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

DATABASE SYSTEMS

Assignment

QUARANTINE CAMP DATABASE

Advisor: Phan Trong Nhan
Class: CC02
Group: 6
Students: Pham Huynh Quoc Thanh - 2152289
Nguyen Phuoc Thinh - 2153838
Banh Tan Thuan - 2153011
Le Van Tien - 2153886
Ly Tran Phuoc Tri - 2153920

HO CHI MINH CITY, DECEMBER 2023

Contents

1 Introduction **3**

2 Physical database design **3**
2.1 Implement the database . . . 3
2.2 Insert data . . . 11
2.3 Deal with constraints . . . 11

3 Stored procedure - Function - SQL **12**
3.1 Update PCR test to positive . . . 12
3.1.1 Code . . . 12
3.1.2 Before execution . . . 13
3.1.3 After execution . . . 13
3.2 Search all patient's information by name . . . 13
3.2.1 Code . . . 13
3.2.2 Execution . . . 14
3.3 Calculate the testing for patients . . . 14
3.3.1 Code . . . 14
3.3.2 Execution . . . 15
3.4 Sort list of nurses with conditions . . . 15
3.4.1 Code . . . 15
3.4.2 Execution . . . 16

4 Building application **16**
4.1 Technologies . . . 16
4.2 User account . . . 17
4.3 Requirement functions . . . 17
4.3.1 Log in - Log out . . . 17
4.3.2 Search patient's information . . . 19
4.3.3 Add new patient's information . . . 21
4.3.4 List of patient's testing . . . 23
4.3.5 Patient report . . . 25

5 Database management **29**
5.1 Indexing efficiency . . . 29
5.1.1 Time efficiency . . . 31
5.1.2 Execution plan . . . 32
5.2 Database security - SQL Injection . . . 32

6 Source code **36**

Member list & Workload

No.	Fullname	Student ID	Percentage of work
1	Pham Huynh Quoc Thanh	2152289	100%
2	Nguyen Phuoc Thinh	2153838	100%
3	Banh Tan Thuan	2153011	100%
4	Le Van Tien	2153886	100%
5	Ly Tran Phuoc Tri	2153920	100%

1 Introduction

To prevent the spread of Covid-19, which has become increasingly complex, Vietnam implemented many strict measures. Anyone who has been in close contact with an infected person or who has traveled from an affected area will be required to self-isolate or stay in a quarantine camp run by the military and medical personnel. In assignment 1, we based on the basic information describing the operation of a quarantine camp as well as the management of its patients to analyze and propose a suitable model for this database.

In this assignment 2, we implement and query the database from our assignment 1. We will sequentially perform the tasks that have been requested. First is the design and implementation of the database at the physical level using a Database Management System (DBMS). Next, we will apply the knowledge we have learned to perform operations on the database.

In order to have a clearer view, we will also build a web application. This application will be connected to the database that we have created. Finally, we will discuss two important issues, indexing efficiency and database security, that we have implemented on the database and our application.

2 Physical database design

2.1 Implement the database

Overall, our database consists of 21 tables, as designed in the relational database schema we designed in assignment 1. Here are the code in SQL to create the tables for our database. The data types and data length that we have set up are also mentioned below. About the constraints, we will discuss them in more detail in section 2.3.

- **Patient Table:**
  - `patient_id: CHAR(10)` - Unique identifier for patients. It is fix string of 10 characters with prefix P and 9 following numbers.
  - `identity_number: CHAR(12)` - Unique identification number for each patient, which is based on real-life 12 digit numbers like a citizen identification number.
  - `patient_full_name: CHAR(30)` - Full name of the patient.
  - `phone: CHAR(10)` - Phone number of the patient, which is based on real-life 10 phone digit number.
  - `gender: CHAR(5)` - Gender of the patient.
  - `address: CHAR(100)` - Address of the patient.
  - `warning: CHAR(1)` - A flag indicating a warning, with a CHECK constraint ensuring it's either 'Y' or 'N'.

```
CREATE TABLE Patient(
    patient_id          CHAR(10)    NOT NULL    PRIMARY KEY,
    identity_number     CHAR(12)    UNIQUE      NOT NULL,
    patient_full_name   CHAR(30)    NOT NULL,
    phone               CHAR(10)    NOT NULL,
    gender              CHAR(5)     NOT NULL,
    address             CHAR(100)   NOT NULL,
    warning             CHAR(1)     NULL    CHECK (warning IN ('Y', 'N'))
);

```

- **Symptom Table:**
  - `symptom_name`: `CHAR(100)` - Name of the symptom.
  - `start_date`: `DATE` - Start date of the symptom.
  - `end_date`: `DATE` - End date of the symptom (nullable).
  - `is_serious`: `CHAR(1)` - Flag indicating seriousness, with a CHECK constraint ('Y' or 'N').
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.
  - Composite primary key on `start_date`, `symptom_name`, and `patient_id`.

```
CREATE TABLE Symptom (
    symptom_name    CHAR(100)   NOT NULL,
    start_date      DATE        NOT NULL,
    end_date        DATE        NULL,
    is_serious CHAR(1) NOT NULL CHECK (is_serious IN ('Y', 'N')),
    patient_id REFERENCES Patient(patient_id)   ON DELETE CASCADE   NOT
NULL,
    CONSTRAINT symptom_key PRIMARY KEY(start_date, symptom_name, patient_
id)
);

```

- **DischargeDate Table:**
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.
  - `discharge_date`: `DATE` - Date of patient discharge.
  - Composite primary key on `patient_id` and `discharge_date`.

```
CREATE TABLE DischargeDate(
    patient_id REFERENCES Patient(patient_id)   ON DELETE CASCADE   NOT
NULL,
    discharge_date  DATE    NOT NULL,
    CONSTRAINT discharge_date_key PRIMARY KEY(patient_id, discharge_date)
);

```

- **Test Table:**
  - `test_id`: `CHAR(10)` - Unique identifier for tests. It is fix string of 10 characters with prefix T and 9 following numbers.
  - `datetime`: `TIMESTAMP` - Date and time of the test.
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.
  - Composite primary key on `test_id`, `datetime`, and `patient_id`.

```
CREATE TABLE Test(
    test_id     CHAR(10)    NOT NULL    PRIMARY KEY,
    datetime    TIMESTAMP   NOT NULL,
    patient_id REFERENCES Patient(patient_id)   ON DELETE CASCADE   NOT
NULL
);

```

- **Room Table:**
  - `building`: `CHAR(10)` - Name of the building.
  - `floor`: `INT` - Floor number.
  - `room_number`: `CHAR(10)` - Room number.
  - `capacity`: `INT` - Maximum capacity of the room.
  - `room_type`: `CHAR(20)` - Type of the room.
  - Composite primary key on `building`, `floor`, and `room_number`.

```
CREATE TABLE Room(
    building            CHAR(10)     NOT NULL,
    floor               INT          NOT NULL,
    room_number         CHAR(10)     NOT NULL,
    capacity            INT          NOT NULL,
    room_type           CHAR(20)     NOT NULL,
    CONSTRAINT room_key PRIMARY KEY(building, floor, room_number)
);

```

- **Comorbidity Table:**
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.
  - `comorbidities`: `CHAR(50)` - Description of comorbidities.
  - Composite primary key on `patient_id` and `comorbidities`.

```
CREATE TABLE Comorbidity(
    patient_id  REFERENCES Patient(patient_id)  ON DELETE CASCADE   NOT
NULL,
    comorbidities  CHAR(50)                          NOT NULL,
    CONSTRAINT comorbidity_key PRIMARY KEY(patient_id, comorbidities)
);

```

- **RespiratoryRate_Test Table:**
  - `test_id`: `CHAR(10)` - Foreign key referencing the Test table.
  - `respiratory_result`: `FLOAT` - Result of respiratory rate test.
  - Primary key on `test_id`.

```
CREATE TABLE RespiratoryRate_Test(
    test_id REFERENCES Test(test_id)    ON DELETE CASCADE    NOT NULL
PRIMARY KEY,
    respiratory_result  FLOAT   NOT NULL
);

```

- **SPO2_Test Table:**
  - `test_id`: `CHAR(10)` - Foreign key referencing the Test table.
  - `SPO2_result`: `FLOAT` - Result of SPO2 test.
  - Primary key on `test_id`.

```
CREATE TABLE SPO2_Test(
    test_id REFERENCES Test(test_id)    ON DELETE CASCADE    NOT NULL
PRIMARY KEY,
    SPO2_result     FLOAT   NOT NULL
);

