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

Database Systems - CO2013

Assignment 2 Report - Group 6

Hospital Management System

Advisor(s):	Vo Thi Kim Anh	
Student(s):	Tran Dang Hien Long	ID 2252449
	Tran Dinh Dang Khoa	ID 2211649
	Tran Phu Duc	ID 2210814
	Nguyen Nhat Huy	ID 2053042

HO CHI MINH CITY, MAY 2025

Contents

PROJECT TIMELINE

Date	Version	Changes	Person in charge
11.4.2025	1.0	List out functional, non-functional and devices needed	Team
5.2.2025	1.1	Fix Technician entity	Tran Dang Hien Long

1 Database Management System Installation and Usage

1.1 Choosing and Installing the DBMS

Based on the requirements of Assignment 2, the chosen Database Management System (DBMS) for installation and use is **MySQL**. MySQL was selected for several reasons: it is a widely used, robust, and well-supported open-source relational database system. Its compatibility with Django is excellent and provides the necessary features to manage the structured data required by the Hospital Management System project.

The installation and configuration process for MySQL was carried out using **Docker** and **Docker Compose**. This approach was chosen over a direct system installation for its significant advantages. Furthermore, given that the team had prior experience with Docker but limited or no prior experience with direct MySQL installation and configuration on individual machines, using Docker provided a familiar and controlled environment to integrate the database, making it the most practical and efficient approach for this project.

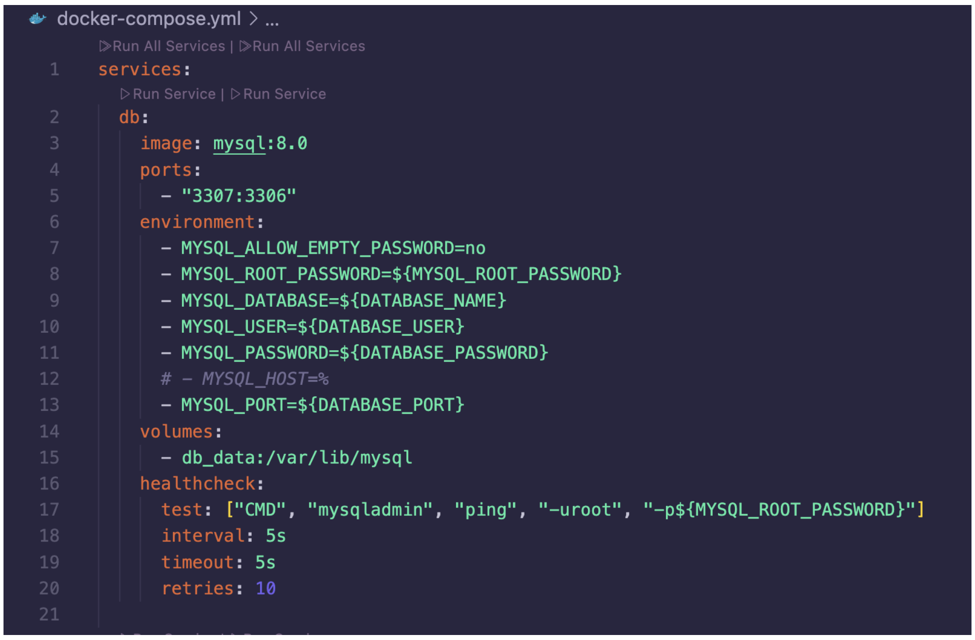
- **Isolation:** Docker containers isolate the database environment from the host system and other applications, preventing conflicts and ensuring a clean setup.
- **Consistency:** The Docker set-up guarantees that the database environment is the same across different development machines, reducing the "it works on my machine" problem.
- **Portability:** The entire database setup, including its configuration, is defined in code (docker-compose.yml, Dockerfile), making it easily portable and shareable.
- **Ease of Setup:** Compared to manually installing and configuring MySQL and its dependencies directly on the operating system, using Docker Compose simplifies the process to just building and running the defined services.

The detailed installation and configuration steps using Docker and Docker Compose are as follows:

1. **Set up the Docker environment:** The first step is to ensure that Docker Engine and Docker Compose are installed and running on the development machine. These tools provide the platform for building and managing containerized applications.

2. **Define services in docker-compose.yml:** The core of the setup is the docker-compose.yml file, which orchestrates the different services that make up the application. In this project, it defines two main services: `db` for the MySQL database and `app` for the Django application.

3. **db Service Configuration:**



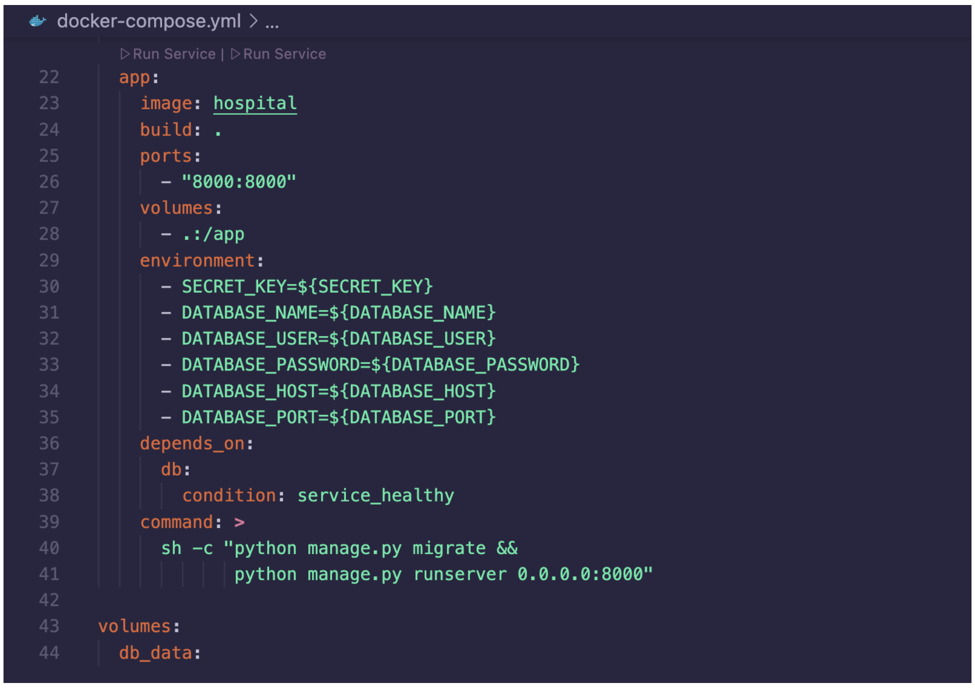
```yaml
services:
  db:
    image: mysql:8.0
    ports:
      - "3307:3306"
    environment:
      - MYSQL_ALLOW_EMPTY_PASSWORD=no
      - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${DATABASE_NAME}
      - MYSQL_USER=${DATABASE_USER}
      - MYSQL_PASSWORD=${DATABASE_PASSWORD}
      # - MYSQL_HOST=%
      - MYSQL_PORT=${DATABASE_PORT}
    volumes:
      - db_data:/var/lib/mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-uroot", "-p${MYSQL_ROOT_PASSWORD}"]
      interval: 5s
      timeout: 5s
      retries: 10

