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

DATABASE SYSTEM (CO2014)

Assignment 2

QUARANTINE CAMP DATABASE

Instructor: PhD. Phan Trong Nhan, CSE-HCMUT
Students: Le Thi Phuong Thao - 2252757
Huynh Ngoc Van - 2252898
Nguyen Thuy Tien - 2252806
Doan Anh Quang - 2252666
Tran Minh Hieu - 2252216

HO CHI MINH CITY, NOVEMBER 2024

Contents

1 Physical Database Design

1.1 Database Schema Explanation

PATIENT

```
CREATE TABLE PATIENT
(   PNUMBER     CHAR(8) NOT NULL PRIMARY KEY,
    PID         CHAR(8),
    FULLNAME    VARCHAR(15) NOT NULL,
    PHONE       CHAR(10),
    GENDER      CHAR(1),
    ADDRESS     VARCHAR(30),
    RISK_LEVEL  ENUM('1','2','3')
);
```

- **PNUMBER** `CHAR(8)` **NOT NULL PRIMARY KEY**
  - **Data Type**: `CHAR(8)`
  - **Reason**: Patient numbers are fixed-length identifiers consisting of 8 characters. Using `CHAR` ensures consistent storage size and optimal performance for fixed-length data.
  - **Constraints**: `NOT NULL` and `PRIMARY KEY` ensure that every patient has a unique, non-null identifier.
- **PID** `VARCHAR(12)`
  - **Data Type**: `VARCHAR(12)`
  - **Reason**: Patient identity numbers (e.g., national ID) may vary in length up to 12 characters. `VARCHAR` is suitable for variable-length alphanumeric data.
- **FULLNAME** `VARCHAR(30)` **NOT NULL**
  - **Data Type**: `VARCHAR(30)`
  - **Reason**: Allows for full names up to 30 characters, accommodating most names while saving space for shorter ones.
  - **Constraints**: `NOT NULL` ensures that the patient's name is always provided.
- **PHONE** `CHAR(10)`
  - **Data Type**: `CHAR(10)`
  - **Reason**: Phone numbers are standardized to 10 digits. Using `CHAR` ensures consistent length and storage optimization.
- **GENDER** `CHAR(1)`
  - **Data Type**: `CHAR(1)`
  - **Reason**: Stores a single character representing gender (e.g., 'M' or 'F'), optimizing storage for a single character.
- **ADDRESS** `TEXT`
  - **Data Type**: `TEXT`

  - **Reason**: Addresses can be lengthy and vary greatly in size. `TEXT` allows for variable-length strings up to 65,535 characters, accommodating detailed addresses.

- **RISK_LEVEL** `ENUM('1','2','3')`
  - **Data Type**: `ENUM`
  - **Reason**: Restricts input to predefined risk levels ('1', '2', or '3'), ensuring data consistency and integrity.

EMPLOYEE

```
Create table employee
(
    ID char(4) PRIMARY KEY,
    address varchar(255),
    phone varchar(10),
    fullname varchar(30),
    Employee_type ENUM('STAFF','VOLUNTEER','NURSE','MANAGER','DOCTOR')
);
```

- **ID** `CHAR(4)` **PRIMARY KEY**
  - **Data Type**: `CHAR(4)`
  - **Reason**: Employee IDs are fixed at 4 characters, ensuring uniformity across records.
- **ADDRESS** `VARCHAR(255)`
  - **Data Type**: `VARCHAR(255)`
  - **Reason**: Addresses can be lengthy and vary greatly in size. `TEXT` allows for variable-length strings up to 65,535 characters, accommodating detailed addresses.
- **PHONE** `CHAR(10)`
  - **Data Type**: `CHAR(10)`
  - **Reason**: Phone numbers are standardized to 10 digits. Using `CHAR` ensures consistent length and storage optimization.
- **FULLNAME** `VARCHAR(30)`
  - **Data Type**: `VARCHAR(30)`
  - **Reason**: Allows for full names up to 30 characters, accommodating most names while saving space for shorter ones.
- **EMPLOYEE_TYPE** `ENUM('STAFF','VOLUNTEER','NURSE','MANAGER','DOCTOR')`
  - **Data Type**: `ENUM`
  - **Reason**: Restricts the type of employee to predefined categories, ensuring data integrity.

BUILDING

```
    create table BUILDING
(
    BUILDING_ID VARCHAR(3) PRIMARY KEY,
    BUILDING_NAME   VARCHAR(20) UNIQUE NOT NULL
);
```

- **BUILDING_ID** `VARCHAR(3)` **PRIMARY KEY**
  - **Data Type**: `VARCHAR(3)`
  - **Reason**: Building IDs include one character to define the area and 2 digits for each building in that area.
- **BUILDING_NAME** `VARCHAR(20)` **UNIQUE NOT NULL**
  - **Data Type**: `VARCHAR(20)`
  - **Reason**: Allows for building names up to 20 characters.
  - **Constraints**: `UNIQUE` ensures each building name is distinct; `NOT NULL` mandates that every building has a name.

ROOM

```
create table ROOM
(
    ROOM_ID     CHAR(3),
    BUILDING_ID VARCHAR(3),
    ROOM_TYPE   ENUM ('Normal', 'Emergency', 'Recuperation'),           -- 1 byte
    CAPACITY    INT,
    NUMPATIENT  INT,
    PRIMARY KEY (ROOM_ID, BUILDING_ID),
    CHECK   (ROOM_ID    BETWEEN 100 AND 650),
    CHECK   (NUMPATIENT < CAPACITY),
    CONSTRAINT fk_building_id
               FOREIGN KEY (BUILDING_ID) REFERENCES BUILDING(BUILDING_ID)
               ON DELETE CASCADE
               ON UPDATE CASCADE
);
```

- **ROOM_ID** `CHAR(3)`
  - **Data Type**: `CHAR(3)`
  - **Reason**: Room numbers are standardized to 3 characters for consistency across the database, with the first digit represent the floor the room on and the others increase sequentially for each room on that floor.
- **BUILDING_ID** `VARCHAR(3)`
  - **Data Type**: `VARCHAR(3)`
  - **Reason**: References the `BUILDING_ID` from the `BUILDING` table, maintaining referential integrity.
- **ROOM_TYPE** `ENUM('Normal', 'Emergency', 'Recuperation')`

  - **Data Type**: `ENUM`
  - **Reason**: Restricts room types to predefined categories, ensuring data consistency.

- **CAPACITY** `INT`
  - **Data Type**: `INT`
  - **Reason**: Stores the maximum number of patients that the room can accommodate.

- **NUMPATIENT** `INT`
  - **Data Type**: `INT`
  - **Reason**: Tracks the current number of patients in the room.

Constraints

- **PRIMARY KEY** (`ROOM_ID`, `BUILDING_ID`) ensures each room is uniquely identified within a building.
- **CHECK** (`ROOM_ID BETWEEN 100 AND 650`) ensures room numbers fall within a valid range, reflecting possible room numbering in the facility.
- **CHECK** (`NUMPATIENT < CAPACITY`) ensures that the number of patients does not exceed the room's capacity.
- **FOREIGN KEY** (`BUILDING_ID`) references `BUILDING(BUILDING_ID)`, maintaining the relationship between rooms and buildings.

MEDICINE

```
    CREATE TABLE MEDICINE(
    MCODE    INT      PRIMARY KEY      AUTO_INCREMENT,
    MNAME    VARCHAR(30)      NOT NULL,
    EFFECT   TEXT,
    PRICE    DECIMAL(10,2),
    EXP DATE     NOT NULL
);
```

- **MCODE** `INT` **PRIMARY KEY AUTO_INCREMENT**
  - **Data Type**: `INT`
  - **Reason**: A unique numeric identifier for each medicine, automatically incremented to ensure uniqueness without manual intervention.

- **MNAME** `VARCHAR(30)` **NOT NULL**
  - **Data Type**: `VARCHAR(30)`
  - **Reason**: Stores the name of the medicine, up to 30 characters.
  - **Constraints**: `NOT NULL` ensures that every medicine has a name.

- **EFFECT** `TEXT`
  - **Data Type**: `TEXT`

  - **Reason**: Accommodates detailed descriptions of the medicine's effects, which can be lengthy.

- **PRICE** `DECIMAL(10,2)`
  - **Data Type**: `DECIMAL(10,2)`
  - **Reason**: Stores the price of the medicine with precision up to two decimal places, allowing for accurate financial calculations.
- **EXP** `DATE` **NOT NULL**
  - **Data Type**: `DATE`
  - **Reason**: Records the expiration date of the medicine.
  - **Constraints**: `NOT NULL` ensures that the expiration date is provided for safety and compliance.

TEST_RESULT

```
    CREATE TABLE TEST_RESULT (
    TEST_ID INT,
    PNUMBER  CHAR(8) NOT NULL,
    DATE_TIME DATETIME ,
    RESPIRATORY_RATE DECIMAL(5,2),
    SPO2 DECIMAL(5,2),
    PCR_ct_value DECIMAL(5,2),
    PCR_result BOOLEAN,
    QT_ct_value DECIMAL(5,2),
    QT_result BOOLEAN,
    PRIMARY KEY (PNUMBER, TEST_ID),
    CONSTRAINT fk_pnumber FOREIGN KEY (PNUMBER)
                          REFERENCES PATIENT(PNUMBER)
                          ON DELETE CASCADE
                          ON UPDATE CASCADE
);
```

- **TEST_ID** `INT`
  - **Data Type**: `INT`
  - **Reason**: Unique identifier for each test conducted on a patient.
- **PNUMBER** `CHAR(8)` **NOT NULL**
  - **Data Type**: `CHAR(8)`
  - **Reason**: References the patient's unique number.
  - **Constraints**: `NOT NULL` ensures the test result is associated with a patient.
- **DATE_TIME** `DATETIME`
  - **Data Type**: `DATETIME`
  - **Reason**: Records the exact date and time when the test was conducted.
- **RESPIRATORY_RATE** `DECIMAL(5,2)`
  - **Data Type**: `DECIMAL(5,2)`

  - **Reason**: Stores the respiratory rate with precision, allowing for values up to 999.99 breaths per minute.