```

- **Quick_Test Table:**
  - `test_id:` `CHAR(10)` - Foreign key referencing the Test table.
  - `quick_test_result:` `CHAR(10)` - Result of the quick test.
  - `cycle_threshold_value:` `FLOAT (nullable)` - Value associated with positive results.
  - CHECK constraint ensuring if the result is positive, the value must not be null.

```
CREATE TABLE Quick_Test(
    test_id REFERENCES Test(test_id)    ON DELETE CASCADE    NOT NULL
PRIMARY KEY,
    quick_test_result       char(10)        NOT NULL,
    cycle_threshold_value   FLOAT   NULL,
    --If quick_test_result is positive, the cycle_threshold_value must
not be null
    CONSTRAINT positive_result_requires CHECK (
        (quick_test_result = 'Positive' AND cycle_threshold_value IS NOT
NULL) OR
        (quick_test_result != 'Positive')
    )
);

```

- **PCR_Test Table:**
  - `test_id:` `CHAR(10)` - Foreign key referencing the Test table.
  - `PCR_result:` `CHAR(10)` - Result of the PCR test.
  - `cycle_threshold_value:` `FLOAT (nullable)` - Value associated with positive results.
  - CHECK constraint ensuring if the result is positive, the value must not be null.

```
CREATE TABLE PCR_Test(
    test_id REFERENCES Test(test_id)    ON DELETE CASCADE    NOT NULL
PRIMARY KEY,
    PCR_result  char(10)        NOT NULL,
    cycle_threshold_value   FLOAT   NULL,
    --If pcr_result is positive, the cycle_threshold_value must not be
null
    CONSTRAINT positive_result_requires_value CHECK (
        (PCR_result = 'Positive' AND cycle_threshold_value IS NOT NULL)
OR
        (PCR_result != 'Positive')
    )
);

```

- **People Table:**
  - `person_id`: `CHAR(10)` - Unique identifier for people. It is fix string of 10 characters with prefix E and 9 following numbers.
  - `first_name`: `CHAR(20)` - First name of the person.
  - `last_name`: `CHAR(20)` - Last name of the person.
  - `date_of_birth`: `DATE` - Birthdate of the person.
  - `gender`: `CHAR(5)` - Gender of the person.
  - `address`: `CHAR(100)` - Address of the person.
  - `start_date_of_work`: `DATE` - Start date of work.
  - Flags (`volunteer_flag`, `staff_flag`, `nurse_flag`, `doctor_flag`, `manager_flag`) indicating roles.

```
CREATE TABLE People(
    person_id           CHAR(10)        NOT NULL    PRIMARY KEY,
    first_name          CHAR(20)        NOT NULL,
    last_name           CHAR(20)        NOT NULL,
    date_of_birth       DATE            NOT NULL,
    gender              CHAR(5)         NOT NULL,
    address             CHAR(100)       NOT NULL,
    start_date_of_work  DATE            NOT NULL,
    volunteer_flag      char(1)         NULL,
    staff_flag          char(1)         NULL,
    nurse_flag          char(1)         NULL,
    doctor_flag         char(1)         NULL,
    manager_flag        char(1)         NULL
);

```

- **HeadOfCamp Table:**
  - `head_of_camp_id`: `CHAR(10)` - Foreign key referencing the People table.
  - Additional attributes such as `first_name`, `last_name`, `date_of_birth`, `gender`, `address`, `start_date_of_work`.

```
CREATE TABLE HeadOfCamp(
    head_of_camp_id REFERENCES People(person_id)    ON DELETE CASCADE
NOT NULL    PRIMARY KEY,
    first_name          CHAR(20)        NOT NULL,
    last_name           CHAR(20)        NOT NULL,
    date_of_birth       DATE            NOT NULL,
    gender              CHAR(5)         NOT NULL,
    address             CHAR(100)       NOT NULL,
    start_date_of_work  DATE            NOT NULL
);

CREATE UNIQUE INDEX head ON headofcamp(1);


```

- **Admission Table:**
  - `admission_id`: `CHAR(10)` - Unique identifier for admissions. It is fix string of 10 characters with prefix A and 9 following numbers.

- `admission_date`: `DATE` - Date of admission.
- `from_where`: `CHAR(100)` - Source of admission.
- `staff_id`: `CHAR(10)` - Foreign key referencing the People table (staff involved in admission).
- `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.

```
1  CREATE TABLE Admission(
2      admission_id   CHAR(10)    NOT NULL    PRIMARY KEY,
3      admission_date DATE        NOT NULL,
4      from_where      CHAR(100)   NOT NULL,
5      staff_id    REFERENCES People(person_id)    ON DELETE CASCADE   NOT
   NULL,
6      patient_id  REFERENCES Patient(patient_id)  ON DELETE CASCADE   NOT
   NULL
7  );
8
```

- **PhoneNumber Table:**
  - `person_id`: `CHAR(10)` - Foreign key referencing the People table.
  - `phone_number`: `CHAR(10)` - Phone number associated with the person.
  - Composite primary key on `person_id` and `phone_number`.

```
1  CREATE TABLE PhoneNumber(
2      person_id REFERENCES People(person_id)  ON DELETE CASCADE   NOT NULL,
3      phone_number    CHAR(10)    NOT NULL,
4      CONSTRAINT  phone_number_key PRIMARY KEY(person_id, phone_number)
5  );
6
```

- **Medication Table:**
  - `unique_code`: `CHAR(10)` - Unique identifier for medications. It is fix string of 10 characters with prefix M and 9 following numbers.
  - `medication_name`: `CHAR(50)` - Name of the medication.
  - `effects`: `CHAR(100)` - Effects of the medication.
  - `price`: `FLOAT` - Price of the medication.
  - `expiration_date`: `DATE` - Expiration date of the medication.
  - Primary key on `unique_code`.

```
1  CREATE TABLE Medication(
2      unique_code         CHAR(10)    NOT NULL    PRIMARY KEY,
3      medication_name     CHAR(50)    NOT NULL,
4      effects             CHAR(100)   NOT NULL,
5      price               FLOAT       NOT NULL,
6      expriration_date    DATE        NOT NULL
7  );
8
```

- **BelongTO Table:**
  - `test_id`: `CHAR(10)` - Foreign key referencing the Test table.
  - `admission_id`: `CHAR(10)` - Foreign key referencing the Admission table.
  - Composite primary key on `test_id` and `admission_id`.

```
CREATE TABLE BelongTO(
    test_id REFERENCES Test(test_id)    ON DELETE CASCADE   NOT NULL ,
    admission_id REFERENCES Admission(admission_id),
    CONSTRAINT belong_to_key PRIMARY KEY(test_id, admission_id)
);

```

- **Treatment Table:**
  - `treatment_id`: `CHAR(10)` - Unique identifier for treatments. It is fix string of 10 characters with prefix T and 9 following numbers.
  - `initiation_date`: `DATE` - Date of treatment initiation.
  - `completion_date`: `DATE (nullable)` - Date of treatment completion.
  - `overall_result`: `CHAR(100) (nullable)` - Overall result of the treatment.
  - Primary key on `treatment_id`.

```
CREATE TABLE Treatment(
    treatment_id        CHAR(10)        NOT NULL    PRIMARY KEY,
    initiation_date     DATE            NOT NULL,
    completion_date     DATE             NULL,
    overall_result              CHAR(100)          NULL
);

```

- **Treat Table:**
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.
  - `doctor_id`: `CHAR(10)` - Foreign key referencing the People table.
  - `treatment_id`: `CHAR(10)` - Foreign key referencing the Treatment table.
  - Composite primary key on `patient_id`, `doctor_id`, and `treatment_id`.

```
CREATE TABLE Treat(
    patient_id      REFERENCES Patient(patient_id)      ON DELETE CASCADE
   NOT NULL,
    doctor_id       REFERENCES People(person_id)        ON DELETE CASCADE
   NOT NULL,
    treatment_id    REFERENCES Treatment(treatment_id)  ON DELETE CASCADE
   NOT NULL,
    CONSTRAINT treat_key PRIMARY KEY(patient_id, doctor_id, treatment_id)
);

```

- **TakeCare Table:**
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.