```

docker-compose.yml > ...
▷Run All Services | ▷Run All Services
▷Run Service | ▷Run Service

- `image: mysql:8.0`: Specifies that the `db` service will use the official MySQL 8.0 Docker image. This image contains a pre-built MySQL server environment.
- `ports: - "3307:3306"`: Maps port 3307 on the host machine to port 3306 inside the `db` container. This allows accessing the MySQL server from the host machine using port 3307, while the application inside the Docker network connects to it on its default port 3306.
- `environment:`: This section configures the MySQL server using environment variables, whose values are loaded from the `.env` file
- `volumes: - db_data:/var/lib/mysql`: This line mounts a named volume (`db_data`) to the `/var/lib/mysql` directory inside the container. This is crucial for data persistence; the actual database files are stored on the host machine (managed by Docker) in the `db_data` volume, so they survive container restarts, removals, or updates.
- `healthcheck:`: Configures a check to determine if the MySQL service is healthy and ready to accept connections. This is used by the `app` service's `depends_on`

to ensure the database is fully operational before the application container starts.

4. **app Service Configuration:** (Depends on `db`)



```yaml
  app:
    image: hospital
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    environment:
      - SECRET_KEY=${SECRET_KEY}
      - DATABASE_NAME=${DATABASE_NAME}
      - DATABASE_USER=${DATABASE_USER}
      - DATABASE_PASSWORD=${DATABASE_PASSWORD}
      - DATABASE_HOST=${DATABASE_HOST}
      - DATABASE_PORT=${DATABASE_PORT}
    depends_on:
      db:
        condition: service_healthy
    command: >
      sh -c "python manage.py migrate &&
             python manage.py runserver 0.0.0.0:8000"

volumes:
  db_data:
```

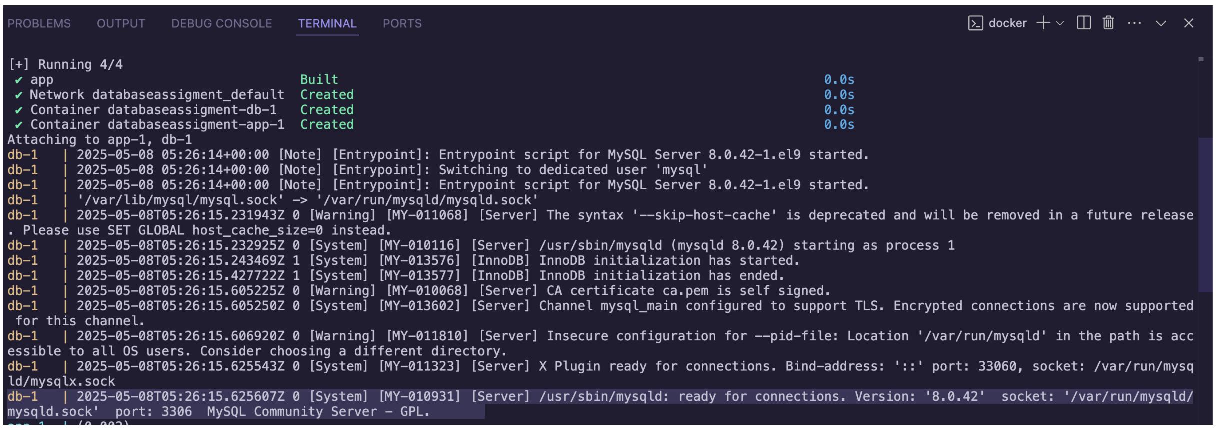
- `build: .`: Instructs Docker Compose to build the image for this service using the Dockerfile located in the current directory (.).
- `depends_on: db: condition: service_healthy`: Ensures that the `app` container will only start after the `db` container is running and has passed its health check, guaranteeing the database is available when the application tries to connect.

5. **Build and run containers:** Navigate to the project's root directory in the terminal and execute the command `docker compose up −−build`.

