), all non-key attributes (`Treatment`, `DescriptionDetail`, and `Stage`) depend on the entire primary key { `RecordID`, `TypeName` }. Therefore, the `MEDICAL_HISTORY` relation is in **2NF**.

{ `RecordID`, `TypeName` } is the primary key, and all other attributes (`Treatment`, `DescriptionDetail`, and `Stage`) depend directly on { `RecordID`, `TypeName` }. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `Treatment`, `DescriptionDetail`, and `Stage` are directly related to { `RecordID`, `TypeName` } and not to each other. Therefore, the `MEDICAL_HISTORY` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\{\texttt{RecordID}, \texttt{TypeName}\} \rightarrow \{\texttt{Treatment}, \texttt{DescriptionDetail}, \texttt{Stage}\}$$

Since { `RecordID`, `TypeName` } is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have { `RecordID`, `TypeName` } as the determinant, and { `RecordID`, `TypeName` } is the candidate key, we conclude that the `MEDICAL_HISTORY` relation is in **BCNF**.

2.2.16 NURSE

The `NURSE` relation is derived from the `EMPLOYEE` relation and stores information specific to employees who are nurses. It has the following columns:

- `NurseID` (Primary Key, Foreign Key referencing the `EMPLOYEE` relation, `int`, `NOT NULL`)
- `Specialty` (`nvarchar(100)`, can be `NULL`)

All attributes in this relation (`NurseID` and `Specialty`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `NurseID`. Therefore, the `NURSE` relation is in **1NF**.

The primary key of this relation is a single attribute, `NurseID`, which uniquely identifies each nurse. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `NURSE` relation is in **2NF**.

`NurseID` is the primary key, and the only other attribute, `Specialty`, depends directly on `NurseID`. There are no transitive dependencies in this relation, as `Specialty` is directly related to `NurseID` and not to any other non-key attribute. Therefore, the `NURSE` relation is in **3NF**.

The only functional dependency in this relation is:

$$\texttt{NurseID} \rightarrow \texttt{Specialty}$$

Since `NurseID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `NurseID` as the determinant, and `NurseID` is the candidate key, we conclude that the `NURSE` relation is in **BCNF**.

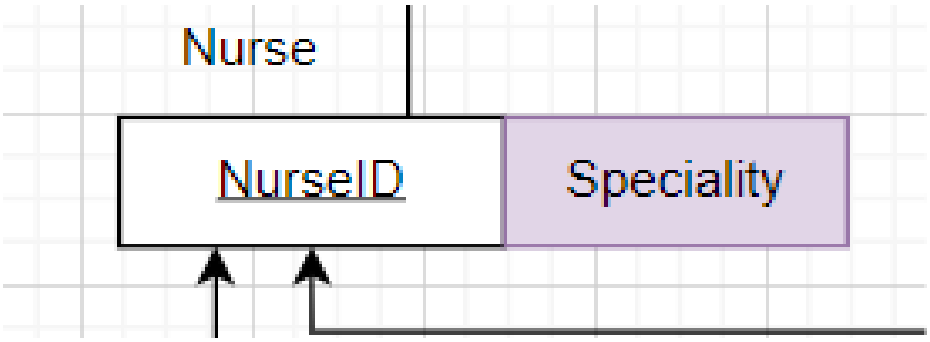


Figure 2.16: relation NURSE

2.2.17 OTHER_EMPLOYEE

The `OTHER_EMPLOYEE` relation is derived from the `EMPLOYEE` relation and stores information specific to employees in categories other than nurses or other specialized roles. It has the following columns:

- `EmployeeID` (Primary Key, Foreign Key referencing the `EMPLOYEE` relation, `int`, `NOT NULL`)
- `Specialty` (`varchar(100)`, can be `NULL`)

All attributes in this relation (`EmployeeID` and `Specialty`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `EmployeeID`. Therefore, the `OTHER_EMPLOYEE` relation is in **1NF**.

The primary key of this relation is a single attribute, `EmployeeID`, which uniquely identifies each "other" employee. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `OTHER_EMPLOYEE` relation is in **2NF**.

`EmployeeID` is the primary key, and the only other attribute, `Specialty`, depends directly on `EmployeeID`. There are no transitive dependencies in this relation, as `Specialty` is directly related to `EmployeeID` and not to any other non-key attribute. Therefore, the `OTHER_EMPLOYEE` relation is in **3NF**.

The only functional dependency in this relation is:

$$\texttt{EmployeeID} \rightarrow \texttt{Specialty}$$

Since `EmployeeID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `EmployeeID` as the determinant, and `EmployeeID` is the candidate key, we conclude that the `OTHER_EMPLOYEE` relation is in **BCNF**.

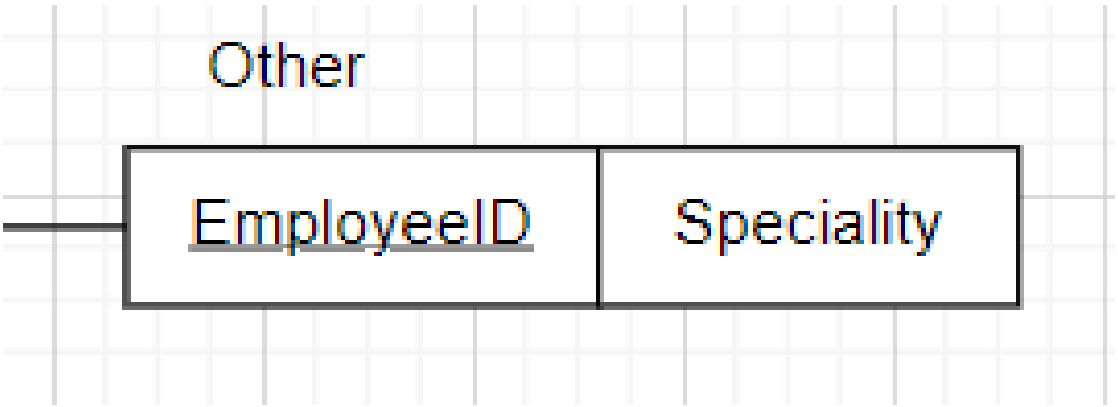


Figure 2.17: relation OTHER_EMPLOYEE

2.2.18 PATIENT_RECORD

The `PATIENT_RECORD` relation stores information about patients. It has the following columns:

- `RecordID` (Primary Key, `int`, `NOT NULL`)
- `FirstName` (`nvarchar(50)`, `NOT NULL`)
- `LastName` (`nvarchar(50)`, `NOT NULL`)
- `Gender` (`nvarchar(10)`, `NOT NULL`)
- `ContactInfo` (`nvarchar(255)`, `NOT NULL`)
- `EmerContactInfo` (Emergency Contact Info, `nvarchar(255)`, can be `NULL`)
- `Address` (`nvarchar(255)`, can be `NULL`)
- `CurrentMedication` (`nvarchar(255)`, can be `NULL`)

All attributes in this relation (`RecordID`, `FirstName`, `LastName`, `Gender`, `ContactInfo`, `EmerContactInfo`, `Address`, and `CurrentMedication`) contain only atomic values, and each column holds a single

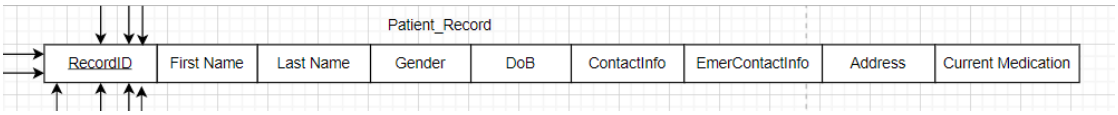


Figure 2.18: relation PATIENT_RECORD

type of data. Each row is uniquely identifiable by the primary key, which is `RecordID`. Therefore, the `PATIENT_RECORD` relation is in **1NF**.