- `nurse_id`: `CHAR(10)` - Foreign key referencing the People table.
- `start_date`: `DATE` - Start date of taking care.
- Composite primary key on `patient_id`, `nurse_id`, and `start_date`.

```
    CREATE TABLE TakeCare(
        patient_id  REFERENCES Patient(patient_id)  ON DELETE CASCADE   NOT
    NULL,
        nurse_id    REFERENCES People(person_id)    ON DELETE CASCADE   NOT
    NULL,
        start_date  DATE    NOT NULL,
        CONSTRAINT take_care_key PRIMARY KEY(patient_id, nurse_id, start_date
    )
    );

```

- **Use Table:**
  - `unique_code`: `CHAR(10)` - Foreign key referencing the Medication table.
  - `treatment_id`: `CHAR(10)` - Foreign key referencing the Treatment table.
  - Composite primary key on `unique_code` and `treatment_id`.

```
    CREATE TABLE Use (
    unique_code          CHAR(10)    NOT NULL,
    treatment_id         CHAR(10)    NOT NULL,
    amount               NUMBER, -- Add the new column
    CONSTRAINT use_key PRIMARY KEY(unique_code, treatment_id),
    FOREIGN KEY (unique_code) REFERENCES Medication(unique_code) ON DELETE
    CASCADE,
    FOREIGN KEY (treatment_id) REFERENCES Treatment(treatment_id) ON DELETE
    CASCADE
);

```

- **LocationHistory Table:**
  - `building`: `CHAR(10)` - Building name.
  - `floor`: `INT` - Floor number.
  - `room_number`: `CHAR(10)` - Room number.
  - `patient_id`: `CHAR(10)` - Foreign key referencing the Patient table.
  - `checkin_datetime`: `TIMESTAMP` - Check-in date and time.
  - `check_out_datetime`: `TIMESTAMP (nullable)` - Check-out date and time.
  - Composite primary key on `building`, `floor`, `room_number`, `patient_id`, and `checkin_datetime`.
  - Foreign key constraint referencing the Room table.

```
    CREATE TABLE LocationHistory (
        building            CHAR(10)    NOT NULL,
        floor               INT         NOT NULL,
        room_number         CHAR(10)    NOT NULL,
        patient_id          REFERENCES  Patient(patient_id)     ON DELETE
    CASCADE   NOT NULL,
        checkin_datetime    TIMESTAMP   NOT NULL,
        check_out_datetime  TIMESTAMP,
```

```
8         CONSTRAINT location_history_key PRIMARY KEY(building, floor, room_
      number, patient_id, checkin_datetime),
9         CONSTRAINT fk_location_history_room FOREIGN KEY (building, floor,
      room_number) REFERENCES Room(building, floor, room_number)    ON DELETE
      CASCADE
10    );
11
```

2.2 Insert data

To ensure realism, the database we created contains information on 100 patients in the Ho Chi Minh city area and neighboring provinces. In addition, the database also includes 1100 test values for these patients as well as a lot of other related information. Since it is a quarantine camp database, it also includes information on more than 70 doctors, nurses, and staff of the camp. The data was collected starting from mid-August 2020.

Therefore, the total of nearly 9000 lines of SQL for inserting data. Although we generated these information ourselves, we still ensure that they are meaningful, relevant, and close to reality. We want to emphasize this because in section 5.1 Indexing efficiency, we will add a large amount of dummy data to the database for the main purpose of demonstrating the effectiveness of the indexing method.

2.3 Deal with constraints

- Constraint 1: One doctor will be designated as the head of the camp

```
1 CREATE UNIQUE INDEX head ON headofcamp(1); --Only one row constaint
2
```

We create a unique index on head of camp to be 1. It will not allow the second row to be exist.

- Constraint 2: PCR test: the result is true (positive) or false (negative). In case it is positive, the camp wants to track the corresponding cycle threshold (ct) value.

```
1  CREATE TABLE PCR_Test(
2          test_id REFERENCES Test(test_id)    ON DELETE CASCADE    NOT NULL
       PRIMARY KEY,
3          PCR_result  char(10)        NOT NULL,
4          cycle_threshold_value   FLOAT   NULL,
5          CONSTRAINT positive_result_requires_value CHECK (
6          (PCR_result = 'Positive' AND cycle_threshold_value IS NOT NULL) OR
7          (PCR_result != 'Positive')
8      )
9  );
10
```

We add the constraint positive_result_requires_value to make sure that if the patient test is positive, the patient must have cycle threshold value.

- Constraint 3: Quick test: the result is true (positive) or false (negative). In case it is positive, the camp wants to track the corresponding cycle threshold (ct) value.

```
1  CREATE TABLE Quick_Test(
2      test_id REFERENCES Test(test_id)    ON DELETE CASCADE    NOT NULL
       PRIMARY KEY,
```

```
    quick_test_result       char(10)        NOT NULL,
    cycle_threshold_value   FLOAT   NULL,
    CONSTRAINT positive_result_requires CHECK (
        (quick_test_result = 'Positive' AND cycle_threshold_value IS NOT NULL
    ) OR
        (quick_test_result != 'Positive')
    )
);

```

We add the constraint positive_result_requires to make sure that if the patient test is positive, the patient must have cycle threshold value.

- Constraint 4: A patient may have many testing during his or her stay. If the SPO2 is smaller than 96% and the respiratory rate is larger than 20 breaths per minute, the patient is marked “warning”.

```
CREATE OR REPLACE PROCEDURE warning(
    start_time TIMESTAMP,
    end_time TIMESTAMP
)IS
BEGIN
    UPDATE Patient
    SET warning = 'Y'
    WHERE Patient.patient_id IN(
    SELECT t.patient_id
    FROM test t
    LEFT JOIN SPO2_Test sp ON sp.test_id = t.test_id
    LEFT JOIN Respiratoryrate_Test r ON r.test_id = t.test_id
    WHERE t.datetime <= end_time AND t.datetime >= start_time
    );
END warning;
/

EXEC warning(to_timestamp('17/10/2020 00:00:00', 'DD/MM/YYYY HH24:MI:SS'), to
    _timestamp('17/10/2020 09:54:54', 'DD/MM/YYYY HH24:MI:SS'));

```

We create a procedure to update the patient status if their SPO2 is smaller than 96% and the respiratory rate is larger than 20 breaths per minute in a period of specific time to be marked as ”warning”.

3 Stored procedure - Function - SQL

3.1 Update PCR test to positive

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

3.1.1 Code

```
--a. Update patient PCR test to positive with null cycle threshold value for all
--patients whose admission date is from 01/09/2020.
UPDATE PCR_Test
SET PCR_Result = 'Positive', cycle_threshold_value = NULL
WHERE PCR_Test.test_id IN(
```

```sql
SELECT Test.test_id
FROM Test, BelongTo, Admission, PCR_test
WHERE BelongTo.test_id = Test.test_id
AND BelongTo.admission_id = Admission.admission_id
AND Test.test_id = PCR_Test.test_id
AND Admission.admission_date >= TO_DATE('01/09/2020', 'DD/MM/YYYY')
);

--Test was PCR_Test updated
SELECT * FROM PCR_Test WHERE cycle_threshold_value IS NULL;
```

3.1.2 Before execution

HONE	GENDER	ADDRESS	WARNING
23066633	M	28 Long Vinh, Binh Hung Commune, Binh Chanh Dist, Ho Chi Minh City	(null)
96518749	F	42 Cao Lo, Ward 4, Dist 8, Ho Chi Minh City	(null)
42311156	F	11 Phan Van Chuong, Tan Phu Ward, Dist 7, Ho Chi Minh City	(null)
51182020	F	77 Thoai Ngoc Hau, Hoa Thanh Ward, Tan Phu Dist, Ho Chi Minh City	(null)
70009999	F	17 Thu Khoa Huan, Ward 8, Tan Binh Dist, Ho Chi Minh City	(null)
32002034	F	71 Nguyen Van Luong, Ward 10, Dist 6, Ho Chi Minh City	(null)
49135455	M	52 Pho Duc Chinh, Nguyen Thai Binh Ward, Dist 1, Ho Chi Minh City	(null)
39112544	F	32 Phan Van Bay, Hiep Phuoc Commune, Nha Be Dist, Ho Chi Minh City	(null)