- `−−build`: This flag ensures that Docker Compose builds the image for the `app` service using the Dockerfile before starting the containers. The Dockerfile handles installing Python dependencies (from `requirements.txt`) and copying the project code into the container.
- `docker compose up`: This command starts the services defined in `docker-compose.yml`. Docker Compose handles creating the network, starting the `db` container, waiting for it to become healthy, and then starting the

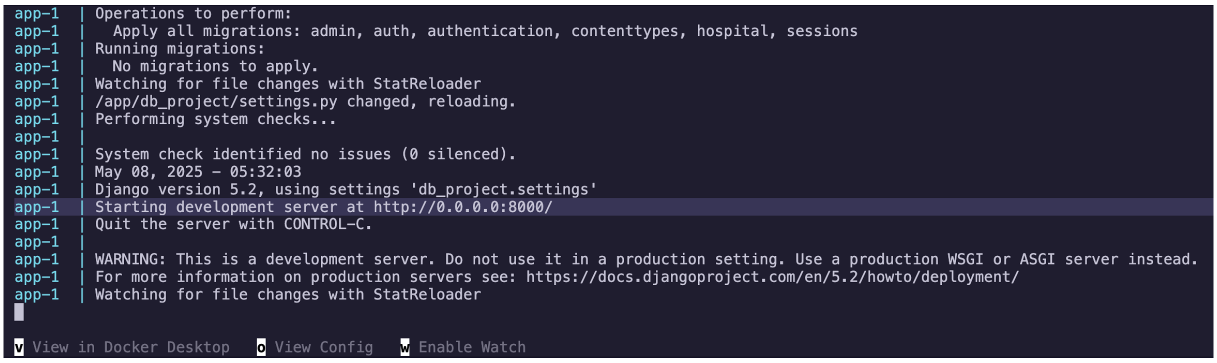
`app` container. The command specified in the `app` service (`sh -c "python manage.py migrate && python manage.py runserver 0.0.0.0:8000"`) is then executed, which includes applying database migrations and starting the Django development server.

6. **Indicators that the docker-compose services are healthy and running:**



```
[+] Running 4/4
 ✔ app                                  Built                                              0.0s
 ✔ Network databaseassigment_default   Created                                            0.0s
 ✔ Container databaseassigment-db-1    Created                                            0.0s
 ✔ Container databaseassigment-app-1   Created                                            0.0s
Attaching to app-1, db-1
db-1   | 2025-05-08 05:26:14+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.42-1.el9 started.
db-1   | 2025-05-08 05:26:14+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
db-1   | 2025-05-08 05:26:14+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.42-1.el9 started.
db-1   | '/var/lib/mysql/mysql.sock' -> '/var/run/mysqld/mysqld.sock'
db-1   | 2025-05-08T05:26:15.231943Z 0 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future release
. Please use SET GLOBAL host_cache_size=0 instead.
db-1   | 2025-05-08T05:26:15.232925Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.42) starting as process 1
db-1   | 2025-05-08T05:26:15.243469Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
db-1   | 2025-05-08T05:26:15.427722Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
db-1   | 2025-05-08T05:26:15.605225Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
db-1   | 2025-05-08T05:26:15.605250Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now supported
 for this channel.
db-1   | 2025-05-08T05:26:15.606920Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is acc
essible to all OS users. Consider choosing a different directory.
db-1   | 2025-05-08T05:26:15.625543Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 33060, socket: /var/run/mysq
ld/mysqlx.sock
db-1   | 2025-05-08T05:26:15.625607Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.42'  socket: '/var/run/mysqld/
mysqld.sock'  port: 3306  MySQL Community Server - GPL.
```

Figure 1.1: The highlighted line from the db-1 container's logs explicitly states that the MySQL server is "ready for connections". This is the primary indicator that the database service is healthy and operational.



```
app-1  | Operations to perform:
app-1  |   Apply all migrations: admin, auth, authentication, contenttypes, hospital, sessions
app-1  | Running migrations:
app-1  |   No migrations to apply.
app-1  | Watching for file changes with StatReloader
app-1  | /app/db_project/settings.py changed, reloading.
app-1  | Performing system checks...
app-1  |
app-1  | System check identified no issues (0 silenced).
app-1  | May 08, 2025 - 05:32:03
app-1  | Django version 5.2, using settings 'db_project.settings'
app-1  | Starting development server at http://0.0.0.0:8000/
app-1  | Quit the server with CONTROL-C.
app-1  |
app-1  | WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.
app-1  | For more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/
app-1  | Watching for file changes with StatReloader
```

Figure 1.2: The fact that the app-1 container successfully starts the Django development server, indicated by the line Starting development server at http://0.0.0.0:8000/, confirms that it was able to connect to the database service.

You can view the definition of docker-compose.yml and Dockerfile here:

- GitHub link: https://github.com/kchan139/db-systems

1.2 Database Creation

The database is created based on the relational model designed in Assignment 1. In this project, the database schema is defined using **Django ORM** through the models.py files in the authentication and hospital applications.

The Django ORM allows defining table structures, data fields, data types, and constraints (such as primary keys and foreign keys) using Python classes. Django then automatically generates the corresponding SQL statements to create the database on the connected DBMS.

Using the Django ORM for database definition provides significant benefits. It allows developers to interact with the database using Python code, which is often more intuitive and less error-prone than writing raw SQL for schema definition. The ORM handles the translation between Python objects and database tables, abstracting away much of the complexity of database interactions. It also facilitates database migrations, making it easier to evolve the database schema as the application develops.

The steps performed:

1. **Define Models:** The authentication/models.py and hospital/models.py files contain the definitions of the database tables (models), including data fields, data types (CharField, AutoField, DateField, ForeignKey, OneToOneField, etc.), and relationships between tables. Primary key (`primary_key=True`) and foreign key (`ForeignKey`, `OneToOneField`) constraints are defined directly in the model fields.

2. **Create Migrations:** After defining or changing models, the command `python manage.py makemigrations` is used to create migration files. These files describe the necessary changes to update the database schema to match the models. In this project, the `0001_initial.py` files in the migrations directory of each app are the result of this step, containing the operations needed to create the initial tables.

3. **Apply Migrations:** The command `python manage.py migrate` is used to execute the operations in the migration files, applying the schema changes to the installed MySQL database. This command has been integrated into the app container startup script in docker-compose.yml.

2 SQL Statements

2.1 Database and Table Creation

The database schema for the Hospital Management System is defined using defined Relational Data Model. The `schema.sql` file contains the SQL commands for creating the tables (around **800-900** lines), including the entire content would be impractical for this report. Therefore, we will present representative statements to illustrate the table structure and key constraints.