The primary key of this relation is a single attribute, `RecordID`, which uniquely identifies each patient record. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `PATIENT_RECORD` relation is in **2NF**.

`RecordID` is the primary key, and all other attributes (`FirstName`, `LastName`, `Gender`, `ContactInfo`, `EmerContactInfo`, `Address`, and `CurrentMedication`) depend directly on `RecordID`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `FirstName`, `LastName`, `ContactInfo`, and `CurrentMedication` are directly related to `RecordID` and not to each other. Therefore, the `PATIENT_RECORD` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\texttt{RecordID} \rightarrow \{\texttt{FirstName}, \texttt{LastName}, \texttt{Gender}, \texttt{ContactInfo}, \texttt{EmerContactInfo}, \texttt{Address}, \texttt{CurrentMedication}\}$$

Since `RecordID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `RecordID` as the determinant, and `RecordID` is the candidate key, we conclude that the `PATIENT_RECORD` relation is in **BCNF**.

2.2.19 PAYMENT

The `PAYMENT` relation stores information about payments made. It has the following columns:

- `ReceiptID` (Primary Key, `int`, `NOT NULL`)
- `Amount` (`decimal(18,2)`, can be `NULL`)
- `Note` (`varchar(255)`, can be `NULL`)
- `Method` (Payment method, `varchar(50)`, can be `NULL`)
- `DatePay` (Date of payment, `date`, can be `NULL`)
- `BillingID` (Foreign Key referencing `BILLING` relation, `int`, can be `NULL`)

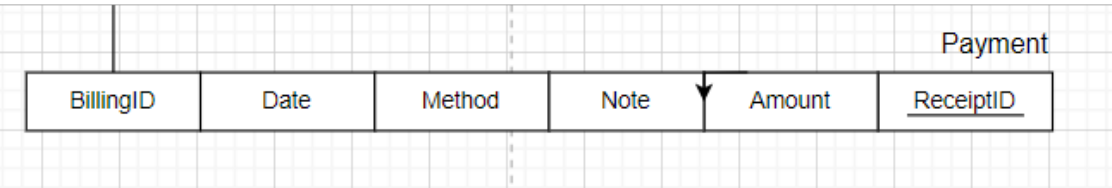


Figure 2.19: relation PAYMENT

All attributes in this relation (`ReceiptID`, `Amount`, `Note`, `Method`, `DatePay`, and `BillingID`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `ReceiptID`. Therefore, the `PAYMENT` relation is in **1NF**.

The primary key of this relation is a single attribute, `ReceiptID`, which uniquely identifies each payment record. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `PAYMENT` relation is in **2NF**.

`ReceiptID` is the primary key, and all other attributes (`Amount`, `Note`, `Method`, `DatePay`, and `BillingID`) depend directly on `ReceiptID`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `Amount`, `Method`, `DatePay`, and `BillingID` are directly related to `ReceiptID` and not to each other. Therefore, the `PAYMENT` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\texttt{ReceiptID} \rightarrow \{\texttt{Amount}, \texttt{Note}, \texttt{Method}, \texttt{DatePay}, \texttt{BillingID}\}$$

Since `ReceiptID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `ReceiptID` as the determinant, and `ReceiptID` is the candidate key, we conclude that the `PAYMENT` relation is in **BCNF**.

2.2.20 PERFORM_SURGERY

The `PERFORM_SURGERY` relation describes the relationship involving a `DOCTOR`, `PATIENT_RECORD`, and `SURGERY`. It has the following columns:

- `SurgeryID` (Primary Key, Foreign Key referencing the `SURGERY` relation, `int`, `NOT NULL`)
- `DoctorID` (Primary Key, Foreign Key referencing the `DOCTOR` relation, `int`, `NOT NULL`)
- `RecordID` (Foreign Key referencing the `PATIENT_RECORD` relation, `int`, can be `NULL`)

All attributes in this relation (`SurgeryID`, `DoctorID`, and `RecordID`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the

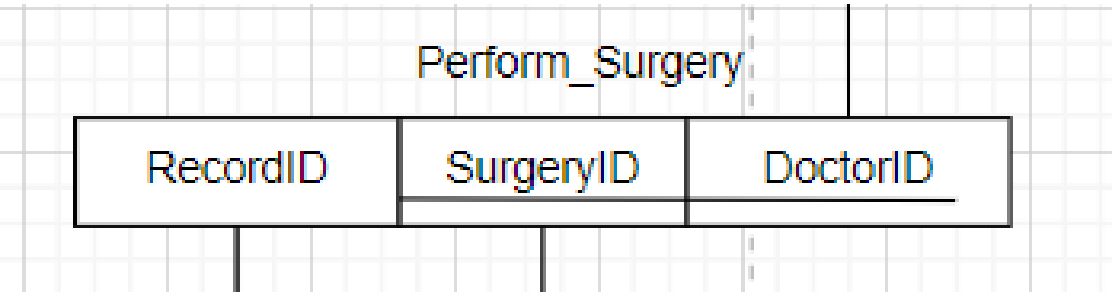


Figure 2.20: relation PERFORM_SURGERY

composite primary key { `SurgeryID`, `DoctorID` }. Therefore, the `PERFORM_SURGERY` relation is in **1NF**.

The primary key of this relation is a composite key { `SurgeryID`, `DoctorID` }, which uniquely identifies each entry in the table. Since there are no partial dependencies (dependencies on only part of a composite key), all non-key attributes (`RecordID`) depend on the entire primary key { `SurgeryID`, `DoctorID` }. Therefore, the `PERFORM_SURGERY` relation is in **2NF**.

{ `SurgeryID`, `DoctorID` } is the primary key, and the only non-key attribute, `RecordID`, depends directly on { `SurgeryID`, `DoctorID` }. There are no transitive dependencies in this relation, as `RecordID` is directly related to { `SurgeryID`, `DoctorID` } and not to any other non-key attribute. Therefore, the `PERFORM_SURGERY` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\{\texttt{SurgeryID}, \texttt{DoctorID}\} \rightarrow \texttt{RecordID}$$

Since { `SurgeryID`, `DoctorID` } is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have { `SurgeryID`, `DoctorID` } as the determinant, and { `SurgeryID`, `DoctorID` } is the candidate key, we conclude that the `PERFORM_SURGERY` relation is in **BCNF**.