Figure 1: Before executing warning procedure

3.1.3 After execution

	HONE	GENDER	ADDRESS	WARNING
40	23066633	M	28 Long Vinh, Binh Hung Commune, Binh Chanh Dist, Ho Chi Minh City	(null)
41	96518749	F	42 Cao Lo, Ward 4, Dist 8, Ho Chi Minh City	(null)
42	42311156	F	11 Phan Van Chuong, Tan Phu Ward, Dist 7, Ho Chi Minh City	(null)
43	51182020	F	77 Thoai Ngoc Hau, Hoa Thanh Ward, Tan Phu Dist, Ho Chi Minh City	(null)
44	70009999	F	17 Thu Khoa Huan, Ward 8, Tan Binh Dist, Ho Chi Minh City	Y
45	32002034	F	71 Nguyen Van Luong, Ward 10, Dist 6, Ho Chi Minh City	(null)
46	49135455	M	52 Pho Duc Chinh, Nguyen Thai Binh Ward, Dist 1, Ho Chi Minh City	(null)
47	39112544	F	32 Phan Van Bay, Hiep Phuoc Commune, Nha Be Dist, Ho Chi Minh City	(null)

Figure 2: After executing warning procedure

3.2 Search all patient's information by name

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

3.2.1 Code

```sql
--b. Select all the patient information whose name is Nguyen Van A
SELECT *
FROM Patient p
LEFT JOIN Comorbidity c ON c.patient_id = p.patient_id
WHERE p.patient_full_name = 'Nguyen Van A';
```

3.2.2 Execution

	PATIENT_ID	IDENTITY_NUMBER	PATIENT_FULL_NAME	PHONE	GENDER	ADDRESS	WARNING
1	P000000001	079081002023	Nguyen Van A	0392002023	M	286 Ly Thuong Kiet, Ward 14, Dist 10, Ho Chi Minh City	(null)
2	P000000025	075086001122	Nguyen Van A	0984646688	M	566 Lac Long Quan, Ward 10, Tan Binh Dist, Ho Chi Minh City	(null)
3	P000000040	038069001690	Nguyen Van A	0323066633	M	28 Long Vinh, Binh Hung Commune, Binh Chanh Dist, Ho Chi Minh City	(null)
4	P000000063	086068000446	Nguyen Van A	0827717189	M	82 Tue Tinh, Ward 13, Dist 11, Ho Chi Minh City	(null)

Figure 3: Example of search information by name

3.3 Calculate the testing for patients

Write a function to calculate the testing for each patient.
Input: Patient ID
Output: A list of testing

3.3.1 Code

```
SET SERVEROUTPUT ON;
DROP TYPE TestResultType force;
CREATE OR REPLACE TYPE TestResultType AS OBJECT (
    test_id CHAR(10),
    test_type CHAR(30),
    datetime TIMESTAMP,
    respiratori_result  FLOAT,
    spo2_result FLOAT,
    quick_test_result   CHAR(10),
    ct_quick_test   FLOAT,
    pcr_result      CHAR(10),
    ct_pcr_test     FLOAT
);
/

CREATE OR REPLACE TYPE TestResultTypeTable AS TABLE OF TestResultType;
/

CREATE OR REPLACE FUNCTION get_patient_testing(
    patient_id_input VARCHAR2
)   RETURN TestResultTypeTable AS
    test_list TestResultTypeTable := TestResultTypeTable();
BEGIN
    FOR row IN(
        SELECT t.test_id,
               CASE
                   WHEN r.test_id IS NOT NULL THEN 'Respiratory Rate Test'
                   WHEN s.test_id IS NOT NULL THEN 'SPO2 Test'
                   WHEN q.test_id IS NOT NULL THEN 'Quick Test'
                   WHEN p.test_id IS NOT NULL THEN 'PCR Test'
                   ELSE 'No Test Found'
               END AS test_type,
               t.datetime,
               r.respiratory_result,
               s.spo2_result,
               q.quick_test_result,
               q.cycle_threshold_value AS ct_quick_test,
               p.pcr_result,
               p.cycle_threshold_value AS ct_pcr_test
        FROM Test t
        LEFT JOIN RespiratoryRate_Test r ON t.test_id = r.test_id
        LEFT JOIN SPO2_Test s ON t.test_id = s.test_id
```

```
        LEFT JOIN Quick_Test q ON t.test_id = q.test_id
        LEFT JOIN PCR_Test p ON t.test_id = p.test_id
        WHERE t.patient_id = patient_id_input
        ORDER BY datetime DESC
    )LOOP
        test_list.extend();
        test_list(test_list.count) := TestResultType(row.test_id, row.test_type,
    row.datetime, row.respiratory_result, row.spo2_result, row.quick_test_result,
    row.ct_quick_test,
        row.pcr_result, row.ct_pcr_test);
    END LOOP;
    RETURN test_list;
END;
/

SELECT * FROM TABLE(get_patient_testing('P000000001'));
```

3.3.2 Execution

	TEST_ID	TEST_TYPE	DATETIME	RESPIRATORI_RESULT	SPO2_RESULT	QUICK_TEST_RESULT	CT_QUICK_TEST	PCR_RESULT	CT_PCR_TEST
1	T000000312	PCR Test	21-OCT-20 03.00.02.000000000 PM	(null)	(null)	(null)	(null)	Negative	35
2	T000000299	Quick Test	18-OCT-20 03.00.02.000000000 PM	(null)	(null)	Positive	29	(null)	(null)
3	T000000284	Quick Test	15-OCT-20 06.23.42.000000000 PM	(null)	(null)	Positive	28	(null)	(null)
4	T000000265	Respiratory Rate Test	12-OCT-20 07.01.27.000000000 AM	21	(null)	(null)	(null)	(null)	(null)
5	T000000264	SPO2 Test	12-OCT-20 07.01.22.000000000 AM	(null)	98	(null)	(null)	(null)	(null)
6	T000000263	Quick Test	12-OCT-20 07.00.06.000000000 AM	(null)	(null)	Negative	29	(null)	(null)
7	T000000262	PCR Test	12-OCT-20 07.00.05.000000000 AM	(null)	(null)	(null)	(null)	Negative	29
8	T000000198	Respiratory Rate Test	25-SEP-20 02.48.49.000000000 PM	21.5	(null)	(null)	(null)	(null)	(null)
9	T000000197	SPO2 Test	25-SEP-20 02.47.42.000000000 PM	(null)	97	(null)	(null)	(null)	(null)
10	T000000196	Quick Test	25-SEP-20 02.45.29.000000000 PM	(null)	(null)	Negative	29	(null)	(null)
11	T000000195	PCR Test	25-SEP-20 02.45.02.000000000 PM	(null)	(null)	(null)	(null)	Negative	29
12	T000000159	Respiratory Rate Test	16-SEP-20 01.33.51.000000000 PM	24	(null)	(null)	(null)	(null)	(null)
13	T000000158	SPO2 Test	16-SEP-20 01.32.44.000000000 PM	(null)	90	(null)	(null)	(null)	(null)
14	T000000157	Quick Test	16-SEP-20 01.31.29.000000000 PM	(null)	(null)	Positive	21.5	(null)	(null)
15	T000000156	PCR Test	16-SEP-20 01.31.05.000000000 PM	(null)	(null)	(null)	(null)	Positive	21.5
16	T000000009	Respiratory Rate Test	16-AUG-20 08.34.56.000000000 AM	21	(null)	(null)	(null)	(null)	(null)
17	T000000008	SPO2 Test	16-AUG-20 08.32.43.000000000 AM	(null)	95.5	(null)	(null)	(null)	(null)
18	T000000007	Quick Test	16-AUG-20 08.31.12.000000000 AM	(null)	(null)	Positive	25	(null)	(null)

Figure 4: Get patient P000000001's testing

3.4 Sort list of nurses with conditions

Write a procedure to sort the nurses in decreasing number of patients he/she takes care in a period of time.
Input: Start date, End date
Output: A list of sorting nurses

3.4.1 Code

```
CREATE OR REPLACE PROCEDURE SortNursesByPatientCount(
    start_date DATE,
    end_date DATE
) IS
BEGIN
    -- Populate the nested table with nurse information and patient count
    FOR row IN(
        SELECT
            People.person_id,
            People.first_name,
            People.last_name,
            COUNT(p.patient_id) AS patient_count
        FROM
```

```
14              TakeCare n
15              LEFT JOIN Patient p ON  n.start_date >= start_date
16              LEFT JOIN People ON n.nurse_id = People.person_id
17              WHERE n.start_date <= end_date AND n.patient_id = p.patient_id
18          GROUP BY
19              People.person_id, People.first_name, People.last_name
20          ORDER BY
21              COUNT(p.patient_id) DESC
22      )LOOP
23          DBMS_OUTPUT.PUT_LINE('Nurse ID: ' || row.person_id || ', Nurse FirstName:
        ' || row.first_name  ||', Nurse LastName: ' || row.last_name || ', Patient
        Count: ' || row.patient_count);
24      END LOOP;
25  END SortNursesByPatientCount;
26  /
27
28  EXEC SortNursesByPatientCount(TO_DATE('16/08/2020', 'DD/MM/YYYY'), TO_DATE('24/08/
        2020', 'DD/MM/YYYY'));
```