- **SPO2** `DECIMAL(5,2)`
  - **Data Type**: `DECIMAL(5,2)`
  - **Reason**: Records the oxygen saturation percentage in the blood with precision.
- **PCR_ct_value** `DECIMAL(5,2)`
  - **Data Type**: `DECIMAL(5,2)`
  - **Reason**: Stores the cycle threshold (Ct) value for PCR tests, which may have decimal precision.
- **PCR_result** `BOOLEAN`
  - **Data Type**: `BOOLEAN`
  - **Reason**: Indicates the result of the PCR test; `TRUE` for positive, `FALSE` for negative.
- **QT_ct_value** `DECIMAL(5,2)`
  - **Data Type**: `DECIMAL(5,2)`
  - **Reason**: Stores the cycle threshold (Ct) value for Quick Tests, requiring decimal precision.
- **QT_result** `BOOLEAN`
  - **Data Type**: `BOOLEAN`
  - **Reason**: Indicates the result of the Quick Test; `TRUE` for positive, `FALSE` for negative.

Constraints

- **PRIMARY KEY** (`PNUMBER`, `TEST_ID`) ensures each test result for a patient is unique.
- **FOREIGN KEY** (`PNUMBER`) references `PATIENT(PNUMBER)`, maintaining referential integrity.

CONDUCT_RESULT

```
    CREATE TABLE CONDUCT_RESULT (
    TEST_ID INT,
    PNUM CHAR(8),
    TESTER_ID CHAR(4),
    PRIMARY KEY (TEST_ID, PNUM, TESTER_ID),
    CONSTRAINT fk_pnum FOREIGN KEY (PNUM, TEST_ID) REFERENCES TEST_RESULT(PNUMBER,
     TEST_ID) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT fk_testerid FOREIGN KEY (TESTER_ID) REFERENCES EMPLOYEE(ID) ON
    DELETE CASCADE ON UPDATE CASCADE
);
```

- **TEST_ID** `INT`
  - **Data Type**: `INT`

  - **Reason**: References the unique identifier of the test conducted, matching the TEST_ID in the TEST_RESULT table.

- **PNUM** CHAR(8)
  - **Data Type**: CHAR(8)
  - **Reason**: Represents the patient number, matching the PNUMBER in the PATIENT table.
- **TESTER_ID** CHAR(4)
  - **Data Type**: CHAR(4)
  - **Reason**: Identifies the employee who conducted the test, referencing the ID in the EMPLOYEE table.

Constraints

- **PRIMARY KEY** (TEST_ID, PNUM, TESTER_ID) ensures that each test conducted by a tester on a patient is uniquely identified.
- **FOREIGN KEY** (PNUM, TEST_ID) references TEST_RESULT(PNUMBER, TEST_ID), maintaining referential integrity between the test results and the conduct records.
- **FOREIGN KEY** (TESTER_ID) references EMPLOYEE(ID), ensuring that the tester is a valid employee.

IS_ASSIGNED_TO

```
    CREATE TABLE IS_ASSIGNED_TO (
   PNUM CHAR(8),
   ASSIGN_DATE DATE,
   ASSIGN_TIME TIME,
   ROOM_ID CHAR(3),
   BUILDING_ID VARCHAR(3),
   CURRENT_STATUS ENUM('1','2','3'),
   PRIMARY KEY (PNUM, ASSIGN_DATE, ASSIGN_TIME, ROOM_ID, BUILDING_ID),
   CONSTRAINT fk_room FOREIGN KEY (ROOM_ID, BUILDING_ID) REFERENCES ROOM(ROOM_ID,
    BUILDING_ID)
              ON DELETE CASCADE
              ON UPDATE CASCADE,
  constraint fk_pnum_assign_to foreign key (pnum) references patient(pnumber)
  ON DELETE CASCADE
  on UPDATE CASCADE
);
```

- **PNUM** CHAR(8)
  - **Data Type**: CHAR(8)
  - **Reason**: Represents the patient number, referencing the PNUMBER in the PATIENT table.
- **ASSIGN_DATE** DATE
  - **Data Type**: DATE
  - **Reason**: Records the date on which the patient was assigned to a room.

- **ASSIGN_TIME** `TIME`
  - **Data Type**: `TIME`
  - **Reason**: Records the exact time of assignment, allowing for precise tracking of patient movements.
- **ROOM_ID** `CHAR(3)`
  - **Data Type**: `CHAR(3)`
  - **Reason**: Identifies the room to which the patient is assigned, referencing `ROOM(ROOM_ID)`.
- **BUILDING_ID** `VARCHAR(3)`
  - **Data Type**: `VARCHAR(3)`
  - **Reason**: Identifies the building where the room is located, referencing `BUILDING(BUILDING_ID)`.
- **CURRENT_STATUS** `ENUM('1','2','3')`
  - **Data Type**: `ENUM`
  - **Reason**: Represents the patient's current health status or condition level, restricted to predefined categories for consistency.

Constraints

- **PRIMARY KEY** (`PNUM`, `ASSIGN_DATE`, `ASSIGN_TIME`, `ROOM_ID`, `BUILDING_ID`) ensures that each assignment record is unique, capturing the patient's assignment history over time.
- **FOREIGN KEY** (`ROOM_ID`, `BUILDING_ID`) references `ROOM(ROOM_ID, BUILDING_ID)`, maintaining the relationship between room assignments and actual rooms.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`, ensuring that the assignment is associated with a valid patient.
- **Trigger** `CHECK_ASSIGNED_DATE`: Validates that the assignment date is on or after the patient's admission date, preserving chronological consistency.

SYMPTOMS

```
    CREATE TABLE SYMPTOMS
(   PNUM     CHAR(8),
    SYMP_NAME VARCHAR(30),
    START_DATE DATETIME DEFAULT CURRENT_TIMESTAMP,
    END_DATE DATETIME DEFAULT CURRENT_TIMESTAMP,
    SERIOUS_LEVEL  ENUM('1','2','3'),
    PRIMARY KEY (PNUM, SYMP_NAME, START_DATE),
    CONSTRAINT SYMP1 FOREIGN KEY (PNUM)
                REFERENCES PATIENT(PNUMBER)
                ON UPDATE CASCADE
                ON DELETE CASCADE
);
```

- **PNUM** `CHAR(8)`

  - **Data Type**: `CHAR(8)`
  - **Reason**: Represents the patient number, referencing the `PNUMBER` in the `PATIENT` table.

- **SYMP_NAME** `VARCHAR(200)`
  - **Data Type**: `VARCHAR(200)`
  - **Reason**: Stores the name of the symptom experienced by the patient, accommodating longer symptom descriptions.
- **START_DATE** `DATETIME` **DEFAULT CURRENT_TIMESTAMP**
  - **Data Type**: `DATETIME`
  - **Reason**: Records when the symptom began.
  - **Constraints**: `DEFAULT CURRENT_TIMESTAMP` ensures the current date and time are recorded if no value is provided.
- **END_DATE** `DATETIME` **DEFAULT CURRENT_TIMESTAMP**
  - **Data Type**: `DATETIME`
  - **Reason**: Records when the symptom ended or was resolved.
- **SERIOUS_LEVEL** `ENUM('1','2','3')`
  - **Data Type**: `ENUM`
  - **Reason**: Indicates the severity of the symptom, with predefined levels for standardized assessment.

Constraints

- **PRIMARY KEY** (`PNUM`, `SYMP_NAME`, `START_DATE`) ensures that each symptom record for a patient is unique based on the symptom and its start time.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`, maintaining referential integrity with the patient records.
- **CHECK** (`END_DATE` $\geq$ `START_DATE`): Ensures that the end date of a symptom is not before its start date.

COMORBIDITY

```
    CREATE TABLE COMORBIDITY
(
    COMORBIDITY_NAME VARCHAR(100) PRIMARY KEY,
    COMO_DESCRIPTION VARCHAR(255)
);
```

- **COMORBIDITY_NAME** `VARCHAR(100)` **PRIMARY KEY**
  - **Data Type**: `VARCHAR(100)`
  - **Reason**: Stores the name of the comorbidity, serving as a unique identifier.

- **COMO_DESCRIPTION** `VARCHAR(255)`
  - **Data Type**: `VARCHAR(255)`
  - **Reason**: Provides a description or additional details about the comorbidity.

Constraints

- **PRIMARY KEY** (`COMORBIDITY_NAME`) ensures that each comorbidity is uniquely identifiable by its name.

PATIENT_HAS_COMORBIDITY

```
    CREATE TABLE PATIENT_HAS_COMORBIDITY
(
    PNUM CHAR(8),
    COMORBIDITY_NAME VARCHAR(100),
    PRIMARY KEY (PNUM, COMORBIDITY_NAME),
    CONSTRAINT PHC1 FOREIGN KEY (PNUM)
                REFERENCES PATIENT(PNUMBER)
                ON UPDATE CASCADE
                ON DELETE CASCADE,
    CONSTRAINT PHC2 FOREIGN KEY (COMORBIDITY_NAME)
                REFERENCES COMORBIDITY(COMORBIDITY_NAME)
                ON UPDATE CASCADE
                ON DELETE CASCADE
);
```

- **PNUM** `CHAR(8)`
  - **Data Type**: `CHAR(8)`
  - **Reason**: Represents the patient number.
- **COMORBIDITY_NAME** `VARCHAR(100)`
  - **Data Type**: `VARCHAR(100)`
  - **Reason**: References the name of the comorbidity from the `COMORBIDITY` table.

Constraints

- **PRIMARY KEY** (`PNUM`, `COMORBIDITY_NAME`) ensures that each patient-comorbidity association is unique.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`, linking the comorbidity to a valid patient.
- **FOREIGN KEY** (`COMORBIDITY_NAME`) references `COMORBIDITY(COMORBIDITY_NAME)`, ensuring that the comorbidity exists in the system.