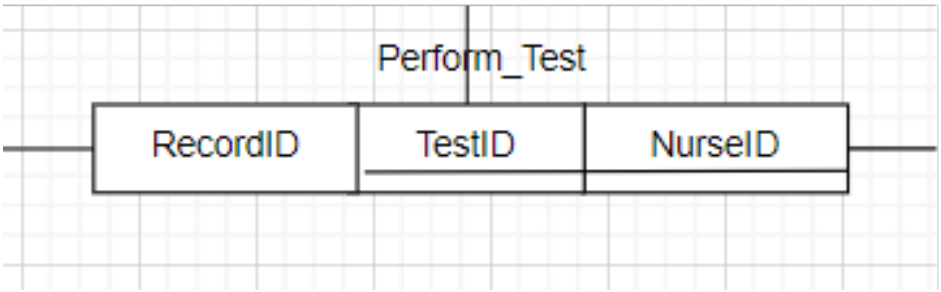


Figure 2.21: relation PERFORM_TEST

2.2.21 PERFORM_TEST

The `PERFORM_TEST` relation describes the relationship involving a `NURSE`, `PATIENT_RECORD`, and `TEST`. It has the following columns:

- `TestID` (Primary Key, Foreign Key referencing the `TEST` relation, `int`, `NOT NULL`)
- `NurseID` (Primary Key, Foreign Key referencing the `NURSE` relation, `int`, `NOT NULL`)
- `RecordID` (Foreign Key referencing the `PATIENT_RECORD` relation, `int`, can be `NULL`)

All attributes in this relation (`TestID`, `NurseID`, and `RecordID`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the composite primary key { `TestID`, `NurseID` }. Therefore, the `PERFORM_TEST` relation is in **1NF**.

The primary key of this relation is a composite key { `TestID`, `NurseID` }, which uniquely identifies each entry in the table. Since there are no partial dependencies (dependencies on only part of a composite key), all non-key attributes (`RecordID`) depend on the entire primary key { `TestID`, `NurseID` }. Therefore, the `PERFORM_TEST` relation is in **2NF**.

{ `TestID`, `NurseID` } is the primary key, and the only non-key attribute, `RecordID`, depends directly on { `TestID`, `NurseID` }. There are no transitive dependencies in this relation, as `RecordID` is directly related to { `TestID`, `NurseID` } and not to any other non-key attribute. Therefore, the `PERFORM_TEST` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\{\texttt{TestID}, \texttt{NurseID}\} \rightarrow \texttt{RecordID}$$

Since { `TestID`, `NurseID` } is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have { `TestID`, `NurseID` } as the determinant, and { `TestID`, `NurseID` } is the candidate key, we conclude that the `PERFORM_TEST` relation is in **BCNF**.

2.2.22 ROOM

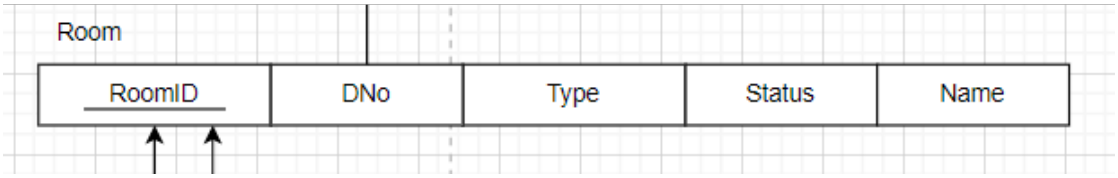


Figure 2.22: relation ROOM

The `ROOM` relation stores information about rooms. It has the following columns:

- `RoomID` (Primary Key, `int`, `NOT NULL`)
- `Dno` (Foreign Key referencing `DEPARTMENT` relation, `int`, can be `NULL`)
- `TypeRoom` (`nvarchar(100)`, can be `NULL`)
- `StatusRoom` (`nvarchar(100)`, can be `NULL`)

All attributes in this relation (`RoomID`, `Dno`, `TypeRoom`, and `StatusRoom`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `RoomID`. Therefore, the `ROOM` relation is in **1NF**.

The primary key of this relation is a single attribute, `RoomID`, which uniquely identifies each room. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `ROOM` relation is in **2NF**.

`RoomID` is the primary key, and all other attributes (`Dno`, `TypeRoom`, and `StatusRoom`) depend directly on `RoomID`. There are no transitive dependencies in this relation, as no non-key attribute depends on another non-key attribute. For example, `Dno`, `TypeRoom`, and `StatusRoom` are directly related to `RoomID` and not to each other. Therefore, the `ROOM` relation is in **3NF**.

The only functional dependencies in this relation are:

$$\texttt{RoomID} \rightarrow \{\texttt{Dno}, \texttt{TypeRoom}, \texttt{StatusRoom}\}$$

Since `RoomID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `RoomID` as the determinant, and `RoomID` is the candidate key, we conclude that the `ROOM` relation is in **BCNF**.

2.2.23 TECHNICIAN

The `TECHNICIAN` relation is derived from the `EMPLOYEE` relation and stores information specific to employees who are technicians. It has the following columns:

- `EmployeeID` (Primary Key, Foreign Key referencing the `EMPLOYEE` relation, `int`, `NOT NULL`)

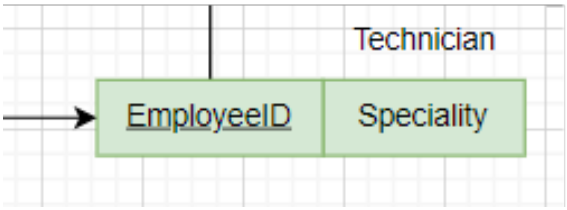


Figure 2.23: relation TECHNICIAN

- `Specialty` (`nvarchar(100)`, can be `NULL`)

All attributes in this relation (`EmployeeID` and `Specialty`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the primary key, which is `EmployeeID`. Therefore, the `TECHNICIAN` relation is in **1NF**.

The primary key of this relation is a single attribute, `EmployeeID`, which uniquely identifies each technician. Since there is only one attribute in the primary key, there can be no partial dependencies (dependencies on only part of a composite key). Therefore, the `TECHNICIAN` relation is in **2NF**.

`EmployeeID` is the primary key, and the only other attribute, `Specialty`, depends directly on `EmployeeID`. There are no transitive dependencies in this relation, as `Specialty` is directly related to `EmployeeID` and not to any other non-key attribute. Therefore, the `TECHNICIAN` relation is in **3NF**.

The only functional dependency in this relation is:

$$\texttt{EmployeeID} \rightarrow \texttt{Specialty}$$

Since `EmployeeID` is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have `EmployeeID` as the determinant, and `EmployeeID` is the candidate key, we conclude that the `TECHNICIAN` relation is in **BCNF**.