3.4.2 Execution

```
Procedure SORTNURSESBYPATIENTCOUNT compiled

Nurse ID: E000000006, Nurse FirstName: Chanh Cuong        , Nurse LastName: Nguyen            , Patient Count: 4
Nurse ID: E000000005, Nurse FirstName: Hoai An            , Nurse LastName: Nguyen            , Patient Count: 3
Nurse ID: E000000009, Nurse FirstName: Tran Khanh Thi     , Nurse LastName: Nguyen            , Patient Count: 2
Nurse ID: E000000007, Nurse FirstName: Thi Thanh Tam      , Nurse LastName: Pham              , Patient Count: 2
Nurse ID: E000000008, Nurse FirstName: Thi Thu Dung       , Nurse LastName: Nguyen            , Patient Count: 1


PL/SQL procedure successfully completed.
```

Figure 5: Execute sort nurse procedure

4 Building application

4.1 Technologies

Understanding this issue, our website was created as a solution to help all of us. This website will provide users with a convenient application that meets basic needs.

- **App users:** This is the group of people who directly use this website: Doctors, nurses, head of camp.
- **Website manager:** They need control over user access and permissions to ensure data security and privacy.

Website manager has their own administrator account in the system. This provides access to higher privileges.

Manager has managing tools for configuring all system preferences as well as other related tasks.

For this website, we completed this with several technologies such as ReactJS for Front-end, NodeJS/ExpressJS for building APIs and connect to our SQL database.

4.2 User account

In our application, we already provided the 2 accounts for the admin doctor and staff for demonstration, the admin can have access to full functions of the web, the staff has access to almost every functions too but exclude the add patient's information function.

So this is the intro page of the website which the user will see first.

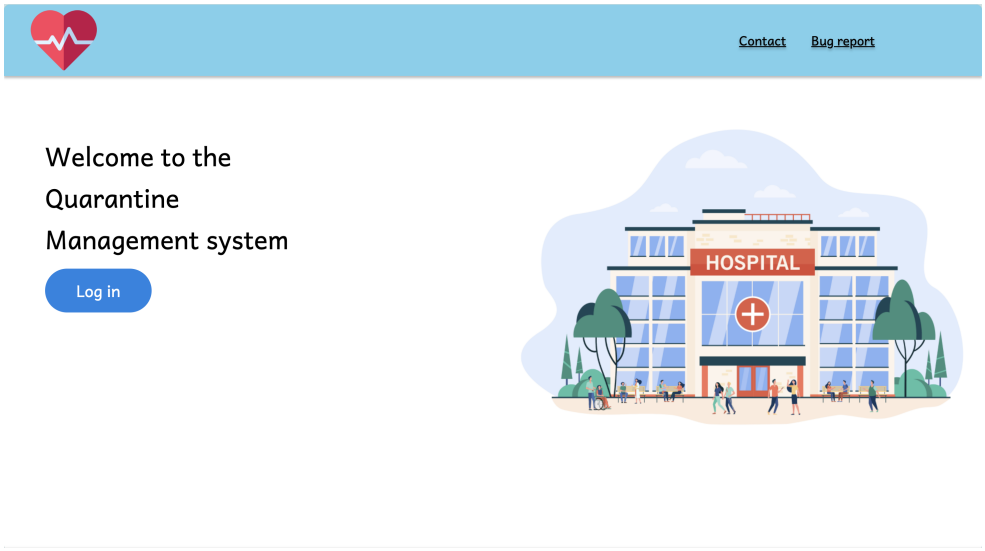


Figure 6: Portal interface

In Oracle DBMS, we create the user and password account named "manager" to login to the database with DBA privilege

```
CREATE USER c##manager identified by managerfullmana;
GRANT CREATE SESSION TO hospitalCamp;
GRANT SYSDBA TO c##manager;
```

after this implementation, any people with this account can access to the database with fully privileges.

4.3 Requirement functions

4.3.1 Log in - Log out

So before accessing to the home page and exploit all the functions, the user have to have to authenticate their identities and the web will authorize them with the right access level through the log in page.

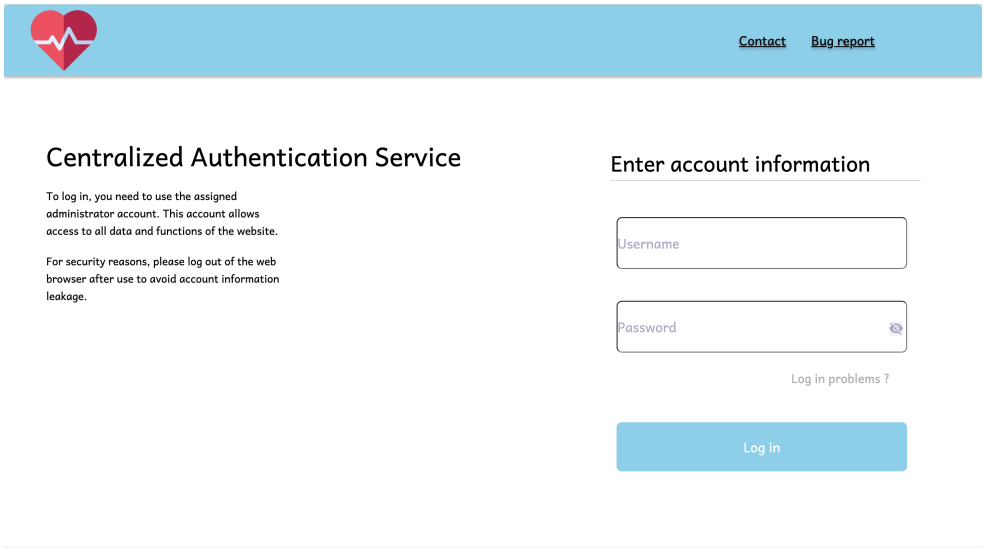


Figure 7: Login interface

```
app.post("/login", async (req, res, next) => {
  const { username, password } = req.body;

  // Perform authentication against the Oracle database
  try {
    const connection = await pool.getConnection();
    let result; // Define result outside the try block

    try {
      result = await connection.execute(
        `SELECT user_id, username FROM useraccount WHERE username = :username AND user_password = :password`,
        { username, password }
      );
      console.log(result);
    } catch (error) {
      console.error("Error executing query:", error);
      throw error; // Re-throw the error to propagate it to the outer catch block
    } finally {
      // Close the connection in the finally block
      connection.close();
    }

    if (result && result.rows.length === 1) {
      const user = {
        user_id: result.rows[0][0],
        username: result.rows[0][1],
      };
      console.log(user);
      // Generate JWT token
      const token = jwt.sign(user, "THISISMYAREABABE", { expiresIn: 1440 });

      res.cookie("token", token, {
        withCredentials: true,
        httpOnly: true,
        secure: false,
```

```
        domain: "localhost",
        path: "/",
        sameSite: "none",
      });
      res
        .status(201)
        .json({ message: "User logged in successfully", success: true, token });
    } else {
      res.status(401).json({ message: "Invalid account" });
    }
    next();
  } catch (error) {
    console.error("Error during login:", error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

After this, they will go to the Home page which contains the functions choices for the users to use as their needs. Once they finish their job, they will use log out button in the homepage to navigate back to the login page.

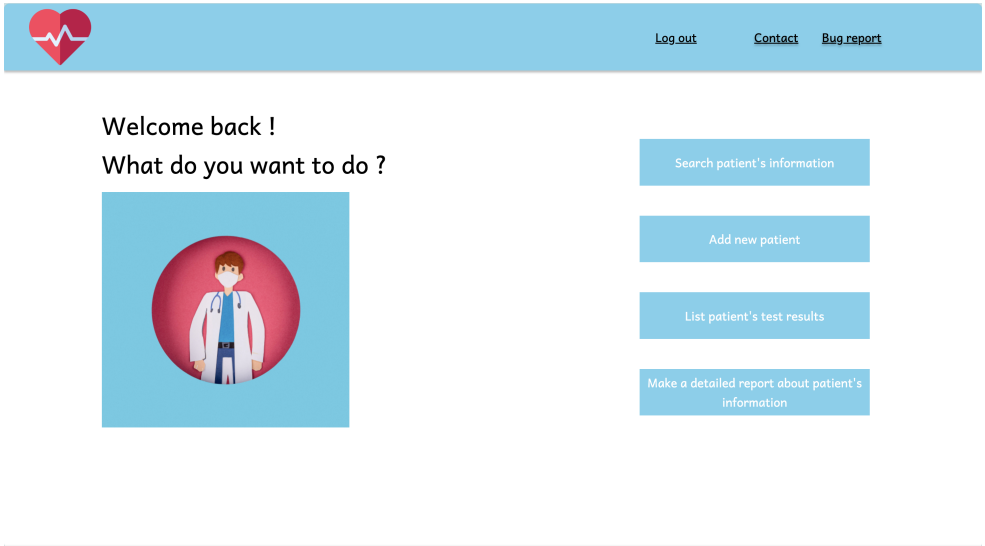


Figure 8: Homepage interface

4.3.2 Search patient's information

User will type in the patient's id as a searching keyword, and the output will be the patient name, patient phone number, patient comorbidities (if have)