ADMIT_DATE

```
    CREATE TABLE ADMIT_DATE
(
    PNUM CHAR(8),
    ADMITTER_ID CHAR(8),
    TRANSFERED_ADDRESS VARCHAR(255),
    ADMIT_DATE DATETIME DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (PNUM, ADMITTER_ID, ADMIT_DATE),
    CONSTRAINT A1 FOREIGN KEY (PNUM)
                REFERENCES PATIENT(PNUMBER)
                ON UPDATE CASCADE
                ON DELETE CASCADE,
    CONSTRAINT A2 FOREIGN KEY (ADMITTER_ID)
                REFERENCES EMPLOYEE(ID)
                ON UPDATE CASCADE
                ON DELETE CASCADE

);
```

- **PNUM** `CHAR(8)`
  - **Data Type**: `CHAR(8)`
  - **Reason**: Represents the patient number, matching the `PNUMBER` in the `PATIENT` table. Using `CHAR(8)` ensures a fixed-length identifier for consistency.
- **ADMITTER_ID** `CHAR(4)`
  - **Data Type**: `CHAR(4)`
  - **Reason**: Identifies the employee who admitted the patient, referencing `EMPLOYEE(ID)`. The data type `CHAR(4)` matches the fixed length of employee IDs.
- **TRANSFERED_ADDRESS** `VARCHAR(255)`
  - **Data Type**: `VARCHAR(255)`
  - **Reason**: Stores the address from which the patient was transferred. `VARCHAR(255)` allows for variable-length addresses up to 255 characters.
- **ADMIT_DATE** `DATETIME` **DEFAULT CURRENT_TIMESTAMP**
  - **Data Type**: `DATETIME`
  - **Reason**: Records the date and time of admission. The `DEFAULT CURRENT_TIMESTAMP` ensures that the current date and time are automatically recorded if no value is provided.

Constraints

- **PRIMARY KEY** (`PNUM`, `ADMITTER_ID`, `ADMIT_DATE`) ensures that each admission record is unique based on the patient, the admitting employee, and the admission date and time.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`, maintaining referential integrity with the patient records.
- **FOREIGN KEY** (`ADMITTER_ID`) references `EMPLOYEE(ID)`, ensuring that the admitter is a valid employee.

DISCHARGE_DATE

```
    CREATE TABLE DISCHARGE_DATE
(
    PNUM CHAR(8),
    DISCHARGE_ID CHAR(8),
    DISCHARGE_DAY DATETIME DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (PNUM, DISCHARGE_ID, DISCHARGE_DAY),
CONSTRAINT D1 FOREIGN KEY (PNUM)
                REFERENCES PATIENT(PNUMBER)
                ON UPDATE CASCADE
                ON DELETE CASCADE,
CONSTRAINT D2 FOREIGN KEY (DISCHARGE_ID)
                REFERENCES EMPLOYEE(ID)
                ON UPDATE CASCADE
                ON DELETE CASCADE
);
```

- **PNUM** `CHAR(8)`
- **DISCHARGE_ID** `CHAR(4)`
  - **Data Type**: `CHAR(4)`
  - **Reason**: Identifies the employee who discharged the patient, referencing `EMPLOYEE(ID)`.
- **DISCHARGE_DAY** `DATETIME` **DEFAULT CURRENT_TIMESTAMP**
  - **Data Type**: `DATETIME`
  - **Reason**: Records the date and time of discharge.

Constraints

- **PRIMARY KEY** (`PNUM`, `DISCHARGE_ID`, `DISCHARGE_DAY`) ensures that each discharge record is unique.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`.
- **FOREIGN KEY** (`DISCHARGE_ID`) references `EMPLOYEE(ID)`.

TAKE_CARE_PERIOD

```
    CREATE TABLE TAKE_CARE_PERIOD(
    PNUM    CHAR(8),
    CAREGIVER_ID    CHAR(4),
    START_DATE DATETIME,
    END_DATE    DATETIME,
    CONSTRAINT TCP1 PRIMARY KEY (PNUM, CAREGIVER_ID, START_DATE),
    CONSTRAINT TCP2 FOREIGN KEY (PNUM) REFERENCES PATIENT(PNUMBER) ON DELETE
    CASCADE ON UPDATE CASCADE,
    CONSTRAINT TCP4 FOREIGN KEY (CAREGIVER_ID) REFERENCES EMPLOYEE(ID) ON DELETE
    CASCADE ON UPDATE CASCADE,
    CONSTRAINT TCP3 CHECK (END_DATE > START_DATE)
);
```

- **PNUM** `CHAR(8)`

- **CAREGIVER_ID** `CHAR(4)`
  - **Reason**: Identifies the caregiver (employee) assigned to the patient.
- **START_DATE** `DATETIME`
  - **Reason**: Records when the caregiver started caring for the patient.
- **END_DATE** `DATETIME`
  - **Reason**: Records when the caregiver stopped caring for the patient.

Constraints

- **PRIMARY KEY** (`PNUM`, `CAREGIVER_ID`, `START_DATE`) ensures that each care period is uniquely identified.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`.
- **FOREIGN KEY** (`CAREGIVER_ID`) references `EMPLOYEE(ID)`.
- **CHECK** (`END_DATE` > `START_DATE`) ensures that the care period is valid, with the end date after the start date.

TREATMENT_RECORD

```
    CREATE TABLE TREATMENT_RECORD(
    PNUM    CHAR(8),
    TREAT_ID    INT,
    START_DATE  DATETIME    NOT NULL,
    END_DATE    DATETIME,
    RESULT TEXT,
    CONSTRAINT TR1 PRIMARY KEY (PNUM, TREAT_ID),
    CONSTRAINT TR2 FOREIGN KEY (PNUM) REFERENCES PATIENT(PNUMBER) ON DELETE
    CASCADE ON UPDATE CASCADE,
    CONSTRAINT TR3 CHECK (END_DATE > START_DATE)
);
```

- **PNUM** `CHAR(8)`
  - **Data Type**: `CHAR(8)`
  - **Reason**: Represents the patient number, consistent with `PATIENT(PNUMBER)`. Using `CHAR(8)` ensures a fixed-length identifier for efficient storage and comparison.
- **TREAT_ID** `INT`
  - **Data Type**: `INT`
  - **Reason**: A unique integer identifier for each treatment record associated with a patient. Using `INT` allows for a large number of treatment records per patient.
- **START_DATE** `DATETIME` **NOT NULL**
  - **Data Type**: `DATETIME`
  - **Reason**: Records the exact date and time when the treatment began. This is essential for tracking the patient's medical history and treatment timeline.

- **Constraints**: `NOT NULL` ensures that the start date is always provided, as it is critical information for any treatment record.

- **END_DATE** `DATETIME`
  - **Data Type**: `DATETIME`
  - **Reason**: Records when the treatment ended. This may be `NULL` if the treatment is ongoing.

- **RESULT** `TEXT`
  - **Data Type**: `TEXT`
  - **Reason**: Stores a detailed description of the outcome or effectiveness of the treatment, accommodating lengthy notes or observations made by medical staff.

Constraints

- **PRIMARY KEY** (`PNUM`, `TREAT_ID`) ensures that each treatment record for a patient is uniquely identifiable, preventing duplicate records.
- **FOREIGN KEY** (`PNUM`) references `PATIENT(PNUMBER)`, maintaining referential integrity by ensuring that every treatment record is associated with an existing patient.
- **CHECK** (`END_DATE` > `START_DATE`) ensures the treatment duration is valid by preventing the end date from being before the start date.

MAKE_TREATRECORD

```
    CREATE TABLE MAKE_TREATRECORD(
    PNUM CHAR(8),
    TREAT_ID    INT,
    DOCTOR_ID   CHAR(4),
    CONSTRAINT PK_PNUM_TREATID_DOCTORID PRIMARY KEY (PNUM, TREAT_ID, DOCTOR_ID),
    CONSTRAINT FK_PNUM_TREATID FOREIGN KEY (PNUM, TREAT_ID) REFERENCES
    TREATMENT_RECORD(PNUM, TREAT_ID) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT FK_DOCTOR_ID FOREIGN KEY (DOCTOR_ID) REFERENCES EMPLOYEE(ID) ON
    DELETE CASCADE ON UPDATE CASCADE
);
```

- **PNUM** `CHAR(8)`
  - **Data Type**: `CHAR(8)`
  - **Reason**: References the patient number, ensuring consistency with the `PATIENT` table.

- **TREAT_ID** `INT`
  - **Reason**: Identifies the specific treatment record associated with the patient.

- **DOCTOR_ID** `CHAR(4)`
  - **Data Type**: `CHAR(4)`
  - **Reason**: Represents the ID of the doctor who created or is responsible for the treatment record. Consistent with `EMPLOYEE(ID)`.

Constraints

- **PRIMARY KEY** (PNUM, TREAT_ID, DOCTOR_ID) ensures that each record of a doctor making a treatment record for a patient is unique.
- **FOREIGN KEY** (PNUM, TREAT_ID) references TREATMENT_RECORD(PNUM, TREAT_ID), maintaining the link between the treatment record and the patient.
- **FOREIGN KEY** (DOCTOR_ID) references EMPLOYEE(ID), ensuring that the doctor is a valid employee in the system.