Below are examples of SQL statements used to create some of the main tables in the database:

```sql
CREATE TABLE PATIENT (
    PatientID INT AUTO_INCREMENT PRIMARY KEY,
    FName VARCHAR(50) NOT NULL,
    LName VARCHAR(50) NOT NULL,
    Gender ENUM('Male', 'Female', 'Other'),
    ContactInfo VARCHAR(100),
    Address_Street VARCHAR(100),
    Address_District VARCHAR(50),
    Address_City VARCHAR(50),
    DOB DATE,
    CurrentMeds TEXT,
    EmergencyContactPhone VARCHAR(20)
);
```

- Creates the Patient table to store patient information. The table includes fields for personal details, contact information, address, date of birth, current medications, and emergency contact. PatientID is the auto-incrementing primary key.

```sql
CREATE TABLE EMPLOYEE (
    EmployeeID VARCHAR(20) PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Gender ENUM('Male', 'Female', 'Other'),
    DOB DATE,
    JobType VARCHAR(50),
    Experience INT,
    Salary DECIMAL(10, 2),
    ContactDetails VARCHAR(100),
    StartDate DATE,
    DepartmentID INT NULL, -- Made nullable for flexibility
    FOREIGN KEY (DepartmentID) REFERENCES DEPARTMENT(DepartmentID)
);
```

- Creates the EMPLOYEE table for storing employee details, including their job type, experience, salary, and their assigned department via the DepartmentID foreign key. EmployeeID is the primary key.

```sql
CREATE TABLE ROOM (
    RoomID INT AUTO_INCREMENT PRIMARY KEY,
    Type VARCHAR(50),
    Name VARCHAR(50),
    Status ENUM('Available', 'Occupied', 'Maintenance'),
    DepartmentID INT NULL, -- Assuming a room might not always be tied
   to a department or to allow department deletion
    FOREIGN KEY (DepartmentID) REFERENCES DEPARTMENT(DepartmentID)
);
```

- Creates the ROOM table to manage room information within departments. It includes the room number, type, name, status, and a foreign key DepartmentID linking it to the DEPARTMENT table.

The complete set of generated table and constraint creation statements can be found in the schema.sql file. **You can view the schema.sql file here:**

- GitHub link:
  https://github.com/kchan139/db-systems/blob/main/docs/sql_files/schema.sql

2.2 Data Manipulation

The `query.sql` file defines how to use and manage the hospital's database. It creates stored procedures—reusable routines within the database—for tasks such as adding patients, updating records, scheduling surgeries, and generating bills. These procedures help standardize workflows, reduce errors, and improve data integrity.

2.2.1 User Access Control

2.2.2 Patient Management

Patient Record Management

```sql
-- -----------------------------------------------------
CREATE PROCEDURE AddPatient (
    IN p_FName VARCHAR(50),
    IN p_LName VARCHAR(50),
    IN p_Gender ENUM('Male', 'Female', 'Other'),
    IN p_ContactInfo VARCHAR(100),
    IN p_Address_Street VARCHAR(100),
    IN p_Address_District VARCHAR(50),
    IN p_Address_City VARCHAR(50),
    IN p_DOB DATE,
    IN p_CurrentMeds TEXT,
    IN p_EmergencyContactPhone VARCHAR(20)
)
BEGIN
    INSERT INTO PATIENT (FName, LName, Gender, ContactInfo,
    Address_Street, Address_District, Address_City, DOB, CurrentMeds,
    EmergencyContactPhone)
    VALUES (p_FName, p_LName, p_Gender, p_ContactInfo, p_Address_Street,
     p_Address_District, p_Address_City, p_DOB, p_CurrentMeds,
    p_EmergencyContactPhone);
    SELECT LAST_INSERT_ID() AS PatientID;
END$$

CREATE PROCEDURE UpdatePatientDemographics (
    IN p_PatientID INT,
    IN p_FName VARCHAR(50),
    IN p_LName VARCHAR(50),
    IN p_Gender ENUM('Male', 'Female', 'Other'),
    IN p_DOB DATE,
    IN p_Address_Street VARCHAR(100),
    IN p_Address_District VARCHAR(50),
    IN p_Address_City VARCHAR(50)
)
BEGIN
    UPDATE PATIENT
    SET FName = p_FName, LName = p_LName, Gender = p_Gender, DOB = p_DOB
,
        Address_Street = p_Address_Street, Address_District =
    p_Address_District, Address_City = p_Address_City
    WHERE PatientID = p_PatientID;
END$$

CREATE PROCEDURE UpdatePatientContact (
    IN p_PatientID INT,
    IN p_ContactInfo VARCHAR(100),
    IN p_EmergencyContactPhone VARCHAR(20)
)
BEGIN
    UPDATE PATIENT
    SET ContactInfo = p_ContactInfo, EmergencyContactPhone =
    p_EmergencyContactPhone
    WHERE PatientID = p_PatientID;
END$$

CREATE PROCEDURE UpdatePatientMedications (
    IN p_PatientID INT,
    IN p_CurrentMeds TEXT
)
BEGIN
    UPDATE PATIENT
    SET CurrentMeds = p_CurrentMeds
    WHERE PatientID = p_PatientID;
END$$

CREATE PROCEDURE DeletePatient (
    IN p_PatientID INT
)
BEGIN
    -- Consider implications: related records in ALLERGIES,
    MEDICAL_HISTORY, INSURANCE, BILLING etc.
    -- For simplicity, this is a basic delete. Add cascading deletes or
    checks as needed.
    DELETE FROM ALLERGIES WHERE PatientID = p_PatientID;
```

```sql
    DELETE FROM MEDICAL_HISTORY WHERE PatientID = p_PatientID;
    -- Add more deletions from related tables if direct FKs don't cascade or if soft delete is not used.
    DELETE FROM PATIENT WHERE PatientID = p_PatientID;
END$$
```

- The procedures in this section manage the lifecycle of patient data. AddPatient is responsible for the initial creation of patient records, populating the PATIENT table. Subsequent modifications are handled by the UpdatePatient... procedures, which target specific aspects of patient information. DeletePatient ensures the removal of patient records, along with associated data in related tables, to prevent orphaned entries. These procedures collectively provide a structured interface for interacting with patient data.