```
ProtectedRoutes.get("/patients/:patient_id", async (req, res) => {
  const { patient_id } = req.params;

  try {
    const connection = await pool.getConnection();

    // Query to get full name and phone
    const basicInfoQuery =
      "SELECT patient_full_name, phone FROM Patient WHERE patient_id = :1";
```

```
    const basicInfoResult = await connection.execute(basicInfoQuery, [
      patient_id,
    ]);
    const basicInfo = basicInfoResult.rows[0];

    // Query to get comorbidities
    const comorbiditiesQuery =
      "SELECT comorbidities FROM Comorbidity WHERE patient_id = :1";
    const comorbiditiesResult = await connection.execute(comorbiditiesQuery, [
      patient_id,
    ]);
    const comorbidities = comorbiditiesResult.rows;

    connection.close();
    if (basicInfo) {
      // Combine results and send as JSON
      console.log("basicInfo:", basicInfo); // Add this line
      console.log("comorbidities:", comorbidities); // Add this line
      const result = {
        Patient_full_name: basicInfo[0].trim(),
        Phone: basicInfo[1].trim(),
        Comorbidities: comorbidities, //[0].map((c) => c.trim()),
      };
      res.json(result);
    } else {
      res
        .status(404)
        .json({ message: "No patient found with the specified patient_id" });
    }
  } catch (error) {
    console.error("Error retrieving patient information:", error);
    res.status(500).json({ message: "Internal Server Error" });
  }
});
```

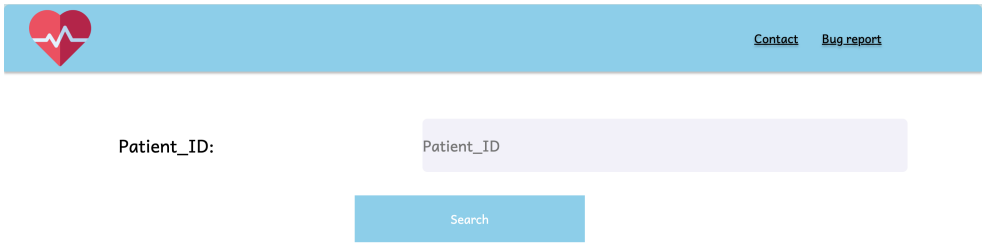


Figure 9: Searching patient's information interface

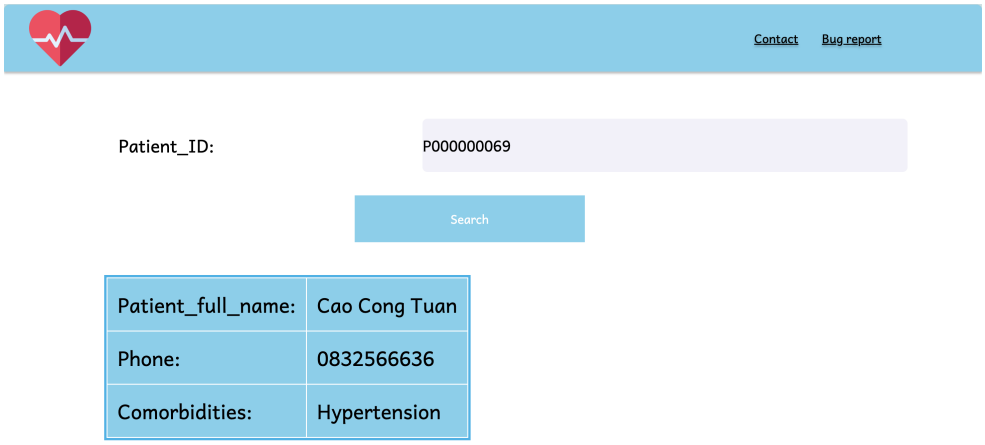


Figure 10: Result after searching patient's information

4.3.3 Add new patient's information

User will input the patient's information they want to add, every information is required except for the comorbidities.

```
ProtectedRoutes.post("/patients", async (req, res) => {
  const {
    patient_id,
    patient_full_name,
    identity_number,
    phone,
    gender,
    address,
    comorbidities, // Assuming comorbidities is an array in the request body
  } = req.body;

  try {
    // Begin a transaction
    const connection = await pool.getConnection();

    await connection.execute("BEGIN NULL; END;");

    try {
      // Insert into the Patient table
      const patientInsertQuery = `
        INSERT INTO Patient (patient_id, identity_number, patient_full_name, phone
    , gender, address)
        VALUES (:1, :2, :3, :4, :5, :6)
      `;

      const patientValues = [
        patient_id,
        identity_number,
        patient_full_name,
        phone,
        gender,
```

```
        address,
      ];
      console.log(patientValues);
      console.log(Array.isArray(comorbidities));
      console.log(comorbidities.length);
      console.log(comorbidities);
      console.log(comorbidities.at(0));
      const patientResult = await connection.execute(
        patientInsertQuery,
        patientValues,
        { autoCommit: false }
      );

      // Insert into the Comorbidity table (if comorbidities are provided)
      if (
        Array.isArray(comorbidities) &&
        comorbidities.length > 0 &&
        comorbidities.at(0) != ""
      ) {
        const comorbidityInsertQuery = `
          INSERT INTO Comorbidity (patient_id, comorbidities)
          VALUES (:1, :2)
        `;

        for (const comorbidity of comorbidities) {
          const comorbidityValues = [patient_id, comorbidity];
          await connection.execute(comorbidityInsertQuery, comorbidityValues, {
            autoCommit: false,
          });
        }
      }

      // Commit the transaction
      await connection.commit();

      // Respond with the newly added patient information
      res.status(201).json(patientResult);
    } catch (error) {
      // Rollback the transaction if an error occurs
      await connection.rollback();
      throw error;
    } finally {
      // Release the client back to the pool
      connection.close();
    }
  } catch (error) {
    console.error("Error adding new patient information:", error);
    res.status(500).json({ message: error.message || "Internal Server Error" });
  }
});
```

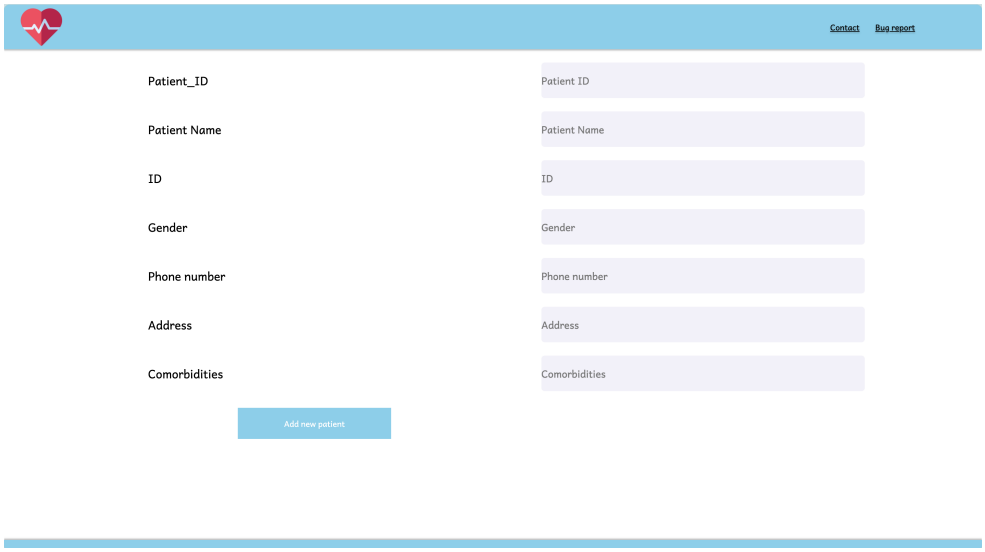


Figure 11: Add new patient's information interface

4.3.4 List of patient's testing

User will type in the patient's id as a searching keyword, the output will be the results of tests of the patient.