HAS_MEDICINE

```
    CREATE TABLE HAS_MEDICINE(
    PNUM CHAR(8),
    TREAT_ID    INT,
    MCODE   INT,
    QUANTITY INT NOT NULL,
    CONSTRAINT HM1 PRIMARY KEY (PNUM, TREAT_ID, MCODE),
    CONSTRAINT HM2 FOREIGN KEY (PNUM, TREAT_ID) REFERENCES TREATMENT_RECORD(PNUM,
    TREAT_ID) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT HM3 FOREIGN KEY (MCODE) REFERENCES MEDICINE(MCODE) ON DELETE
    CASCADE ON UPDATE CASCADE
);
```

- **PNUM** CHAR(8)
  - **Data Type**: CHAR(8)
  - **Reason**: References the patient number, ensuring consistency with the PATIENT table.
- **TREAT_ID** INT
  - **Reason**: Identifies the specific treatment record associated with the patient.
- **DOCTOR_ID** CHAR(4)
  - **Data Type**: CHAR(4)
  - **Reason**: Represents the ID of the doctor who created or is responsible for the treatment record. Consistent with EMPLOYEE(ID).

Constraints

- **PRIMARY KEY** (PNUM, TREAT_ID, MCODE) ensures that each medicine prescribed in a treatment is uniquely identified.
- **FOREIGN KEY** (PNUM, TREAT_ID) references TREATMENT_RECORD(PNUM, TREAT_ID).
- **FOREIGN KEY** (MCODE) references MEDICINE(MCODE).

HEAD_DOCTOR

```
    CREATE TABLE Head_Doctor (
    ID char(4) PRIMARY KEY,
    START_DATE DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (ID) REFERENCES EMPLOYEE(ID) ON DELETE CASCADE ON UPDATE CASCADE
);
```

- **ID** `CHAR(4)` **PRIMARY KEY**
  - **Reason**: Identifies the head doctor, referencing `EMPLOYEE(ID)`.
- **START_DATE** `DATETIME` **DEFAULT CURRENT_TIMESTAMP**
  - **Reason**: Records when the doctor began their tenure as head doctor.

Constraints

- **FOREIGN KEY** (`ID`) references `EMPLOYEE(ID)`.

1.2 Trigger

```
    -- Trigger cho test ID
DELIMITER $$
DROP TRIGGER IF EXISTS trigger1_test_result$$
CREATE TRIGGER trigger1_test_result
BEFORE INSERT ON test_result
FOR EACH ROW
BEGIN
    DECLARE max_test_id INT; -- Variable to hold the maximum TEST_ID

    SELECT MAX(TEST_ID)
    INTO max_test_id
    FROM test_result
    WHERE PNUMBER = NEW.PNUMBER;

    -- If no record exists for the PNUMBER, set TEST_ID to '1'
    IF max_test_id IS NULL THEN
        SET NEW.TEST_ID = 1;
    ELSE
        -- Otherwise, increment the maximum TEST_ID by 1
        SET NEW.TEST_ID = max_test_id + 1;
    END IF;
END$$

DELIMITER ;

-- Trigger cho Treatment Record
DELIMITER $$
DROP TRIGGER IF EXISTS trigger2_treatment_record$$
CREATE TRIGGER trigger2_treatment_record
BEFORE INSERT ON treatment_record
FOR EACH ROW
BEGIN
    DECLARE max_treatment_record INT; -- Variable to hold the maximum TEST_ID

    SELECT MAX(TREAT_ID)
    INTO max_treatment_record
    FROM treatment_record
```

```
    WHERE PNUM = NEW.PNUM;

    -- If no record exists for the PNUM, set TREAT_ID to '1'
    IF max_treatment_record IS NULL THEN
        SET NEW.TREAT_ID = 1;
    ELSE
        -- Otherwise, increment the maximum treatment_record by 1
        SET NEW.TREAT_ID = max_treatment_record + 1;
    END IF;
END$$

DELIMITER ;


SET GROUP_CONCAT_MAX_LEN=32768; -- Increase the max length for GROUP_CONCAT SELECT
     GROUP_CONCAT('DROP TABLE IF EXISTS `', table_name, '`;') FROM
    information_schema.tables WHERE table_schema = 'your_database_name';




DELIMITER $$
CREATE TRIGGER CHECK_ASSIGNED_DATE
BEFORE INSERT ON IS_ASSIGNED_TO
FOR EACH ROW
BEGIN
    DECLARE ADMIT_DAY DATE;

    SELECT ADMIT_DATE INTO ADMIT_DAY
    FROM ADMIT_DATE
    WHERE PNUM = NEW.PNUM
    LIMIT 1; -- Ensures only one row is selected

    -- Check if ASSIGN_DATE is before ADMIT_DAY
    IF NEW.ASSIGN_DATE <= ADMIT_DAY THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ASSIGN_DATE must be after or equal to ADMIT_DAY';
    END IF;
END $$
DELIMITER ;

DELIMITER $$
CREATE TRIGGER CHECK_DISCHARGED_DATE
BEFORE INSERT ON DISCHARGE_DATE
FOR EACH ROW
BEGIN
    DECLARE ADMIT_DAY DATE;

  SELECT MAX(ADMIT_DATE) INTO ADMIT_DAY
    FROM ADMIT_DATE
    WHERE PNUM = NEW.PNUM;
    -- LIMIT 1; -- Ensures only one row is selected

    -- Check if ASSIGN_DATE is before ADMIT_DAY
    IF NEW.DISCHARGE_DAY <= ADMIT_DAY THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'DISCHARGE_DATE must be after or equal to ADMIT_DAY';
    END IF;
END $$
DELIMITER ;

-- Automatically generate pnumber
```

```
98  DELIMITER $$
99  CREATE TRIGGER trg_generate_pnumber
100 BEFORE INSERT ON Patient
101 FOR EACH ROW
102 BEGIN
103     DECLARE next_pnumber INT;
104     SELECT IFNULL(MAX(CAST(Pnumber AS UNSIGNED)), 0) INTO next_pnumber FROM
        Patient;
105     SET NEW.Pnumber = LPAD(next_pnumber + 1, 8, '0');
106 END;
107 $$
108 DELIMITER ;
```

1. **Trigger:** `trigger1_test_result`
   - **Purpose**: Automatically assigns a sequential `TEST_ID` to new records in the `test_result` table for each patient.
   - **Functionality**:
     - Before a new test result is inserted, the trigger checks for the maximum `TEST_ID` associated with the patient (`PNUMBER`).
     - If no previous test exists for the patient, `TEST_ID` is set to 1.
     - If previous tests exist, `TEST_ID` is set to one greater than the current maximum.
   - **Benefit**: Ensures that each patient's test results are sequentially numbered, facilitating easy tracking and management of test records.

2. **Trigger:** `trigger2_treatment_record`
   - **Purpose**: Automatically assigns a sequential `TREAT_ID` to new records in the `treatment_record` table for each patient.
   - **Functionality**:
     - Before a new treatment record is inserted, the trigger retrieves the maximum `TREAT_ID` for the patient (`PNUM`).
     - If no previous treatment exists, `TREAT_ID` is set to 1.
     - If previous treatments exist, `TREAT_ID` is incremented by one from the current maximum.
   - **Benefit**: Maintains sequential numbering of treatment records per patient, aiding in the chronological organization of treatment history.

3. **Trigger:** `CHECK_ASSIGNED_DATE`
   - **Purpose**: Validates that the `ASSIGN_DATE` in the `IS_ASSIGNED_TO` table is on or after the patient's `ADMIT_DATE`.
   - **Functionality**:
     - Before inserting a new room assignment, the trigger retrieves the patient's `ADMIT_DATE` from the `ADMIT_DATE` table.

- It compares the new `ASSIGN_DATE` with the `ADMIT_DATE`.
- If `ASSIGN_DATE` is before the `ADMIT_DATE`, the trigger raises an error and prevents the insertion.

- **Benefit**: Ensures logical consistency by preventing room assignments before a patient is admitted to the facility.

4. **Trigger:** `CHECK_DISCHARGED_DATE`

- **Purpose**: Ensures that the `DISCHARGE_DAY` in the `DISCHARGE_DATE` table is on or after the patient's latest `ADMIT_DATE`.
- **Functionality**:
  - Before inserting a new discharge record, the trigger retrieves the latest `ADMIT_DATE` for the patient.
  - It compares the new `DISCHARGE_DAY` with the `ADMIT_DATE`.
  - If `DISCHARGE_DAY` is before the `ADMIT_DATE`, the trigger raises an error and prevents the insertion.
- **Benefit**: Prevents logical inconsistencies by ensuring that a patient cannot be discharged before they are admitted.

5. **Trigger:** `trg_generate_pnumber`

- **Purpose**: Automatically generates a unique `PNUMBER` for new patients upon insertion into the `Patient` table.
- **Functionality**:
  - Before a new patient record is inserted, the trigger finds the maximum existing `PNUMBER`.
  - It increments this number by one and pads it to 8 characters with leading zeros.
  - Assigns the new `PNUMBER` to the patient record.
- **Benefit**: Automates patient number assignment, ensuring uniqueness and consistency without manual input.

1.3 Generate Data

- **Employee Table**: 35 employees divided into 5 doctors, 7 nurses, 10 volunteers, 5 staff members, and 8 managers. Employees were assigned unique IDs, realistic addresses in Ho Chi Minh City, and phone numbers.
- **Head_Doctor Table**: 1 record, designating doctor with ID '0005' as the head doctor, with a start date of '18-11-2020 09:30:00'.
- **Building Table**: 5 buildings, identified by unique IDs and descriptive names, such as 'VP - QL thuoc' and 'Toa Benh nhan'.
- **Room Table**: 40 rooms across 4 buildings. Rooms include normal, emergency, and recuperation rooms, with IDs ranging from '101' to '205' based on floor and building.