Medical History, Allergies, Insurance Management:

```sql
CREATE PROCEDURE AddMedicalHistory (
    IN p_PatientID INT,
    IN p_Type VARCHAR(100),
    IN p_Description TEXT,
    IN p_Treatment TEXT,
    IN p_Stage VARCHAR(50)
)
BEGIN
    INSERT INTO MEDICAL_HISTORY (PatientID, Type, Description, Treatment, Stage)
    VALUES (p_PatientID, p_Type, p_Description, p_Treatment, p_Stage);
END$$

CREATE PROCEDURE UpdateMedicalHistory (
    IN p_PatientID INT,
    IN p_Type VARCHAR(100),
    IN p_Description TEXT,
    IN p_Treatment TEXT,
    IN p_Stage VARCHAR(50)
)
BEGIN
    UPDATE MEDICAL_HISTORY
    SET Description = p_Description, Treatment = p_Treatment, Stage = p_Stage
    WHERE PatientID = p_PatientID AND Type = p_Type;
END$$
```

```sql

CREATE PROCEDURE AddAllergy (
    IN p_PatientID INT,
    IN p_Allergy VARCHAR(100)
)
BEGIN
    INSERT INTO ALLERGIES (PatientID, Allergy)
    VALUES (p_PatientID, p_Allergy);
END$$

CREATE PROCEDURE DeleteAllergy (
    IN p_PatientID INT,
    IN p_Allergy VARCHAR(100)
)
BEGIN
    DELETE FROM ALLERGIES WHERE PatientID = p_PatientID AND Allergy =
   p_Allergy;
END$$

CREATE PROCEDURE AddInsurance (
    IN p_PatientID INT,
    IN p_PolicyNumber VARCHAR(50),
    IN p_Priority INT,
    IN p_Provider VARCHAR(100),
    IN p_Status ENUM('Active', 'Pending', 'Expired', 'Cancelled'),
    IN p_CoveragePercentage DECIMAL(5, 2),
    IN p_CoverageLimit DECIMAL(12, 2)
)
BEGIN
    INSERT INTO INSURANCE (PatientID, PolicyNumber, Priority, Provider,
   Status, CoveragePercentage, CoverageLimit)
    VALUES (p_PatientID, p_PolicyNumber, p_Priority, p_Provider,
   p_Status, p_CoveragePercentage, p_CoverageLimit);
    SELECT LAST_INSERT_ID() AS InsuranceID;
END$$

CREATE PROCEDURE UpdateInsurance (
    IN p_InsuranceID INT,
    IN p_PolicyNumber VARCHAR(50),
    IN p_Priority INT,
    IN p_Provider VARCHAR(100),
    IN p_Status ENUM('Active', 'Pending', 'Expired', 'Cancelled'),
    IN p_CoveragePercentage DECIMAL(5, 2),
    IN p_CoverageLimit DECIMAL(12, 2)
)
BEGIN
    UPDATE INSURANCE
    SET PolicyNumber = p_PolicyNumber, Priority = p_Priority, Provider =
    p_Provider, Status = p_Status,
        CoveragePercentage = p_CoveragePercentage, CoverageLimit =
    p_CoverageLimit
    WHERE InsuranceID = p_InsuranceID;
END$$

CREATE PROCEDURE DeleteInsurance (
    IN p_InsuranceID INT
)
BEGIN
    DELETE FROM COVER WHERE InsuranceID = p_InsuranceID; -- Remove links
    to bills first
    DELETE FROM INSURANCE WHERE InsuranceID = p_InsuranceID;
END$$
```

- This section defines procedures for managing patient-related medical information. It includes procedures for handling medical history (AddMedicalHistory, UpdateMedicalHistory), allergies (AddAllergy, DeleteAllergy), and insurance details (AddInsurance, UpdateInsurance, DeleteInsurance). These procedures enable the system to record, modify, and delete relevant data, ensuring a comprehensive view of the patient's medical context.