```
// part 3.3 API endpoint to get test results by patient_id
ProtectedRoutes.get("/patients/:patient_id/tests", async (req, res) => {
  const { patient_id } = req.params;

  try {
    const connection = await pool.getConnection();

    const testResultsQuery = `
      SELECT
        Test.test_id,
        Test.datetime,
        RespiratoryRate_Test.respiratory_result,
        SPO2_Test.SPO2_result,
        Quick_Test.quick_test_result, Quick_Test.cycle_threshold_value,
        PCR_Test.PCR_result, PCR_Test.cycle_threshold_value
      FROM Test
      LEFT JOIN RespiratoryRate_Test ON Test.test_id = RespiratoryRate_Test.test_
    id
      LEFT JOIN SPO2_Test ON Test.test_id = SPO2_Test.test_id
      LEFT JOIN Quick_Test ON Test.test_id = Quick_Test.test_id
      LEFT JOIN PCR_Test ON Test.test_id = PCR_Test.test_id
      WHERE Test.patient_id = :1
      ORDER BY Test.datetime DESC
    `;

    const testResultsResult = await connection.execute(testResultsQuery, [
      patient_id,
    ]);
    const testResults = testResultsResult.rows;

    if (testResults.length > 0) {
```

```
      res.json(testResults);
    } else {
      res.status(404).json({
        message: "No test results found for the specified patient_id",
      });
    }

    connection.close();
  } catch (error) {
    console.error("Error retrieving test results:", error);
    res.status(500).json({ message: "Internal Server Error" });
  }
});
```

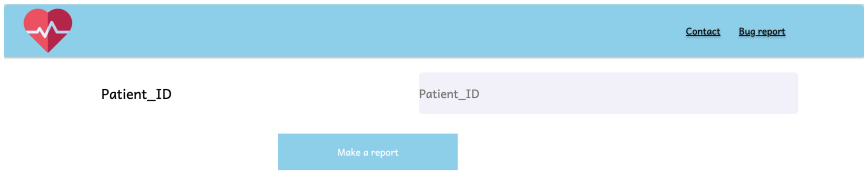


Figure 12: Patient's testing interface

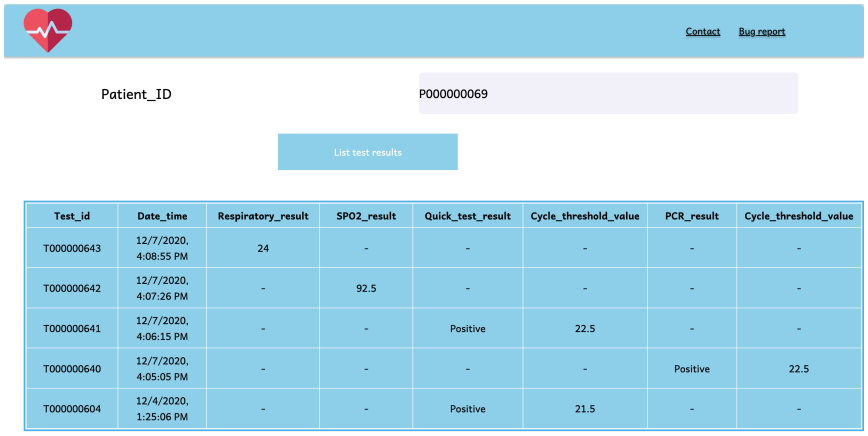


Figure 13: Result after searching patient's testing

4.3.5 Patient report

User will type in the patient's id as a searching keyword, the output will be the full information of the patient.

```
ProtectedRoutes.get("/patients/:patient_id/details", async (req, res) => {
  const { patient_id } = req.params;

  try {
    const connection = await pool.getConnection();

    // Query to get demographic info
    const demographicQuery = `
      SELECT patient_id, patient_full_name, identity_number, phone, gender,
    address
      FROM patient
      WHERE patient_id = :1
    `;

    // Query to get comorbidities
    const comorbidityQuery =
      "SELECT comorbidities FROM comorbidity WHERE patient_id = :1";

    // Query to get symptoms
    const symptomQuery = `
      SELECT symptom_name, start_date, end_date, is_serious
      FROM symptom
      WHERE patient_id = :1
    `;

    // Query to get test results
    const testResultsQuery = `
      SELECT
        Test.test_id,
        Test.datetime,
        RespiratoryRate_Test.respiratory_result,
        SPO2_Test.SPO2_result,
        Quick_Test.quick_test_result, Quick_Test.cycle_threshold_value,
        PCR_Test.PCR_result, PCR_Test.cycle_threshold_value
      FROM Test
      LEFT JOIN RespiratoryRate_Test ON Test.test_id = RespiratoryRate_Test.test_
    id
      LEFT JOIN SPO2_Test ON Test.test_id = SPO2_Test.test_id
      LEFT JOIN Quick_Test ON Test.test_id = Quick_Test.test_id
      LEFT JOIN PCR_Test ON Test.test_id = PCR_Test.test_id
      WHERE Test.patient_id = :1
      ORDER BY Test.datetime DESC
    `;

    // Query to get treatment information
    const treatmentQuery = `
      SELECT t.treatment_id, tm.initiation_date, tm.completion_date, tm.overall_
    result
      FROM TREAT t
      JOIN TREATMENT tm ON t.treatment_id = tm.treatment_id
      WHERE t.patient_id = :1
    `;

    // Execute all queries
    const demographicResult = await connection.execute(demographicQuery, [
      patient_id,
    ]);
```

```
    const comorbidityResult = await connection.execute(comorbidityQuery, [
      patient_id,
    ]);
    const symptomResult = await connection.execute(symptomQuery, [patient_id]);
    const testResultsResult = await connection.execute(testResultsQuery, [
      patient_id,
    ]);
    const treatmentResult = await connection.execute(treatmentQuery, [
      patient_id,
    ]);

    // Extract data from results
    const demographicInfo = demographicResult.rows[0];
    const comorbidities = comorbidityResult.rows;
    const symptoms = symptomResult.rows;
    const testResults = testResultsResult.rows;
    const treatments = treatmentResult.rows;

    // Combine results and send as JSON
    const result = {
      demographicInfo,
      comorbidities, //: comorbidities[0].map((c) => c.trim()),
      symptoms,
      testResults,
      treatments,
    };
    if (testResults.length > 0) {
      res.json(result);
    } else {
      res.status(404).json({
        message: "No test results found for the specified patient_id",
      });
    }