2.2.24 USED_IN_SURGERY

The `USED_IN_SURGERY` relation describes the many-to-many relationship between `SURGERY` and `EQUIPMENT`. It has the following columns:

- `SurgeryID` (Primary Key, Foreign Key referencing the `SURGERY` relation, `int`, `NOT NULL`)
- `EquipmentID` (Primary Key, Foreign Key referencing the `EQUIPMENT` relation, `int`, `NOT NULL`)

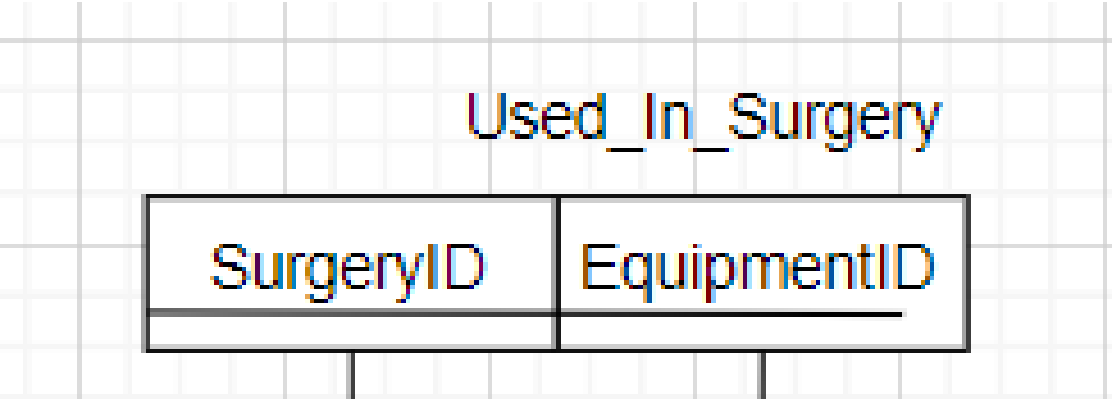


Figure 2.24: relation USED_IN_SURGERY

All attributes in this relation (`SurgeryID` and `EquipmentID`) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the composite primary key { `SurgeryID`, `EquipmentID` }. Therefore, the `USED_IN_SURGERY` relation is in **1NF**.

The primary key of this relation is a composite key { `SurgeryID`, `EquipmentID` }, which uniquely identifies each entry in the table. Since there are no non-key attributes in the `USED_IN_SURGERY` relation (all attributes are part of the primary key), there are no partial dependencies. Therefore, the `USED_IN_SURGERY` relation is in **2NF**.

{ `SurgeryID`, `EquipmentID` } is the primary key, and there are no non-key attributes in the relation. Since there are no non-key attributes, there can be no transitive dependencies. Therefore, the `USED_IN_SURGERY` relation is in **3NF**.

The only functional dependency in this relation is:

$\{\texttt{SurgeryID}, \texttt{EquipmentID}\} \rightarrow \{\texttt{SurgeryID}, \texttt{EquipmentID}\}$ (trivially, the primary key determines itself)

Since { `SurgeryID`, `EquipmentID` } is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have { `SurgeryID`, `EquipmentID` } as the determinant, and { `SurgeryID`, `EquipmentID` } is the

candidate key, we conclude that the USED_IN_SURGERY relation is in **BCNF**.

2.2.25 USED_IN_TEST

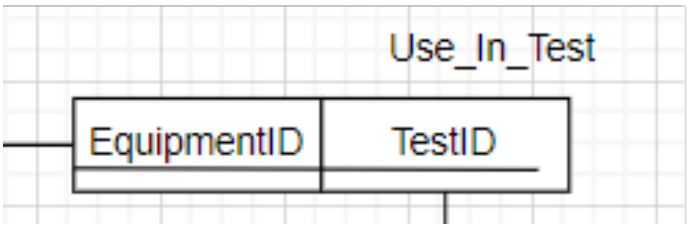


Figure 2.25: relation USED_IN_TEST

The USED_IN_TEST relation describes the many-to-many relationship between TEST and EQUIPMENT. It has the following columns:

- TestID (Primary Key, Foreign Key referencing the TEST relation, int, NOT NULL)
- EquipmentID (Primary Key, Foreign Key referencing the EQUIPMENT relation, int, NOT NULL)

All attributes in this relation (TestID and EquipmentID) contain only atomic values, and each column holds a single type of data. Each row is uniquely identifiable by the composite primary key { TestID, EquipmentID }. Therefore, the USED_IN_TEST relation is in **1NF**.

The primary key of this relation is a composite key { TestID, EquipmentID }, which uniquely identifies each entry in the table. Since there are no non-key attributes in the USED_IN_TEST relation (all attributes are part of the primary key), there are no partial dependencies. Therefore, the USED_IN_TEST relation is in **2NF**.

{ TestID, EquipmentID } is the primary key, and there are no non-key attributes in the relation. Since there are no non-key attributes, there can be no transitive dependencies. Therefore, the USED_IN_TEST relation is in **3NF**.

The only functional dependency in this relation is:

$$\{\texttt{TestID}, \texttt{EquipmentID}\} \rightarrow \{\texttt{TestID}, \texttt{EquipmentID}\} \quad \text{(trivially, the primary key determines itself)}$$

Since { TestID, EquipmentID } is a candidate key for this relation, it satisfies the condition for **BCNF** that every determinant must be a candidate key. Since all functional dependencies have { TestID, EquipmentID } as the determinant, and { TestID, EquipmentID } is the candidate key, we conclude that the USED_IN_TEST relation is in **BCNF**.

Chapter 3

Bonus 1: System design

3.1 System Overview and Requirements

The Hospital Management System is designed to provide a comprehensive solution for managing hospital operations, patient care, and administrative tasks. Based on the detailed database schema provided, our system needs to handle complex relationships between various entities including patients, medical staff, departments, and medical procedures.

3.1.1 Core Requirements

The system must address the following key requirements derived from the database structure:

- **Patient Management**: Complete patient lifecycle from registration to discharge
- **Medical Staff Management**: Doctor, nurse, and technician scheduling and assignment
- **Department Operations**: Management of different hospital departments and their resources
- **Medical Procedures**: Handling of surgeries, diagnostic tests, and treatments
- **Resource Management**: Room, bed, and equipment tracking
- **Financial Operations**: Billing, insurance, and payment processing

3.2 Architectural Approach

3.2.1 Selection of Microservices Architecture

After careful analysis of the system requirements and database schema, we have chosen a microservices architecture for the following reasons:

1. **Data Independence**: The database schema shows clear boundaries between different domains (patient data, medical procedures, billing, etc.)
2. **Scalability Requirements**: Different components of the system require different scaling capabilities:

- Patient registration system needs to handle peak loads during morning hours
- Billing system requires batch processing capabilities
- Medical record system needs high availability and real-time access

3. **Security Isolation**: Different components require different security levels:
   - Patient medical records require HIPAA compliance
   - Billing information needs financial-grade security
   - Administrative functions require audit trails

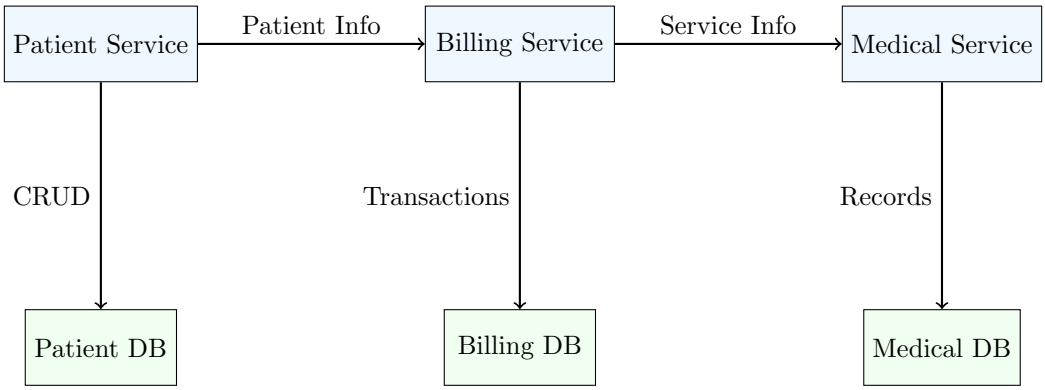


Figure 3.1: High-Level Service Architecture with Data Flow

3.3 Detailed Service Architecture

3.3.1 Patient Management Service

The Patient Management Service is designed to handle all patient-related operations, directly mapping to the PATIENT_RECORD table in our database schema. This service implements the following key functionalities:

- Patient registration and profile management
- Medical history tracking
- Appointment scheduling
- Emergency contact management

```
class PatientService {
private:
    DatabaseConnection db;
    CacheService cache;
    EventBus eventBus;

public:
    Optional<Patient> getPatientById(int patientId) {
        // Implementation details for patient retrieval
```

```
        // Includes caching and database access
    }

    Patient createPatient(PatientDTO dto) {
        // Implementation for patient creation
        // Includes validation and event publishing
    }
};
```

Listing 3.1: Patient Service Core Implementation

3.3.2 Medical Service Architecture

The Medical Service handles all medical procedures, tests, and treatments. This service integrates with the following database tables:

- DIAGNOSTIC_TEST
- SURGERY
- TREATMENT
- PRESCRIPTION

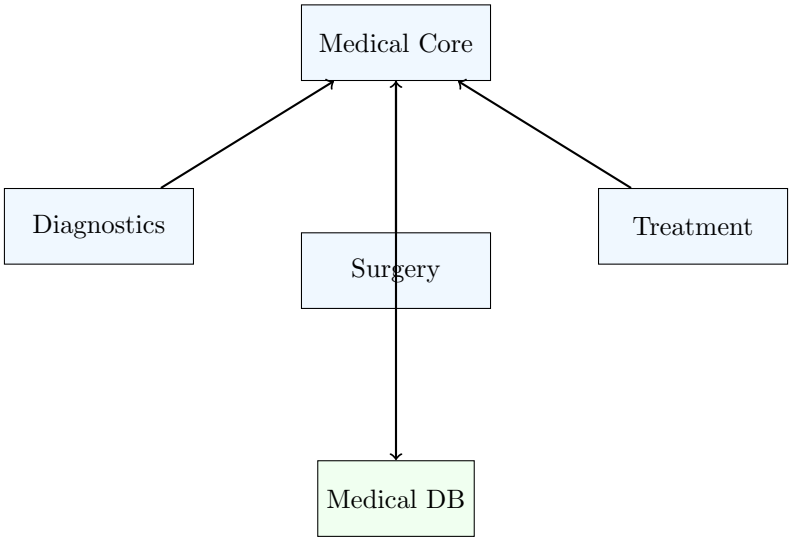


Figure 3.2: Medical Service Internal Architecture

Chapter 4

Introduction to Application Architecture

4.1 Overview

This document outlines the architectural design for the Hospital Management System, based on the comprehensive database schema provided. The architecture is designed to support:

- Multiple user roles (Admin, Doctor, Nurse, Technician, Receptionist)
- Complex medical workflows
- Secure patient data management
- Billing and insurance processing
- Equipment and resource management

4.2 Architectural Style Selection

We have chosen a **Microservices Architecture** with the following justification:

- **Scale**: Independent scaling of services
- **Security**: Isolated security domains
- **Deployment**: Independent deployment
- **Technology**: Flexible technology stack

Chapter 5

System Architecture

5.1 High-Level Architecture

Based on our comprehensive hospital database schema, we have designed a multi-tiered architecture that reflects the complex relationships between medical staff, patients, and hospital resources.

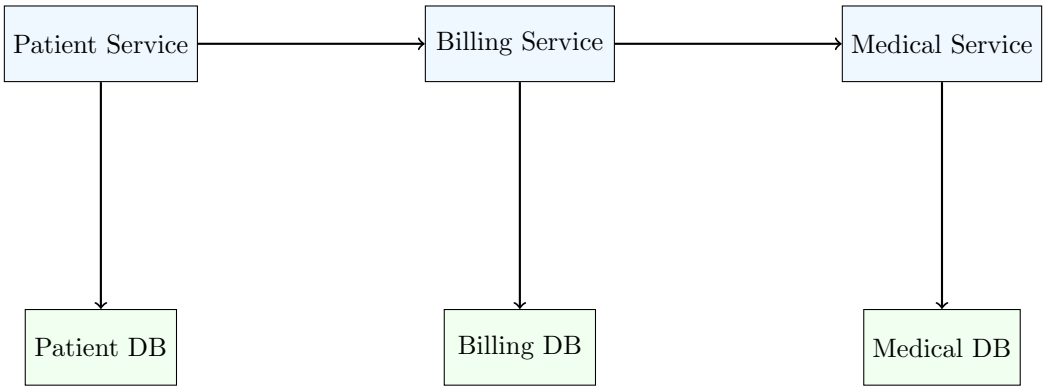


Figure 5.1: High-Level System Architecture

5.2 Service Decomposition and Core Components

The system is divided into core modules that directly map to our database entities:

5.2.1 Patient Service Layer

1. **Core Functionalities**
    - Patient registration and management
    - Medical history tracking
    - Appointment scheduling
2. **Database Mappings**

- Direct mapping to $PATIENT_RECORD$ table
- Medical history relationships
- Appointment management tables

5.2.2 Medical Service Layer

1. **Core Functionalities**
   - Diagnostic test management
   - Treatment planning
   - Prescription management
2. **Database Integration**
   - SURGERY scheduling with doctor and room assignments
   - $DIAGNOSTICTEST$ management
   - Equipment allocation through $USEDINSURGERY$ and $USEDINTEST$ relations

5.2.3 Employee Management Layer

1. **Hierarchical Structure**
   - EMPLOYEE table serves as the base entity with core attributes
   - Specialized roles (DOCTOR, NURSE, TECHNICIAN) inherit from base EMPLOYEE
   - Department associations through DepartmentID foreign key
   - Strict data integrity through CHECK constraints on Gender and Date fields

5.2.4 Resource Management Layer

1. **Physical Resources**
   - ROOM and BED tables with one-to-many relationship
   - EQUIPMENT tracking with maintenance schedules
   - Department-specific resource allocation through Dno foreign key
2. **Billing Components**
   - Invoice generation
   - Payment processing
   - Insurance claims management
   - Integration with patient and service records

5.3 Integration Architecture

The architecture ensures seamless integration between different layers through:

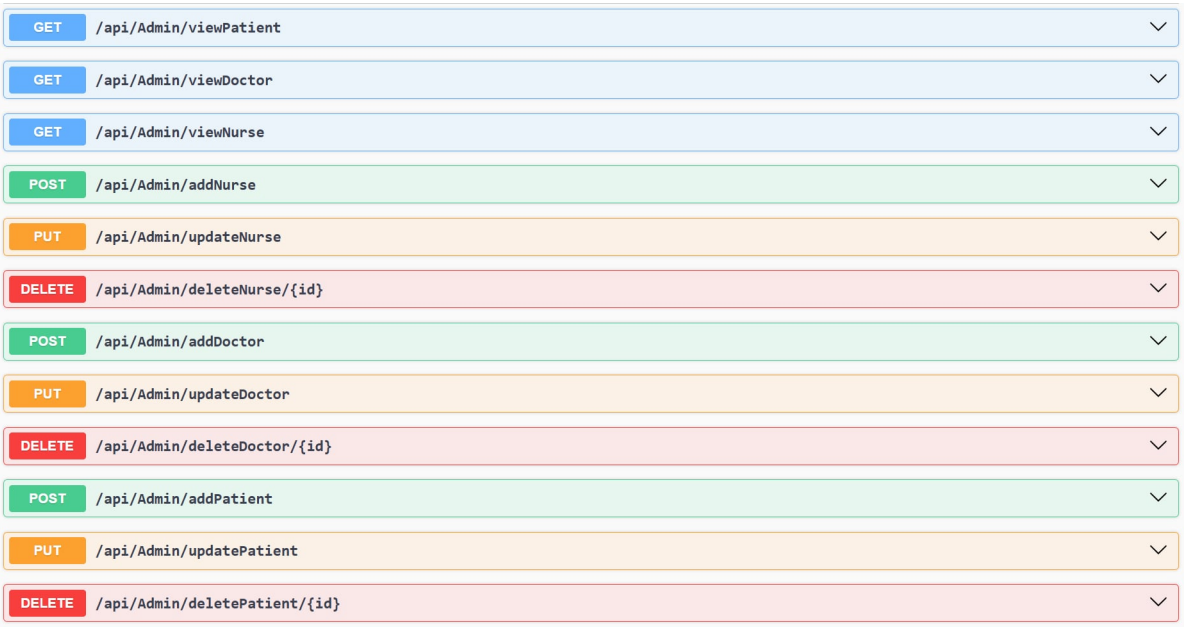
- Standardized API interfaces
- Event-driven communication
- Consistent data mapping
- Transaction management across services

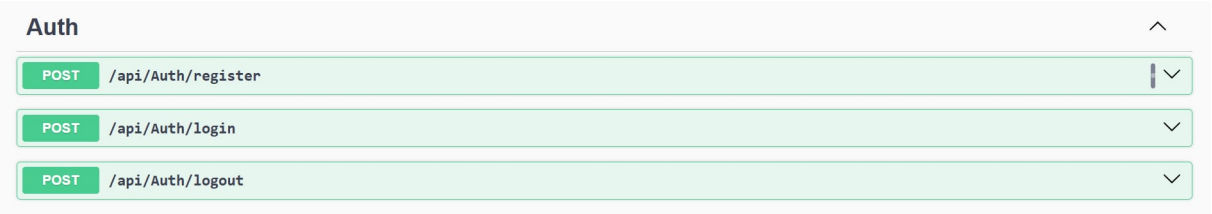
Chapter 6

API Design

6.1 API Overview

The Hospital Management System API is designed to support the database schema and role-based access control defined in our system. All endpoints follow RESTful principles and include proper authentication and authorization.





More details related to API design and other relevant matters, please have a look at project's GitHub repository: https://github.com/Phiung23/ASM-DB

Chapter 7

Data Flow Architecture

7.1 Core Data Flows

Based on our database constraints and relationships:

7.1.1 Patient Flow

1. Patient Registration
   - Creates $PATIENTRECORD$ with mandatory fields (Name, DOB, Gender)
   - Establishes emergency contact information
   - Validates against defined CHECK constraints
2. Medical History
   - Links to $DIAGNOSTICTEST$ and SURGERY records
   - Maintains referential integrity through foreign keys
   - Tracks treatment progression

7.1.2 Resource Allocation Flow

1. Room Assignment
   - Validates department association through Dno foreign key
   - Manages bed availability through BED table
   - Ensures proper resource distribution
2. Equipment Management
   - Tracks through $MAINTAIN$ relationship with $TECHNICIAN$
   - Ensures availability for procedures
   - Maintains maintenance schedules

Chapter 8

Security Architecture

8.1 Role-Based Security Model

Our security architecture directly mirrors the employee hierarchy in our database schema:

8.1.1 Employee Role Hierarchy

1. **Doctor Access Level**
   - Full access to patient medical records
   - Authority to schedule $SURGERY$ and $DIAGNOSTICTEST$
   - Access restricted by $DepartmentID$ association
   - Can view but not modify other doctors' records
2. **Nurse Access Level**
   - Read access to $PATIENTRECORD$
   - Limited write access to medical history
   - Department-specific access based on Dno
   - Cannot modify $SURGERY$ schedules
3. **Technician Access Level**
   - Full access to $EQUIPMENT$ and $MAINTAIN$ tables
   - Read-only access to $SURGERY$ schedule
   - Limited to equipment within assigned departments

8.2 Data Protection Strategy

Based on our database constraints:

- Patient data protected through strict access controls
- Medical records encrypted at rest
- Audit trails for all modifications
- Compliance with healthcare data protection standards

Chapter 9

System Integration Architecture

9.1 Database Integration Patterns

Our integration architecture reflects the complex relationships in our schema:

9.1.1 Core Integration Points

1. **Patient Management Integration**
   - Centralized $PATIENTRECORD$ as the source of truth
   - Bi-directional synchronization with medical history
   - Real-time updates for emergency contacts
   - Integration with external healthcare systems
2. **Resource Management Integration**
   - Real-time ROOM and BED availability updates
   - Equipment maintenance scheduling system
   - Department resource allocation tracking
   - Integration with inventory management
3. **Staff Management Integration**
   - Employee scheduling system integration
   - Department staffing level monitoring
   - Certification and specialization tracking
   - Integration with payroll systems

Chapter 10

Performance Optimization

10.1 Database Optimization Strategy

Based on our schema structure and relationships:

10.1.1 Query Optimization

1. **Index Strategy**
    - Primary indexes on all ID fields
    - Composite indexes for SURGERY scheduling
    - Covering indexes for frequently accessed patient data
    - Specialized indexes for equipment tracking
2. **Partition Strategy**
    - Patient records partitioned by admission date
    - Historical data archival process
    - Department-wise partitioning for resources
    - Equipment maintenance history partitioning

10.2 Performance Monitoring

- Real-time monitoring of query performance
- Resource utilization tracking
- Department-specific load analysis
- Equipment usage optimization

Chapter 11

Deployment Strategy

11.1 Infrastructure Requirements

Based on our database scale and complexity:

11.1.1 Database Tier

1. **Primary Database Cluster**
    - High-availability MS SQL Server cluster
    - Real-time replication for disaster recovery
    - Dedicated storage for patient records
    - Separate instances for different departments
2. **Caching Layer**
    - Distributed cache for patient records
    - Equipment availability cache
    - Room and bed status caching
    - Staff schedule caching

11.2 Scaling Strategy

- Horizontal scaling for application servers
- Vertical scaling for database instances
- Department-wise resource allocation
- Load balancing based on department activity

Chapter 12

Data Flow and Process Management

12.1 Core Business Processes

Based on our database relationships and constraints from the ERD:

12.1.1 Patient Journey Management

1. **Patient Registration Process**
   - Initial entry into $PATIENTRECORD$ with mandatory fields
   - Validation against defined constraints:
     - Date of birth must be valid (current date)
     - Gender must be one of ('M', 'F', 'O')
     - Contact information must be unique
     - Emergency contact is mandatory
   - Assignment to appropriate department based on DEPARTMENT table
   - Creation of associated medical history records
2. **Medical Procedure Scheduling**
   - Complex coordination between multiple tables:
     - DOCTOR availability check
     - ROOM and BED allocation
     - EQUIPMENT reservation
     - NURSE assignment
   - Constraint validations:
     - Doctor must be certified for procedure type
     - Room must be in correct department (Dno foreign key)
     - Equipment must be available and maintained