2.2.3 Staff Management

Admin: Staff Management (Doctor, Nurse, Technician):

```sql
-- ------------------------------------------------------
CREATE PROCEDURE AddEmployeeBase (
    IN p_EmployeeID VARCHAR(20),
    IN p_Name VARCHAR(100),
    IN p_Gender ENUM('Male', 'Female', 'Other'),
    IN p_DOB DATE,
    IN p_JobType VARCHAR(50),
    IN p_Experience INT,
    IN p_Salary DECIMAL(10, 2),
    IN p_ContactDetails VARCHAR(100),
    IN p_StartDate DATE,
    IN p_DepartmentID INT
)
BEGIN
    INSERT INTO EMPLOYEE (EmployeeID, Name, Gender, DOB, JobType, Experience, Salary, ContactDetails, StartDate, DepartmentID)
    VALUES (p_EmployeeID, p_Name, p_Gender, p_DOB, p_JobType, p_Experience, p_Salary, p_ContactDetails, p_StartDate, p_DepartmentID);
END$$

CREATE PROCEDURE UpdateEmployeeBase (
    IN p_EmployeeID VARCHAR(20),
    IN p_Name VARCHAR(100),
    IN p_Gender ENUM('Male', 'Female', 'Other'),
    IN p_DOB DATE,
    IN p_Experience INT,
    IN p_Salary DECIMAL(10, 2),
    IN p_ContactDetails VARCHAR(100),
    IN p_StartDate DATE,
    IN p_DepartmentID INT
)
BEGIN
    UPDATE EMPLOYEE
    SET Name = p_Name, Gender = p_Gender, DOB = p_DOB, Experience = p_Experience, Salary = p_Salary,
        ContactDetails = p_ContactDetails, StartDate = p_StartDate, DepartmentID = p_DepartmentID
    WHERE EmployeeID = p_EmployeeID;
END$$

CREATE PROCEDURE AddDoctor (
    IN p_EmployeeID VARCHAR(20), IN p_Name VARCHAR(100), IN p_Gender ENUM('Male', 'Female', 'Other'), IN p_DOB DATE,
    IN p_Experience INT, IN p_Salary DECIMAL(10, 2), IN p_ContactDetails VARCHAR(100), IN p_StartDate DATE, IN p_DepartmentID INT,
    IN p_Specialty VARCHAR(100), IN p_Certificate VARCHAR(255)
)
BEGIN
    CALL AddEmployeeBase(p_EmployeeID, p_Name, p_Gender, p_DOB, 'Doctor', p_Experience, p_Salary, p_ContactDetails, p_StartDate, p_DepartmentID);
    INSERT INTO DOCTOR (EmployeeID, Specialty, Certificate) VALUES (p_EmployeeID, p_Specialty, p_Certificate);
```

```
END$$

CREATE PROCEDURE UpdateDoctor (
    IN p_EmployeeID VARCHAR(20), IN p_Name VARCHAR(100), IN p_Gender
   ENUM('Male', 'Female', 'Other'), IN p_DOB DATE,
    IN p_Experience INT, IN p_Salary DECIMAL(10, 2), IN p_ContactDetails
    VARCHAR(100), IN p_StartDate DATE, IN p_DepartmentID INT,
    IN p_Specialty VARCHAR(100), IN p_Certificate VARCHAR(255)
)
BEGIN
    CALL UpdateEmployeeBase(p_EmployeeID, p_Name, p_Gender, p_DOB,
   p_Experience, p_Salary, p_ContactDetails, p_StartDate, p_DepartmentID
   );
    UPDATE DOCTOR SET Specialty = p_Specialty, Certificate =
   p_Certificate WHERE EmployeeID = p_EmployeeID;
END$$

CREATE PROCEDURE DeleteDoctor (IN p_EmployeeID VARCHAR(20))
BEGIN
    DELETE FROM ASSIGN_DOC WHERE DoctorID = p_EmployeeID;
    DELETE FROM PERFORM_SURGERY WHERE DoctorID = p_EmployeeID;
    DELETE FROM DOCTOR WHERE EmployeeID = p_EmployeeID;
    DELETE FROM EMPLOYEE WHERE EmployeeID = p_EmployeeID;
END$$

CREATE PROCEDURE AddNurse (
    IN p_EmployeeID VARCHAR(20), IN p_Name VARCHAR(100), IN p_Gender
   ENUM('Male', 'Female', 'Other'), IN p_DOB DATE,
    IN p_Experience INT, IN p_Salary DECIMAL(10, 2), IN p_ContactDetails
    VARCHAR(100), IN p_StartDate DATE, IN p_DepartmentID INT,
    IN p_Specialty VARCHAR(100)
)
BEGIN
    CALL AddEmployeeBase(p_EmployeeID, p_Name, p_Gender, p_DOB, 'Nurse',
    p_Experience, p_Salary, p_ContactDetails, p_StartDate,
   p_DepartmentID);
    INSERT INTO NURSE (EmployeeID, Specialty) VALUES (p_EmployeeID,
   p_Specialty);
END$$

CREATE PROCEDURE UpdateNurse (
    IN p_EmployeeID VARCHAR(20), IN p_Name VARCHAR(100), IN p_Gender
   ENUM('Male', 'Female', 'Other'), IN p_DOB DATE,
    IN p_Experience INT, IN p_Salary DECIMAL(10, 2), IN p_ContactDetails
     VARCHAR(100), IN p_StartDate DATE, IN p_DepartmentID INT,
    IN p_Specialty VARCHAR(100)
)
BEGIN
    CALL UpdateEmployeeBase(p_EmployeeID, p_Name, p_Gender, p_DOB,
   p_Experience, p_Salary, p_ContactDetails, p_StartDate, p_DepartmentID
   );
    UPDATE NURSE SET Specialty = p_Specialty WHERE EmployeeID =
   p_EmployeeID;
END$$

CREATE PROCEDURE DeleteNurse (IN p_EmployeeID VARCHAR(20))
BEGIN
    DELETE FROM ASSIGN_NURSE WHERE NurseID = p_EmployeeID;
    DELETE FROM PERFORM_TEST WHERE NurseID = p_EmployeeID;
    DELETE FROM NURSE WHERE EmployeeID = p_EmployeeID;
    DELETE FROM EMPLOYEE WHERE EmployeeID = p_EmployeeID;
END$$

CREATE PROCEDURE AddTechnician (
    IN p_EmployeeID VARCHAR(20), IN p_Name VARCHAR(100), IN p_Gender
   ENUM('Male', 'Female', 'Other'), IN p_DOB DATE,
    IN p_Experience INT, IN p_Salary DECIMAL(10, 2), IN p_ContactDetails
     VARCHAR(100), IN p_StartDate DATE, IN p_DepartmentID INT,
    IN p_Specialty VARCHAR(100)
)
BEGIN
    CALL AddEmployeeBase(p_EmployeeID, p_Name, p_Gender, p_DOB, '
   Technician', p_Experience, p_Salary, p_ContactDetails, p_StartDate,
   p_DepartmentID);
    INSERT INTO TECHNICIAN (EmployeeID, Specialty) VALUES (p_EmployeeID,
     p_Specialty);
END$$

CREATE PROCEDURE UpdateTechnician (
    IN p_EmployeeID VARCHAR(20), IN p_Name VARCHAR(100), IN p_Gender
   ENUM('Male', 'Female', 'Other'), IN p_DOB DATE,
    IN p_Experience INT, IN p_Salary DECIMAL(10, 2), IN p_ContactDetails
     VARCHAR(100), IN p_StartDate DATE, IN p_DepartmentID INT,
    IN p_Specialty VARCHAR(100)
)
BEGIN
```

```
    CALL UpdateEmployeeBase(p_EmployeeID, p_Name, p_Gender, p_DOB, p_Experience, p_Salary, p_ContactDetails, p_StartDate, p_DepartmentID);
    UPDATE TECHNICIAN SET Specialty = p_Specialty WHERE EmployeeID = p_EmployeeID;
END$$

CREATE PROCEDURE DeleteTechnician (IN p_EmployeeID VARCHAR(20))
BEGIN
    DELETE FROM MAINTAINS WHERE TechID = p_EmployeeID;
    DELETE FROM TECHNICIAN WHERE EmployeeID = p_EmployeeID;
    DELETE FROM EMPLOYEE WHERE EmployeeID = p_EmployeeID;
END$$
```

- The staff management procedures prioritize data integrity by enforcing consistent data handling through base procedures and managing relationships between the EMPLOYEE table and role-specific tables (DOCTOR, NURSE, TECHNICIAN). The Delete... procedures also address the removal of associated records, minimizing the risk of orphaned data.