- **Medicine Table**: 7 commonly used COVID-19 medicines, including details like effect descriptions, prices, and expiration dates.
- **Comorbidity Table**: 8 common comorbidities, such as 'Diabetes' and 'Heart disease'.
- **Patient Table**: 10 patients with unique IDs, personal details, and realistic addresses in Ho Chi Minh City.
- **Test_Result Table**: 14 test results, capturing respiratory rate, SpO2, CT values, and results for PCR and QT tests. Some patients have multiple test results.
- **Patient_Has_Comorbidity Table**: 14 records assigning comorbidities to patients, with some patients having multiple comorbidities.
- **Symptoms Table**: 15 symptom records, capturing details such as symptom names, start and end dates, and severity levels.
- **Is_Assigned_To Table**: 14 records detailing room assignments based on patient risk levels and current status, with assignment dates and times.
- **Conduct_Result Table**: 14 records linking test results to the employee who conducted each test.
- **Treatment_Record Table**: 20 records capturing treatment details, including start and end dates and results for each patient.
- **Make_TreatRecord Table**: 20 records linking doctors to treatment records for each patient.
- **Has_Medicine Table**: 40 records documenting medicines prescribed during treatment, including quantities.
- **Admit_Date Table**: 12 records tracking admission dates and the employees responsible for admissions. Some patients have multiple admissions.
- **Discharge_Date Table**: 2 records of patient discharges, reflecting ongoing treatment for most patients.
- **Take_Care_Period Table**: 22 records of care periods assigned to nurses and volunteers, with start and end dates.
- **Username Table**: 5 records for employees (managers) with login credentials, simulating system authentication.

2 Store Procedure / Function /SQL

2.1 Update Patient PCR Result

Update patient PCR test to positive with null cycle threshold value for all patients whose admission date is from 01/09/2020.

```
UPDATE TEST_RESULT T
JOIN PATIENT P ON P.PNUMBER = T.PNUMBER
JOIN ADMIT_DATE A ON P.PNUMBER = A.PNUM
SET T.PCR_RESULT = '1',
    T.PCR_ct_value = NULL
WHERE A.ADMIT_DATE >= '2020-09-01';
```

Listing 1: Update Patient PCR Result

2.2 Select Patient Information

Select all the patient information whose name is 'Nguyen Van A'.

```

SELECT P.PNUMBER, P.PID, P.FULLNAME, P.PHONE, P.GENDER, P.ADDRESS, P.RISK_LEVEL,
I.ASSIGN_DATE, I.ASSIGN_TIME, I.ROOM_ID, I.BUILDING_ID, I.CURRENT_STATUS,
A.ADMITTER_ID, A.ADMIT_DATE, A.TRANSFERED_ADDRESS,
D.DISCHARGE_ID, D.DISCHARGE_DAY,
T.CAREGIVER_ID, T.START_DATE, T.END_DATE,
R.TREAT_ID, R.START_DATE, R.END_DATE, R.RESULT,
S.SYMP_NAME, S.START_DATE, S.END_DATE, S.SERIOUS_LEVEL,
C.COMORBIDITY_NAME

FROM PATIENT P
LEFT JOIN IS_ASSIGNED_TO I ON P.PNUMBER = I.PNUM
LEFT JOIN ADMIT_DATE A ON P.PNUMBER = A.PNUM
LEFT JOIN DISCHARGE_DATE D ON P.PNUMBER = D.PNUM
LEFT JOIN TAKE_CARE_PERIOD T ON P.PNUMBER = T.PNUM
LEFT JOIN TREATMENT_RECORD R ON P.PNUMBER = R.PNUM
LEFT JOIN SYMPTOMS S ON P.PNUMBER = S.PNUM
LEFT JOIN PATIENT_HAS_COMORBIDITY C ON P.PNUMBER = C.PNUM

WHERE P.FULLNAME = 'Nguyen Van A';
```

Listing 2: Select Patient Information

2.3 Testing for each patient

Write a function to calculate the testing for each patient.

- Input: Patient ID
- Output: A list of testing

```
DELIMITER $$

CREATE FUNCTION get_patient_testing(PNUM INT)
RETURNS TEXT
```

```
DETERMINISTIC
BEGIN
    DECLARE result TEXT DEFAULT '';
    DECLARE done INT DEFAULT 0;
    DECLARE test_id INT;
    DECLARE pcr_result tinyint(1);
    DECLARE pcr_ct_value decimal(5,2);
  DECLARE qt_result tinyint(1);
    DECLARE qt_ct_value decimal(5,2);
    DECLARE respiratory_rate decimal(5,2);
    DECLARE SPO2 decimal(5,2);

    -- Cursor for the query
    DECLARE test_cursor CURSOR FOR
        SELECT T.TEST_ID, T.PCR_RESULT, T.PCR_Ct_Value, T.QT_Result, T.QT_Ct_Value
    , T.respiratory_rate, T.SPO2
        FROM PATIENT P
        JOIN TEST_RESULT T ON P.PNUMBER = T.PNUMBER
        WHERE P.PNUMBER = PNUM;

    -- Handler for when the cursor reaches the end
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    -- Open the cursor
    OPEN test_cursor;

    -- Fetch loop to process rows
    fetch_loop: LOOP
        FETCH test_cursor INTO test_id, pcr_result, pcr_ct_value, qt_result,
    qt_ct_value, respiratory_rate, SPO2;
        IF done THEN
            LEAVE fetch_loop;
        END IF;

        -- Concatenate each row's data into the result
        SET result = CONCAT(
            result,
            'Test ID: ', test_id,
            ', PCR Result: ', pcr_result,
            ', PCR Ct Value: ', COALESCE(CAST(pcr_ct_value AS CHAR), 'NULL'),
      ', QT Result: ', qt_result,
            ', QT Ct Value: ', COALESCE(CAST(qt_ct_value AS CHAR), 'NULL'),
            ', respiratory_rate: ', respiratory_rate,
            ', SPO2: ', SPO2,
            '\n'
        );
    END LOOP;

    -- Close the cursor
    CLOSE test_cursor;

    -- Return the concatenated result
    RETURN result;
END$$

DELIMITER ;
```

Listing 3: Testing for each patient

2.4 Sort the nurses

Write a procedure to sort the nurses in decreasing number of patients he/she takes care in a period of time.

- Start date, End date
- A list of sorting nurses

```
DELIMITER $$
CREATE PROCEDURE num_patient_takecare_in_period(
    IN input_start_date DATE,
    IN input_end_date DATE
)
BEGIN
  IF input_end_date < input_start_date THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error: End date cannot be earlier than start date.
    ';
    END IF;
    SELECT CAREGIVER_ID, COUNT(DISTINCT PNUM) AS PATIENT_COUNT
    FROM take_care_period
    WHERE (END_DATE <= input_end_date AND START_DATE >= input_start_date)
    GROUP BY CAREGIVER_ID
    ORDER BY PATIENT_COUNT DESC;
END$$
DELIMITER ;

CALL num_patient_takecare_in_period('2024-01-31', '2024-01-31');
CALL num_patient_takecare_in_period('2024-01-01', '2024-01-31');
CALL num_patient_takecare_in_period('2024-01-31', '2024-01-01'); -- raise error '
    Error: End date cannot be earlier than start date.'
```

Listing 4: Sort the nurse in decreasing number of patients

3 Building Application

3.1 Application introduction

1. **Programming environment**: Web based application
2. **Program languages and frameworks**:
   (a) **Front end**:
      - Language: JavaScript
      - Framework: ReactJS

   (b) **Backend end**:
      - Language: Python
      - Framework: FastAPI

   (c) **Database**: MySQL
3. **Application - Database connection**: The ***'mysql.connector'*** library is used to connect the FastAPI backend to the MySQL database

```python
import mysql.connector
from mysql.connector import connection

db_config = {
    "host": "localhost",
    "user": "Manager",
    "password": "password",
    "database": "quarantine_camp",
}

# Connect to db
conn = mysql.connector.connect(**db_config)
```

Listing 5: Connect to MySQL database

3.2 Create user

1. **Log in to the database as DBA**

```
# bash
mysql -u root -p
```

   Then enter the password of root

2. **Create the "Manager" User**

```
# sql
CREATE USER 'Manager'@'localhost' IDENTIFIED BY 'ManagerPassword';
```

3. **Grant all access right to 'Manager' user**

```
# sql
GRANT ALL PRIVILEGES ON quarantine_camp.* TO 'Manager'@'localhost';
```

3.3 Requirement function

1. **Log in and Log out**
   Login endpoint to authenticate the user. If successful, establish the app database connection according to the account role. When logout, disconnect the database connection and return to the login UI.

   (a) **Front end**:

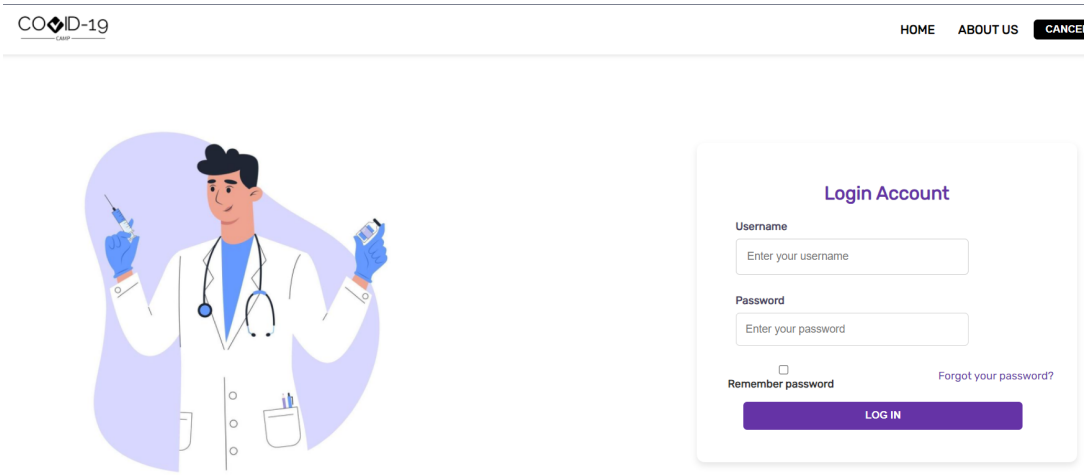


Figure 1: Login UI

   (b) **Back end**:

   - **POST Login**
     - End point: '/login'
     - Query: `"SELECT Password, Authorities FROM USERNAME WHERE Username = %s"`
     - Return:

```
{
  "message": " Login successful. Connected to app database as
    admitter1."
}
```

   - **POST Logout**
     - End point: '/logout'
     - Return:

```
{
  "message": "Logout successful. Disconnected from app database."
}
```

2. **Search patient information:** Search results include the name, phone number and information about his/her comorbidities

(a) **Front end**:

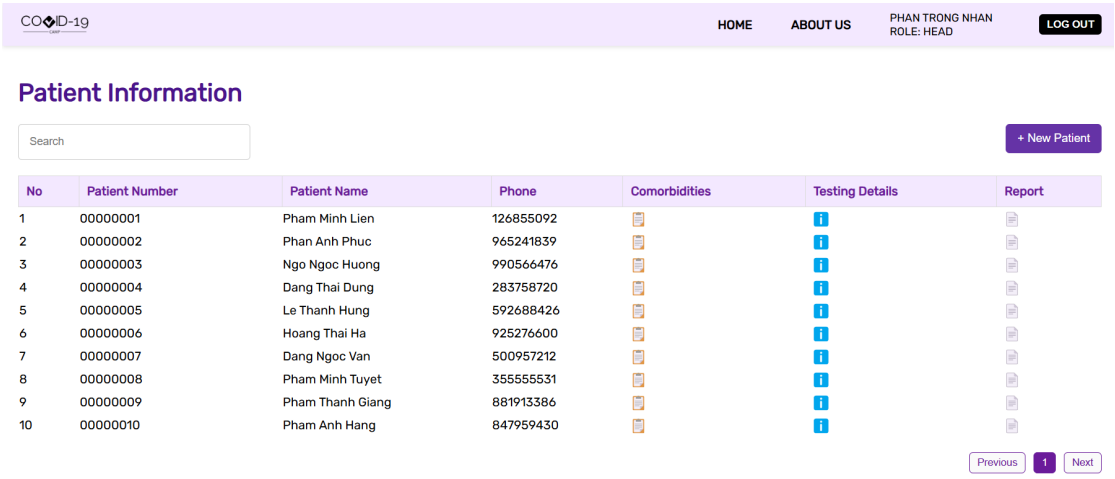


Figure 2: Search patient information UI

- First, the user interface displays the information of all patient in the quarantine camp
- User search patient by input ***name*** of patient on the search bar, the table of patient information will dynamically render base on the input, it will display information of patients whose name contains the string that user input

When user clicks the button in the comorbidities column, the information of comorbidities of the patient in corresponding row will be shown as following

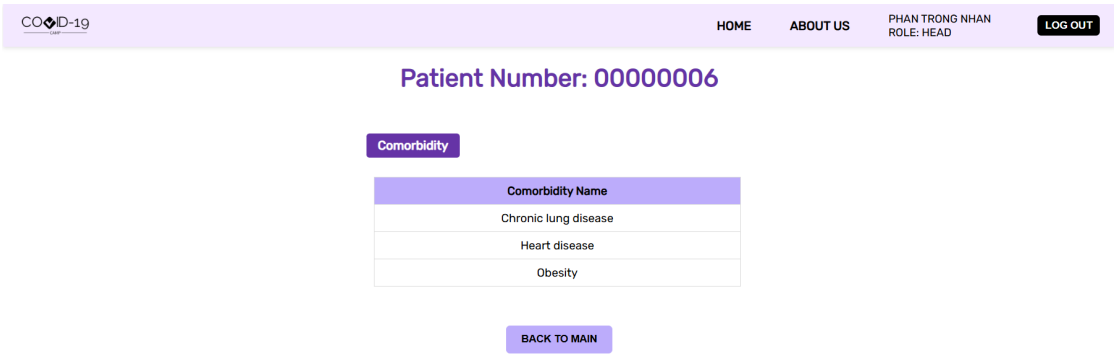


Figure 3: Patient comorbidities UI

(b) **Back end**:

- **GET All Patients**
  - Endpoint: `'/patient/all-patients'`
  - Query: `"SELECT * FROM patient"`
  - Return:

```
[
  {
    "PNUMBER": "stringst",
    "PID": "stringstring",
    "fullname": "string",
    "PHONE": "stringstr",
    "GENDER": "M",
    "address": "string",
    "RISK_LEVEL": "1"
  }
]
```


- **GET Patient By Name**
  - End point: '/patient/search/name/{name}'
  - Query: "SELECT * FROM patient WHERE LOWER(FULLNAME) LIKE LOWER(%{name}%)"
  - Return:


```
[
  {
    "PNUMBER": "stringst",
    "PID": "stringstring",
    "fullname": "string",
    "PHONE": "stringstr",
    "GENDER": "M",
    "address": "string",
    "RISK_LEVEL": "1"
  }
]
```


- **GET Comorbidity By Pnumber**
  - End point: '/patient/comorbidity/{pnum}'
  - Query: "SELECT * FROM patient_has_comorbidity WHERE PNUM = {pnum}"
  - Return:


```
[
  {
    "PNUM": "stringst",
    "COMORBIDITY_NAME": "Diabetes"
  }
]
```


3. **Add information for a new patient**

   (a) **Front end**:

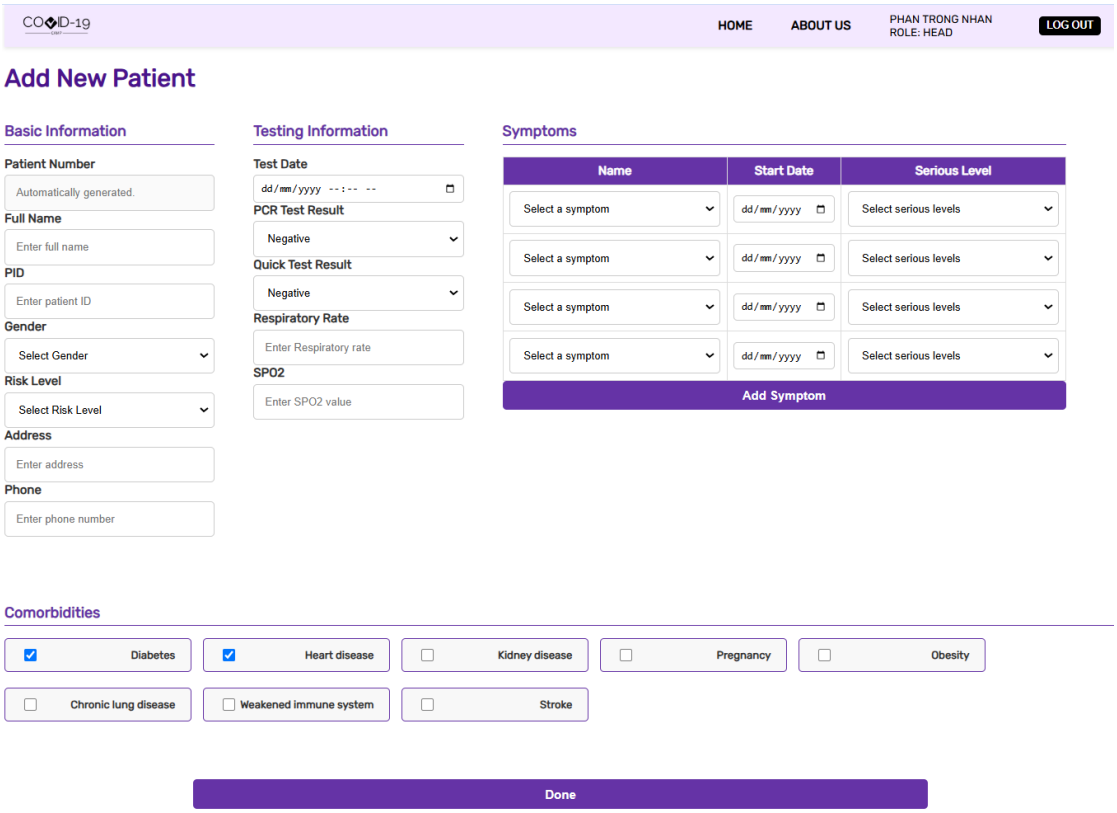


Figure 4: Add new patient UI

(b) **Back end**: **POST Insert Patient**

- End point: '`/patient/insert`'
- Request body:

```
{
  "Fullname": "string",
  "PID": "stringstring",
  "Gender": "M",
  "Risk_level": "1",
  "Address": "string",
  "Phone": "stringstr",
  "Symptom": [
    {
      "name": "string",
      "startDate": "2024-12-02T18:18:37.747Z",
      "endDate": "2024-12-02T18:18:37.747Z",
      "seriousness": "1"
    }
  ],
  "Comorbidity": [
    "Diabetes"
  ],
  "Test": [
    {
      "Date_time": "2024-12-02T18:18:37.749Z",
```

```
      "Respiratory_rate": 999.99,
      "SPO2": 100.99,
      "PCR_ct_value": 100.99,
      "PCR_result": true,
      "QT_ct_value": 999.99,
      "QT_result": true
    }
  ]
}
```

- Query:
  - "INSERT INTO patient (PID, fullname, PHONE, GENDER, ADDRESS, RISK_LEVEL) VALUES (%s, %s, %s, %s, %s, %s)"
  - "INSERT INTO symptoms (PNUM, SYMP_NAME, START_DATE, END_DATE, SERIOUS_LEVEL) VALUES (%s, %s, %s, %s, %s)"
  - "INSERT INTO patient_has_comorbidity (PNUM, COMORBIDITY_NAME) VALUES (%s, %s))

4. **List details of all testing which belong to a patient**

   (a) **Front end**:

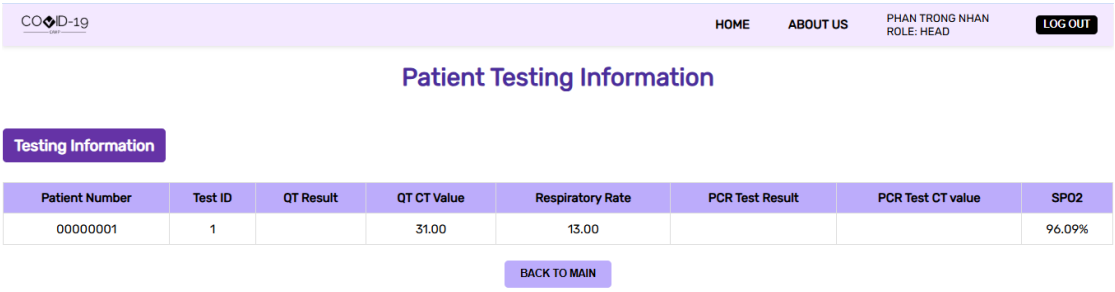


Figure 5: Testing result of a patient UI

   (b) **Back end**: **GET Test By Pnumber**

- End point: '/patient/test/{pnum}'
- Query: "SELECT * FROM test_result WHERE PNUMBER = {pnum}"
- Return:

```
[
  {
    "TEST_ID": 0,
    "PNUMBER": "stringst",
    "DATE_TIME": "2024-12-02T18:33:57.560Z",
    "RESPIRATORY_RATE": "string",
    "SPO2": "string",
    "PCR_ct_value": "string",
    "PCR_result": true,
    "QT_ct_value": "string",
    "QT_result": true
  }
]
```

5. **Make a report that provides full information about the patient including demographic information, comorbidities, symptoms, testing, and treatment**

   (a) **Front end**:

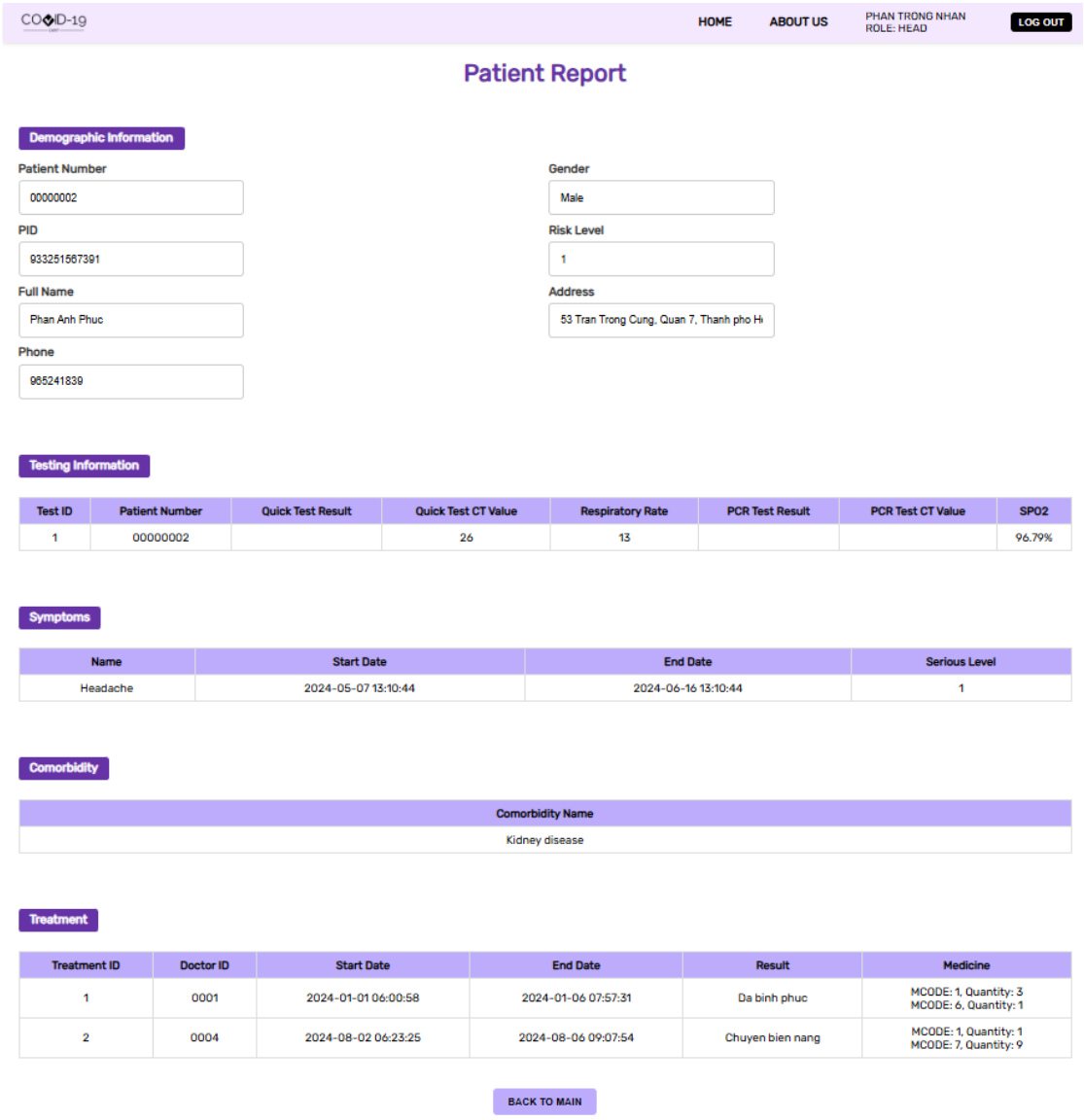


Figure 6: Patient report UI

   (b) **Back end**: **GET Patient Report**

   - End point: '`/patient/report/{pnum}`'
   - Query:
     - `"SELECT * FROM patient WHERE PNUMBER = {pnum}"` to get all basic information of a patient based on `pnumber`

  - `"SELECT * FROM test_result WHERE PNUMBER = {pnum}"` to get all testing result of a patient based on **pnumber**
  - `"SELECT * FROM symptoms WHERE PNUM = {pnum}"` to get symptoms of a patient based on **pnumber**
  - `"SELECT * FROM patient_has_comorbidity WHERE PNUM = {pnum}"` to get all comorbidities of a patient based on **pnumber**
  - `"Select t.PNUM, t.TREAT_ID, m.DOCTOR_ID, t.START_DATE, t.END_DATE, t.RESULT, h.MCODE, h.QUANTITY FROM treatment_record t JOIN make_treatrecord m ON t.pnum = {pnum} AND t.pnum = m.pnum AND t.treat_id = m.treat_id JOIN has_medicine h ON t.pnum = h.pnum AND t.treat_id = h.treat_id"` to get all information related to the treatments of a patient

- Return:


```
{
  "patient_info": [
    {
      "PNUMBER": "00000001",
      "PID": "223005401501",
      "fullname": "Pham Minh Lien",
      "PHONE": "126855092",
      "GENDER": "F",
      "address": "217 To Hieu, Quan Tan Phu, Thanh pho Ho Chi Minh",
      "RISK_LEVEL": "1"
    }
  ],
  "test_results": [
    {
      "TEST_ID": 1,
      "PNUMBER": "00000001",
      "DATE_TIME": "2024-01-12T13:43:48",
      "RESPIRATORY_RATE": 13,
      "SPO2": 96.09,
      "QT_ct_value": 31,
      "QT_result": true,
      "PCR_ct_value": null,
      "PCR_result": null
    }
  ],
  "symptoms": [
    {
      "PNUM": "00000001",
      "SYMP_NAME": "Congestion or runny nose",
      "START_DATE": "2023-12-23T13:43:48",
      "END_DATE": "2024-02-01T13:43:48",
      "SERIOUS_LEVEL": 1
    }
  ],
  "comorbidities": [
    {
      "PNUM": "00000001",
      "COMORBIDITY_NAME": "Diabetes"
    }
  ],
  "treatment_records": [
    {
      "PNUM": "00000001",
```

```
      "TREATMENT": [
        {
          "TREAT_ID": 1,
          "DOCTOR_ID": [
            "0002",
            "0004"
          ],
          "START_DATE": "2024-01-01T05:02:00",
          "END_DATE": "2024-01-06T07:29:34",
          "RESULT": "Nguy cap",
          "MEDICINE": [
            {
              "MCODE": 5,
              "QUANTITY": 5
            },
            {
              "MCODE": 6,
              "QUANTITY": 5
            },
            {
              "MCODE": 5,
              "QUANTITY": 5
            },
            {
              "MCODE": 6,
              "QUANTITY": 5
            }
          ]
        },
        {
          "TREAT_ID": 2,
          "DOCTOR_ID": [
            "0002"
          ],
          "START_DATE": "2024-09-10T04:54:44",
          "END_DATE": "2024-09-15T05:58:14",
          "RESULT": "Nguy cap",
          "MEDICINE": [
            {
              "MCODE": 5,
              "QUANTITY": 5
            },
            {
              "MCODE": 6,
              "QUANTITY": 6
            }
          ]
        }
      ]
    }
  ]
}
```

4 Indexing: Proving one use-case of indexing efficiency

4.1 Explain code

We generate data for task indexing for a quarantine camp, including details about patients, their test results, and their treatment records. The data we randomize are theses tables: patient, test result, treatment records, symptoms and is_assigned_to.

Each time we execute, we generate 10,000 patients. Each patient has 5 test results and 5 treatment records. The process is run 10 times, and each patient is assigned to a department and provided with symptoms.
To demonstrate the time efficiency of indexing in MySQL, we execute this query both with and without using an index:

```
create index idx_1 on symptoms(SYmp_name);
drop index idx_1 on symptoms;

explain select * from symptoms
where symp_name = 'Headache';
```

Listing 6: Sort the nurse in decreasing number of patients

Figure 7: Indexing Example

In this case, the symptoms table contains 815418 rows. When querying to find the symptom name 'headache', the execution time without indexing is 0.016 seconds, while the execution time with indexing is 0.000 seconds. This result demonstrates the efficiency of indexing in MySQL, as it significantly reduces the number of rows that need to be accessed during the query.