12.2 Resource Allocation Workflow

Based on our ROOM, BED, and EQUIPMENT relationships:

12.2.1 Physical Resource Management

1. **Room Assignment Process**
   - Hierarchical structure following database design:
     - DEPARTMENT ← ROOM ← BED relationship chain
     - Department capacity constraints
     - Room type compatibility checks
   - Status tracking and updates:
     - Real-time bed availability
     - Room maintenance scheduling
     - Department occupancy monitoring
2. **Equipment Lifecycle Management**
   - Based on EQUIPMENT and MAINTAIN tables:
     - Regular maintenance scheduling
     - Usage tracking in procedures
     - Technician assignment
   - Integration points:
     - Surgery equipment requirements
     - Diagnostic test equipment allocation
     - Maintenance schedule coordination

Chapter 13

Staff Management Architecture

13.1 Employee Hierarchy Implementation

Based on our employee-related tables (EMPLOYEE, DOCTOR, NURSE, TECHNICIAN):

13.1.1 Role-Based Operations

1. **Medical Staff Management**
   - Hierarchical structure implementation:
     - Base EMPLOYEE attributes
     - Specialized roles with additional attributes
     - Department associations
   - Qualification tracking:
     - Doctor specializations and certifications
     - Nurse specialties
     - Technician expertise areas
2. **Department Operations**
   - Based on DEPARTMENT table relationships:
     - Staff assignment tracking
     - Resource allocation
     - Workload distribution
   - Operational constraints:
     - Minimum staffing requirements
     - Specialty coverage requirements
     - Emergency response capabilities

Chapter 14

System Integration Patterns

14.1 Cross-Functional Processes

Based on our complex table relationships:

14.1.1 Medical Service Integration

1. **Diagnostic Process Flow**
    - Integration points:
        - Patient record access
        - Equipment allocation
        - Staff scheduling
        - Result management
    - Data flow constraints:
        - Test result validation
        - Patient history updates
        - Equipment usage tracking
2. **Treatment Management**
    - Process integration:
        - Surgery scheduling
        - Resource allocation
        - Staff assignment
        - Recovery monitoring
    - System constraints:
        - Resource availability checks
        - Staff qualification validation
        - Patient condition monitoring

Chapter 15

Quality Assurance and Monitoring

15.1 Data Integrity Management

Based on our database constraints and relationships:

15.1.1 Data Validation Framework

1. **Patient Data Integrity**
    - Primary validation rules:
        - Demographic data completeness
        - Medical history consistency
        - Treatment record accuracy
        - Emergency contact verification
    - Cross-reference validations:
        - Doctor-Patient relationships
        - Treatment-Diagnosis consistency
        - Medication history tracking
2. **Resource Data Accuracy**
    - Equipment tracking integrity:
        - Maintenance schedule adherence
        - Usage history accuracy
        - Availability status consistency
    - Facility management validation:
        - Room occupancy tracking
        - Bed allocation accuracy
        - Department resource distribution

15.2 Performance Monitoring Framework

Based on system usage patterns and database load:

15.2.1 System Health Monitoring

1. **Database Performance Metrics**
   - Critical monitoring points:
     - Query response times
     - Transaction throughput
     - Resource utilization
     - Lock contention rates
   - Performance thresholds:
     - Peak load handling
     - Resource allocation efficiency
     - System response times
2. **Operational Efficiency**
   - Department-wise metrics:
     - Staff utilization rates
     - Equipment usage efficiency
     - Patient throughput
   - Resource optimization:
     - Room occupancy rates
     - Equipment allocation efficiency
     - Staff scheduling effectiveness

Here's the rewritten content in LaTeX format, balancing paragraphs and bullet points:

Chapter 16

Self-assessment

16.1 Idea

The database design need to be able to effectively manages critical information on hospital operations, departments, employees, patient records, medical tests, surgeries, billing, insurance, and various other related aspects of hospital management.
From the database, patient and family can see:

- Personal Information
- Doctors and meeting schedules, Nurse
- Medical Records and Treatment
- Billing, Payment and the money they need to pay

Also, information of employees are stored, for management:

- Information
- Specialities for Doctor or Nurse assignments
- Tasks
- Department they belongs to
- Other such as Start date, Salary,.....

Moreover, Operations and Facilities are handled in the database with detailed information:

- The department facilities belong to
- Technicians that maintain special facilities (equipment)
- Operation: Performers, Patients and Results

The web application needs to provide user-friendly interface with data displayed to suitable users based on their roles. Server needs to handle API and retrieve data from database efficiently to provide to clients.

16.2 Reasoning

16.2.1 Database Design

1. Modular Table Design

The schema is broken into separate tables to model real-world entities effectively (e.g., `EMPLOYEE`, `DEPARTMENT`, `PATIENT_RECORD`, `TECHNICIAN`, `DOCTOR`, `SURGERY`, etc.). This modular approach ensures separation of concerns while maintaining logical relationships.

- **Entity Isolation:** Each real-world entity is represented as a distinct table. Examples include:
  - `EMPLOYEE` for employee records.
  - `PATIENT_RECORD` for storing patient data.
  - `DEPARTMENT` for hospital departments.
  - `TECHNICIAN, DOCTOR, NURSE` for specialized roles related to the employee table.
- **Logical Relationships:** These separate tables are interconnected via foreign keys to maintain real-world relationships, such as assigning employees to departments or linking patient medical records with diagnostics and billing.

2. Normalization Principles

The database schema adheres to normalization principles to avoid redundant data and ensure database efficiency and scalability.

- **Avoiding Redundancy:** Data redundancy is minimized by designing tables that separate distinct entities.
  - `PATIENT_RECORD` contains only patient information, isolated from other transactional data.
  - `BILLING`, `INSURANCE`, and `PAYMENT` are distinct but logically connected through foreign keys.
- **Efficient Data Retrieval:** Logical grouping ensures the database scales easily as data volume grows while maintaining relational integrity.

3. Referential Integrity through Foreign Keys

The schema enforces referential integrity by using foreign key constraints to ensure data dependencies remain consistent and valid.

- **Foreign Key Constraints:**
  - The `DEPARTMENT` table uses `ManageID` to establish a foreign key relationship to `EMPLOYEE`.
  - The specialized employee tables, `TECHNICIAN`, `DOCTOR`, and `NURSE`, reference `EMPLOYEE` via foreign keys.
  - `SURGERY`, `DIAGNOSTIC_TEST`, and `EQUIPMENT` leverage foreign keys to track dependencies and data usage.

4. Cascade Actions

The schema implements logical cascading rules to ensure proper handling of deletions or updates without leaving orphan data.

- **ON DELETE CASCADE:** Deleting a parent record cascades deletion to dependent child records.
- **ON UPDATE CASCADE:** Updating a parent key leads to automatic updates in dependent child keys.
- **NO ACTION:** Prevents data inconsistencies by restricting updates or deletions unless explicitly handled.

5. Data logic protection

To safeguard against wrong data operation leading to the unexpected behaviours and protect data integrity, the database uses triggers and procedures. Triggers are to keep the logic flows of data and protect data through transaction. Each procedure handles data and raise error in case of mis-written data.