Assign Doctors/Nurses to Patients:

```
CREATE PROCEDURE AssignDoctorToPatient (IN p_DoctorID VARCHAR(20), IN p_PatientID INT)
BEGIN
    INSERT INTO ASSIGN_DOC (DoctorID, PatientID) VALUES (p_DoctorID, p_PatientID);
END$$

CREATE PROCEDURE RemoveDoctorFromPatient (IN p_DoctorID VARCHAR(20), IN p_PatientID INT)
BEGIN
    DELETE FROM ASSIGN_DOC WHERE DoctorID = p_DoctorID AND PatientID = p_PatientID;
END$$

CREATE PROCEDURE AssignNurseToPatient (IN p_NurseID VARCHAR(20), IN p_PatientID INT)
BEGIN
    INSERT INTO ASSIGN_NURSE (NurseID, PatientID) VALUES (p_NurseID, p_PatientID);
END$$
```

```

CREATE PROCEDURE RemoveNurseFromPatient (IN p_NurseID VARCHAR(20), IN
    p_PatientID INT)
BEGIN
    DELETE FROM ASSIGN_NURSE WHERE NurseID = p_NurseID AND PatientID =
    p_PatientID;
END$$
```

- The procedures effectively manage the relationships between Doctors/Nurses and Patients. AssignDoctorToPatient and AssignNurseToPatient establish these relationships, while RemoveDoctorFromPatient and RemoveNurseFromPatient dissolve them, ensuring accurate tracking of patient-provider assignments.

2.2.4 Billing and Insurance

Billing and Insurance:

```
CREATE PROCEDURE GenerateBill (
    IN p_PatientID INT,
    IN p_InitialAmount DECIMAL(12, 2),
    IN p_DateIssued DATE,
    IN p_DueDate DATE
)
BEGIN
    INSERT INTO BILLING (PatientID, DateIssued, InitialAmount,
    CoverAmount, FinalAmount, DueDate, Status)
    VALUES (p_PatientID, p_DateIssued, p_InitialAmount, 0.00,
    p_InitialAmount, p_DueDate, 'Pending');
    SELECT LAST_INSERT_ID() AS BillingID;
END$$

CREATE PROCEDURE RecalculateBillCoverage (IN p_BillingID INT)
BEGIN
    DECLARE v_InitialAmount DECIMAL(12,2);
    DECLARE v_TotalCover DECIMAL(12,2) DEFAULT 0.00;

    SELECT InitialAmount INTO v_InitialAmount
    FROM BILLING
    WHERE BillingID = p_BillingID;

    SELECT COALESCE(SUM(
                LEAST(
```

```sql
                    (v_InitialAmount * I.CoveragePercentage) / 100.0,
                    I.CoverageLimit
                )
            ), 0.00)
    INTO v_TotalCover
    FROM INSURANCE I
    JOIN COVER C ON I.InsuranceID = C.InsuranceID
    WHERE C.BillingID = p_BillingID AND I.Status = 'Active';

    IF v_TotalCover > v_InitialAmount THEN
        SET v_TotalCover = v_InitialAmount;
    END IF;

    UPDATE BILLING
    SET CoverAmount = v_TotalCover,
        FinalAmount = v_InitialAmount - v_TotalCover
    WHERE BillingID = p_BillingID;
END$$

CREATE PROCEDURE LinkInsuranceToBillAndRecalculate (IN p_BillingID INT,
   IN p_InsuranceID INT)
BEGIN
    -- Check if patient associated with bill has this insurance
    DECLARE v_PatientID_Bill INT;
    DECLARE v_PatientID_Insurance INT;
    DECLARE v_InsuranceActive BOOLEAN;

    SELECT PatientID INTO v_PatientID_Bill FROM BILLING WHERE BillingID
   = p_BillingID;
    SELECT PatientID, (Status = 'Active') INTO v_PatientID_Insurance,
   v_InsuranceActive FROM INSURANCE WHERE InsuranceID = p_InsuranceID;

    IF v_PatientID_Bill IS NOT NULL AND v_PatientID_Insurance IS NOT
   NULL AND v_PatientID_Bill = v_PatientID_Insurance AND
   v_InsuranceActive THEN
        INSERT IGNORE INTO COVER (BillingID, InsuranceID) VALUES (
   p_BillingID, p_InsuranceID);
        CALL RecalculateBillCoverage(p_BillingID);
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Insurance policy
   cannot be linked or is not active for this patient''s bill.';
    END IF;
END$$
```

- This section defines stored procedures to manage patient billing and insurance coverage. GenerateBill creates new billing records, RecalculateBillCoverage calculates the amount covered by insurance, considering coverage percentages and limits, and LinkInsuranceToBillAndRecalculate links insurance policies to bills and triggers the coverage recalculation. These procedures ensure accurate billing and proper application of insurance benefits, with LinkInsuranceToBillAndRecalculate also including validation to ensure the insurance policy is valid for the patient.
- **RecalculateBillCoverage**: Updates insurance coverage calculations for an existing bill
  - Gets original bill amount and all insurance with their coverage percent
  - For each insurance: MIN(percentage_coverage, absolute_limit)

```
LEAST(
    (v_InitialAmount * I.CoveragePercentage) / 100.0,
    I.CoverageLimit
)
```