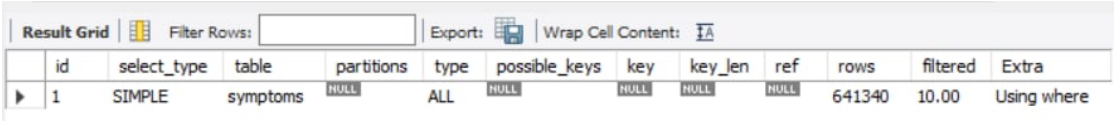
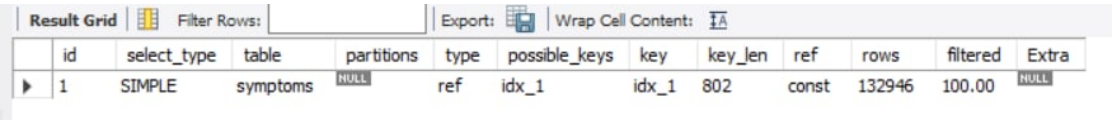


Figure 8: Result of without using Index

The image shows the result for a query on the `symptoms` table without using an index.
Index Usage: No index is utilized in this query, resulting in a full table scan (`type = ALL`).
Row Examination: MySQL estimates that 641,340 rows are scanned during query execution.
Filter: Only 10% of the scanned rows satisfy the query condition, as indicated by the `filtered` value. **Query Type:** This is a simple query with no partitions or additional processing. **Extra:** The `Using where` clause shows that filtering occurs after scanning all rows, further increasing query cost.



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	symptoms	NULL	ref	idx_1	idx_1	802	const	132946	100.00	NULL

Figure 9: Result of Using Index

The image shows the result for a query on the `symptoms` table where an index (`idx_1`) is being utilized.

Index Usage: The query uses the `idx_1` index, which optimizes data retrieval.
Key Efficiency: The index length (`key_len`) is 802 bytes, indicating efficient indexing for the query condition.
Row Examination: MySQL estimates that 132,946 rows will be scanned using the index.
Filter: All rows scanned by the index match the condition, as indicated by the `filtered` value of 100%.
Query Type: It's a simple, direct query with no partitions or extra processing.

5 Database Security

5.1 Login security

5.1.1 Password encryption during registration

- **Purpose:** Safely store user passwords in the database in a non-reversible format.
- **Implementation:**

1. Hash the password:
   Using `bcrypt.gensalt()` function from `bcrypt` library to generate a unique salt for the password received in the `insert_user` function. This will return a unique string.

```
def hash_password(password):
    hashed = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
    return hashed.decode('utf-8')
```

Listing 7: hash the password and return the hashed string

2. The hashed password (along with the username and role) is stored in the `USERNAME` table.

```
    hashed_password = hash_password(password)

    # SQL Insert Query
    query = """
    INSERT INTO USERNAME ( Username, Password, Authorities )
    VALUES ( %s, %s, %s)
    """
    data = ( username, hashed_password, role)

    # Execute and commit
    cursor.execute(query, data)
    connection.commit()
```

The one-way hashing will make the encoded password computationally infeasible to reverse. It ensures that even if the database is compromised, the actual passwords are not exposed.

5.1.2 Password verification during login

In this section, we first retrieve the *Password* and *Authorities* corresponding to the input `username` in `USERNAME` table. Then verify the password using `bcrypt.checkpw()` function to compare the password and hashed password.

```
# Query for hashed password
query = "SELECT Password, Authorities FROM USERNAME WHERE Username = %s"
cursor.execute(query, (username,))
result = cursor.fetchone()

if result:
    hashed_password = result[0]
    if bcrypt.checkpw(password.encode("utf-8"), hashed_password.encode("utf-8")):
    #Establish the Connection with corresponding role if password is matched
```

5.2 Discretionary Access Control (DAC)

Scenario:
In this quarantine camp database system, user with different roles such as: doctor, admitter, nurse, manager,... requires varying levels of access to the database:

- Doctors can view patient details and update treatment, but cannot change admission and discharge data.
- Admitter is allowed to access and manage patient admission and discharge records.

Discretionary Access Control (DAC) is a security model that allows the Database Administrator to control access permissions for their data. This approach was implemented in our database to ensure the security by granting and revoking only necessary privileges for the aforementioned roles.

Implementation
First, we create two users in the database: `doctor1` and `admitter` corresponding to the doctor and admitter roles. Then granting them appropriate privileges:

1. **Admitter**:
    - Insert: table `test_result`, `is_assigned_to`, `patient_has_comorbidity`, `symptoms`
    - Update: table `test_result`, `room`, `building`, `symptoms`
    - Read only: table `employee`, `comorbidity`
    - Insert and Read: table `discharge_date`, `admit_date`, `patient`,`patient_has_comorbidity`
    - Delete: table `discharge_date`, `admit_date`, `is_assigned_to`, `symptoms`

    With those privileges, user `admitter1` can access and insert a new patient in the `addpatient` module:

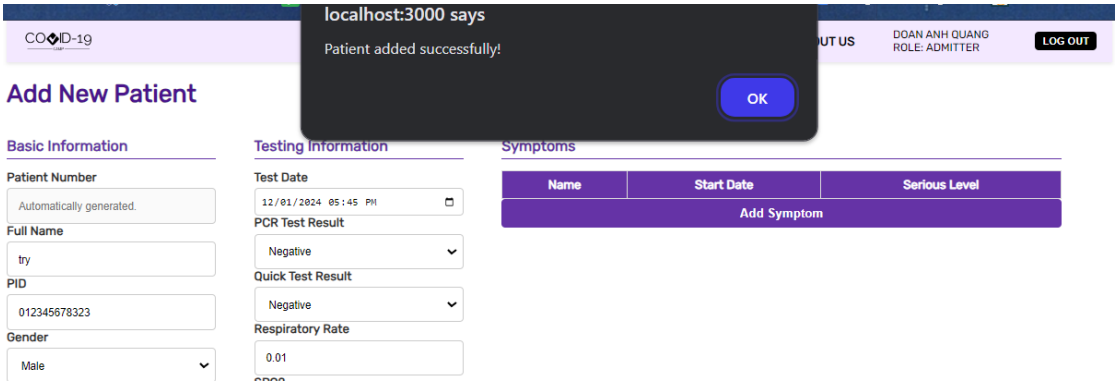


Figure 10: Allowed adding new patient

Since this role is not allowed to `SELECT` some table involving in patient report, the access to a patient report will be denied by the database:

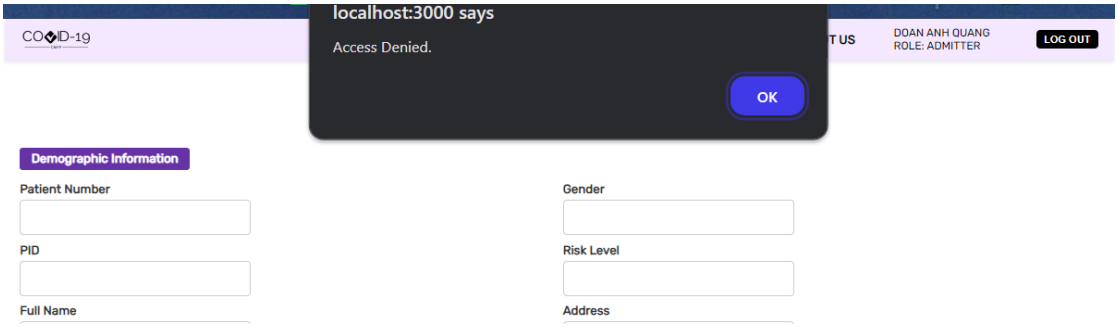


Figure 11: Denied accessing patient report

2. **Doctor**:
    - Insert: table `treatment_record`,`make_treatmentrecord`, `has_medicine`, `patient_has_comorbidity`, `symptoms`
    - Update: table `patient_has_comorbidity`, `has_medicine`, `patient`, `symptoms`, `treatment_record`, `make_treatrecord`
    - Read only: table `room`, `building`, `test_result`, `is_assigned_to`, `medicine`
    - Insert and Read: table `comorbidity`
    - Delete: table `patient_has_comorbidity`, `has_medicine`, `symptoms`

    With those privileges, user `doctor1` can not access and insert a new patient in the `addpatient` module:

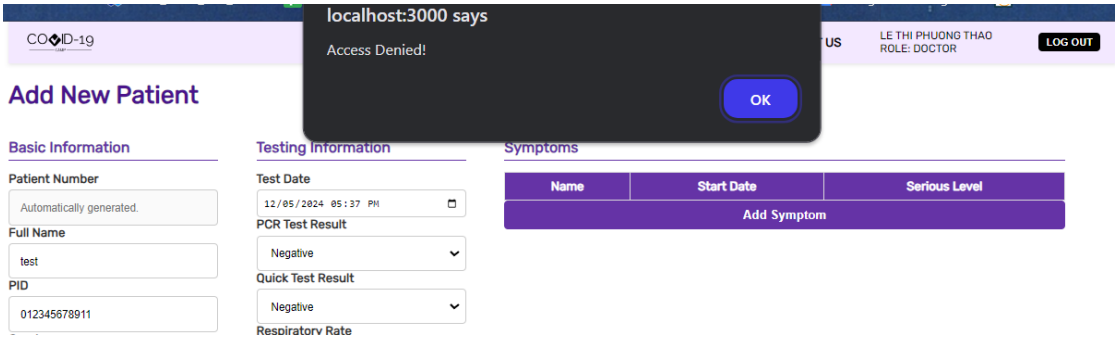


Figure 12: Denied adding new patient

But can view the patient report:

Figure 13: Allowed accessing patient report

By implementing DAC, Doctors were limited to viewing and updating admission-related data, preventing unauthorized access to patient admissions. Admitters managed admission data without access to treatment records. Security and privacy were maintained through granular control over permissions.