6. Summary

By integrating modular table design, normalization principles, referential integrity with foreign keys, cascading rules, and error prevention checks, the schema effectively models the real-world operational structure of a hospital. This design ensures logical separation, scalability, consistency, and reliability in database operations.

16.2.2 Web application

The web application provides a seamless and secure user experience by implementing a user login system with role-based access control to ensure appropriate data visibility. It authorizes users through a login page, verifies credentials securely, and restricts access to authorized resources only.

The web application features the following key components:

1. Authentication & Authorization

Login Mechanism: Users can securely log into their accounts using their credentials. Passwords are securely encrypted to ensure data security.

Role-Based Access Control (RBAC): Users are assigned specific roles (e.g., Administrator, Registered User, Guest User) that determine the level of access to features and data.

- **Administrator:** Has full access to manage the database and user permissions.
- **Registered Users:** Access personal and role-specific data.
- **Guest Users:** Have limited visibility and functionality.

2. APIs for Client-Server Communication

The web application exposes APIs to allow client-server interaction. These APIs enable clients to send requests, upload data, or retrieve information as needed. The APIs follow RESTful design principles and allow for efficient communication.

Key functionalities provided by the APIs:

- **User Registration & Authentication:** Allow new users to register and log in securely.
- **Data Uploads:** Allow users to submit data securely to the server.
- **Data Retrieval:** APIs are available for authorized users to fetch user-specific or role-specific information.

3. Server Backend

The server's backend is designed using [**Insert Technology Here, e.g., Node.js, Flask, Django, .NET, etc.**]. This backend communicates with the database layer, validates user requests, enforces authorization checks, and ensures secure communication with the client application.

Key backend functionalities:

- **Database Communication:** Efficient querying and transaction processing using [**Insert Database Type, e.g., SQL Server, MySQL, MongoDB, etc.**].
- **Data Validation:** All incoming data is validated to avoid SQL injection or other forms of attack.
- **Error Handling:** The server ensures minimal disruption to user experience by gracefully handling unexpected errors.

4. User Interface

The web application provides a user-friendly interface that is intuitive, accessible, and responsive. The interface includes:

- An organized and intuitive navigation menu.
- Dashboard customization based on user roles.
- Feedback mechanisms for better user interaction, such as notifications or error prompts.
-

16.3 Practical Implementation

16.3.1 Implementation Process

1. Database Deployment

- The hospital database schema was deployed on the SQL Server environment using SQL Server Management Studio (SSMS).
- Tables, views, and stored procedures were created according to the finalized schema design.

- Indexes were applied to key fields (e.g., patient ID, doctor ID) to improve data retrieval efficiency.

2. Initiate Web Application
In the terminal, run the server JavaScript file:

```
node server.js
```

All operations are then performed by user's interactions. Through each information or action of an user, APIs are sent to server to perform CRUD, display information and announce errors.

16.3.2 Evaluation

Through testing procedures of multiple users accessing the website via multiple clients host, these evaluations have been conclude:

- The server is capable of handling multiple request from clients by responding with suitable html
- Security has been handled and modulized in the auth.js file. The user needs to login to retrieve the dashboard view corresponding to their roles. User of a role cannot access viewpoint and data authority of different roles.
- The Interface is easy to use, with comprehensible notation provided.
- Response's time is negligible.
- User actions have been confirmed to produce expected result in the Database via APIs.

Chapter 17

Future Scalability and Evolution

17.1 System Growth Management

Managing the growth of a database system requires a forward-thinking approach that prioritizes extensibility and adaptability. This section outlines key strategies for scalability and functional evolution, enriched with examples and implementation details.

17.1.1 Scalability Planning

Effective scalability planning involves addressing both data growth and system performance through practical strategies:

- **Database Growth Management:**
  - *Expanding Patient Records:* As the patient database grows, partitioning strategies such as vertical and horizontal partitioning can be applied to improve performance.
    - Example SQL Query:

```
ALTER TABLE PatientRecords
PARTITION BY RANGE (AdmissionDate) (
    PARTITION p1 VALUES LESS THAN ('2025-01-01'),
    PARTITION p2 VALUES LESS THAN ('2030-01-01')
);
```

  - *Archiving Historical Data:* Data older than a certain period can be archived to a secondary table or external storage to keep the primary database responsive.
    - Example SQL Query:

```
INSERT INTO ArchivedRecords SELECT *
FROM PatientRecords
WHERE AdmissionDate < '2020-01-01';

DELETE FROM PatientRecords
WHERE AdmissionDate < '2020-01-01';
```

- *Supporting New Departments:* New departments can be added by creating reusable schema designs that include tables like 'Department' and linking them via foreign keys.
  - Example SQL Schema:

```
CREATE TABLE Department (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(255),
    HeadOfDepartment INT
);

ALTER TABLE Staff ADD FOREIGN KEY (DepartmentID)

REFERENCES Department(DepartmentID);
```

- **Performance Optimization:**
  - *Query Optimization:* Use explain plans and indexes to improve complex queries.
    - Example Query Analysis:

```
EXPLAIN ANALYZE
SELECT * FROM PatientRecords
WHERE LastName = 'Doe';
```

      Adding an index for faster lookups:

```
CREATE INDEX idx_lastname ON PatientRecords(LastName);
```

  - *Index Strategy Evolution:* Use composite indexes for frequently joined columns.
  - *Dynamic Resource Scaling:* Implement database sharding or load balancers to distribute workloads effectively.

17.1.2 Functional Evolution

To address evolving functional needs, the database system must support new use cases and integrations:

- **System Enhancements:**
  - *New Medical Procedures:* Add tables for robotic surgery metadata, linked to existing patient records.
    - Example Schema:

```
CREATE TABLE RoboticSurgery (
    SurgeryID INT PRIMARY KEY,
    PatientID INT,
    RobotModel VARCHAR(255),
    ProcedureDate DATE,
    FOREIGN KEY (PatientID) REFERENCES PatientRecords(PatientID)
);
```

- *Enhanced Security Measures:* Implement column-level encryption for sensitive data like social security numbers or payment details.
  - Example SQL:

```
UPDATE PaymentDetails
SET CreditCardNumber = AES_ENCRYPT('1234567812345678', 'encryption_key')
```

- **Integration Capabilities:**
  - *API Integration:* Develop REST APIs for billing and insurance systems.
    - Example API Endpoint:

```
POST /api/bill
Body:
{
    "patient_id": 123,
    "amount_due": 500.00,
    "due_date": "2024-12-31"
}
```

  - *Data Standards:* Adopt standards like HL7 for medical record interoperability.

17.1.3 Addressing Untapped Database Components

Several tables, including **BILL**, **INSURANCE**, **PAYMENT**, **DEPARTMENT**, and **ROOM**, remain underutilized. Future development should focus on their integration:

- *Financial Workflows:* Utilize the **BILL** and **PAYMENT** tables for automated invoicing and payment tracking.
- *Insurance Claims:* Implement workflows to interact with the **INSURANCE** table for seamless claims processing.
  - Example Query:

```
SELECT * FROM Insurance
WHERE PatientID = 123 AND ClaimStatus = 'Pending';
```

- *Resource Management:* Use the **DEPARTMENT** and **ROOM** tables to optimize resource allocation and scheduling.
  - Example Query:

```
SELECT RoomID, Availability
FROM Room
WHERE DepartmentID = 5 AND Availability = 'Available';
```