  - Ensures total coverage never exceeds bill amount
  - Updates final patient responsibility amount
- **LinkInsuranceToBillAndRecalculate**: Link insurance to a bill to cover for the cost (must be called before recalculating the bill)
  - Gets the patient's bill and patient's insurance
  - Checks if patient associated with bill has this insurance, it not, signal an error
  - If yes, insert the pair into the junction table (ignoring duplicates)

2.2.5 Diagnostics and Surgery

Diagnostic Tests Management:

```
CREATE PROCEDURE OrderDiagnosticTest (
    IN p_PatientID INT,
    IN p_NurseID_Performer VARCHAR(20), -- Nurse who will perform/is associated
    IN p_TestName VARCHAR(100),
    IN p_TestDescription TEXT,
    IN p_TestDate DATETIME
)
BEGIN
    DECLARE v_TestID INT;
    INSERT INTO DIAGNOSTIC_TEST (Name, Description, Date, Results)
    VALUES (p_TestName, p_TestDescription, p_TestDate, NULL);
    SET v_TestID = LAST_INSERT_ID();

    IF p_NurseID_Performer IS NOT NULL THEN
        INSERT INTO PERFORM_TEST (TestID, NurseID, PatientID)
        VALUES (v_TestID, p_NurseID_Performer, p_PatientID);
    END IF;
    SELECT v_TestID AS TestID;
END$$

CREATE PROCEDURE RecordTestResults (
    IN p_TestID INT,
    IN p_Results TEXT
)
BEGIN
    UPDATE DIAGNOSTIC_TEST
    SET Results = p_Results
    WHERE TestID = p_TestID;
END$$

CREATE PROCEDURE AssignEquipmentToTest (IN p_TestID INT, IN p_EquipID
   INT)
BEGIN
    INSERT INTO USE_IN_TEST (TestID, EquipID) VALUES (p_TestID,
   p_EquipID);
END$$
```

- The diagnostic test workflow is managed here. A test is ordered with OrderDiagnosticTest, results are added with RecordTestResults, and equipment usage is tracked with AssignEquipmentToTest.
- **OrderDiagnosticTest**: Creates a new diagnostic test order and links it to medical staff and patient.
  – Creates a new record in DIAGNOSTIC_TEST table with: Test name, Descrip-

tion, Scheduled date, NULL results (initially)

  – Captures the auto-generated TestID using LAST_INSERT_ID()
  – If a nurse is specified, creates a record in PERFORM_TEST table linking: Test ID, Nurse ID, Patient ID
  – Returns the new TestID to the caller

- **RecordTestResults**: Updates a test with its results after completion.
  – Updates the DIAGNOSTIC_TEST table
  – Sets the Results field for the specified TestID
  – No return value - simple update operation

- **AssignEquipmentToTest**: Links medical equipment to a specific test.
  – Creates a record in USE_IN_TEST junction table
  – Links one piece of equipment to one test
  – Can be called multiple times to assign multiple equipment to a test

Surgery Management:

```
CREATE PROCEDURE ScheduleSurgery (
    IN p_PatientID INT,
    IN p_DoctorID VARCHAR(20),
    IN p_SurgeryType VARCHAR(100),
    IN p_SurgeryDate DATETIME
)
BEGIN
    DECLARE v_SurgeryID INT;
    INSERT INTO SURGERY (Type, Date, Outcome, Complications)
    VALUES (p_SurgeryType, p_SurgeryDate, NULL, NULL);
    SET v_SurgeryID = LAST_INSERT_ID();

    INSERT INTO PERFORM_SURGERY (SurgeryID, DoctorID, PatientID)
    VALUES (v_SurgeryID, p_DoctorID, p_PatientID);
    SELECT v_SurgeryID AS SurgeryID;
END$$

CREATE PROCEDURE UpdateSurgeryOutcome (
    IN p_SurgeryID INT,
    IN p_Outcome VARCHAR(100),
    IN p_Complications TEXT
)
BEGIN
    UPDATE SURGERY
    SET Outcome = p_Outcome, Complications = p_Complications
    WHERE SurgeryID = p_SurgeryID;
END$$

CREATE PROCEDURE AssignEquipmentToSurgery (IN p_SurgeryID INT, IN
   p_EquipID INT)
BEGIN
    INSERT INTO USE_IN_SURGERY (SurgeryID, EquipID) VALUES (p_SurgeryID,
    p_EquipID);
END$$
```

- The surgery management workflow is defined here. ScheduleSurgery initiates a surgical event, UpdateSurgeryOutcome tracks the post-operative phase, and AssignEquipmentToSurgery manages resource allocation.

2.2.6 Equipment Maintenance

Equipment Maintenance (Technician):

```
CREATE PROCEDURE LogEquipmentMaintenance (
    IN p_TechID VARCHAR(20),
    IN p_EquipmentID INT,
    IN p_MaintenanceType VARCHAR(100),
    IN p_MaintenanceDate DATE
)
BEGIN
    INSERT INTO MAINTAINS (TechID, EquipmentID, MaintenanceType,
   MaintenanceDate)
    VALUES (p_TechID, p_EquipmentID, p_MaintenanceType,
   p_MaintenanceDate);
    -- Optionally update equipment status here, e.g., to 'Maintenance'
   or 'Available'
END$$

CREATE PROCEDURE UpdateEquipmentStatus (
    IN p_EquipmentID INT,
    IN p_NewStatus ENUM('Available', 'In Use', 'Maintenance', '
   Decommissioned')
)
BEGIN
    UPDATE EQUIPMENT
    SET Status = p_NewStatus
    WHERE EquipmentID = p_EquipmentID;
END$$
```

- This section defines the technician's equipment maintenance workflow. Technicians log maintenance events with LogEquipmentMaintenance and update equipment status using UpdateEquipmentStatus.

The complete set of queries command can be found in the query.sql file, automatically generated by Django's migration process based on the model definitions. **You can view the query.sql file here:**

- https://github.com/kchan139/db-systems/blob/main/docs/sql_files/query.sql

3 Reference

- GitHub Repository: https://github.com/kchan139/db-systems
- SQL files: https://github.com/kchan139/db-systems/blob/main/docs/sql_files/