    connection.close();
  } catch (error) {
    console.error("Error retrieving patient information:", error);
    res.status(500).json({ message: "Internal Server Error" });
  }
});
```

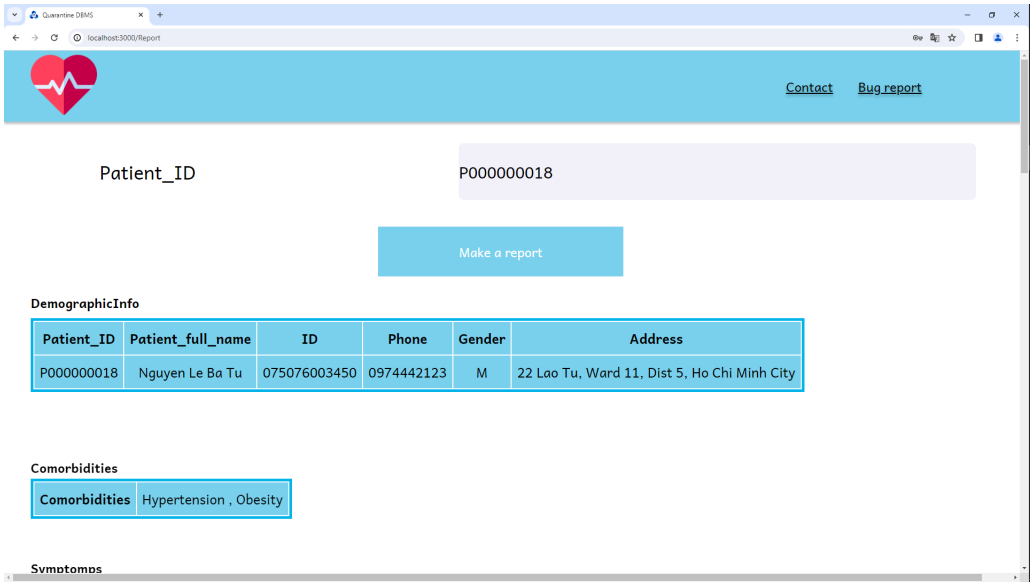


Figure 14: Patient report

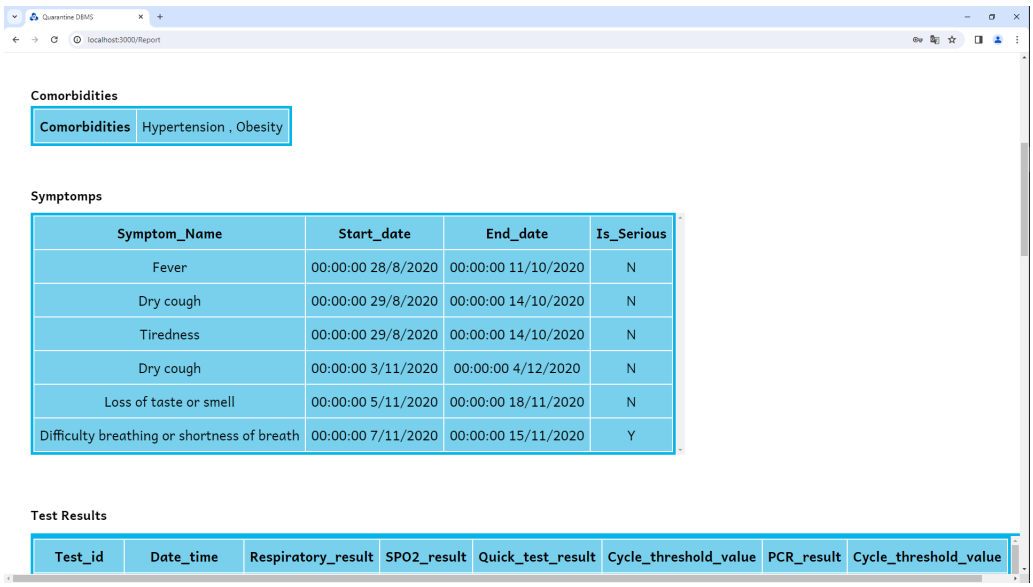
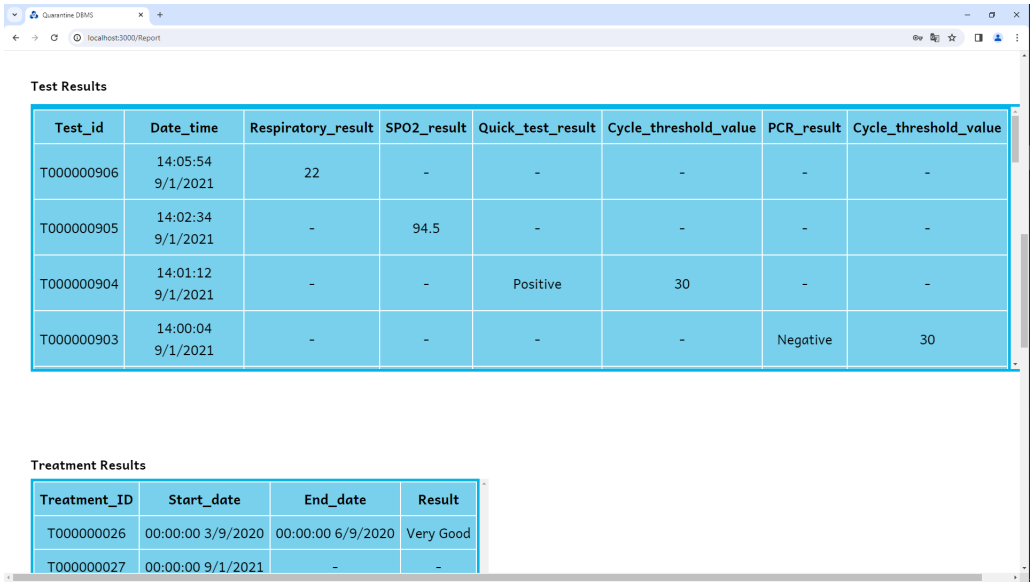


Figure 15: Patient report



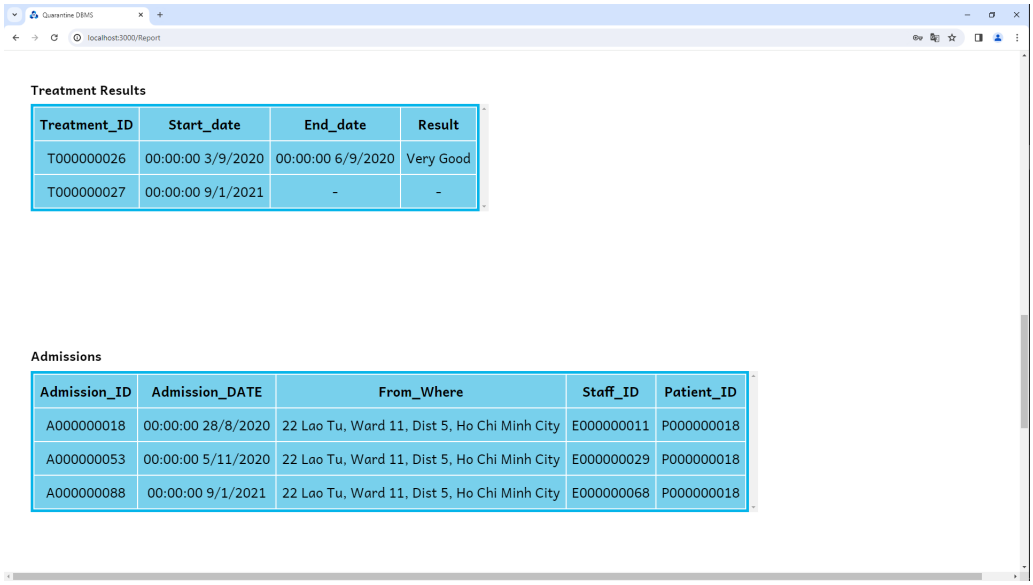
Test Results

Test_id	Date_time	Respiratory_result	SPO2_result	Quick_test_result	Cycle_threshold_value	PCR_result	Cycle_threshold_value
T000000906	14:05:54 9/1/2021	22	-	-	-	-	-
T000000905	14:02:34 9/1/2021	-	94.5	-	-	-	-
T000000904	14:01:12 9/1/2021	-	-	Positive	30	-	-
T000000903	14:00:04 9/1/2021	-	-	-	-	Negative	30

Treatment Results

Treatment_ID	Start_date	End_date	Result
T000000026	00:00:00 3/9/2020	00:00:00 6/9/2020	Very Good
T000000027	00:00:00 9/1/2021	-	-

Figure 16: Patient report



Treatment Results

Treatment_ID	Start_date	End_date	Result
T000000026	00:00:00 3/9/2020	00:00:00 6/9/2020	Very Good
T000000027	00:00:00 9/1/2021	-	-

Admissions

Admission_ID	Admission_DATE	From_Where	Staff_ID	Patient_ID
A000000018	00:00:00 28/8/2020	22 Lao Tu, Ward 11, Dist 5, Ho Chi Minh City	E000000011	P000000018
A000000053	00:00:00 5/11/2020	22 Lao Tu, Ward 11, Dist 5, Ho Chi Minh City	E000000029	P000000018
A000000088	00:00:00 9/1/2021	22 Lao Tu, Ward 11, Dist 5, Ho Chi Minh City	E000000068	P000000018

Figure 17: Patient report

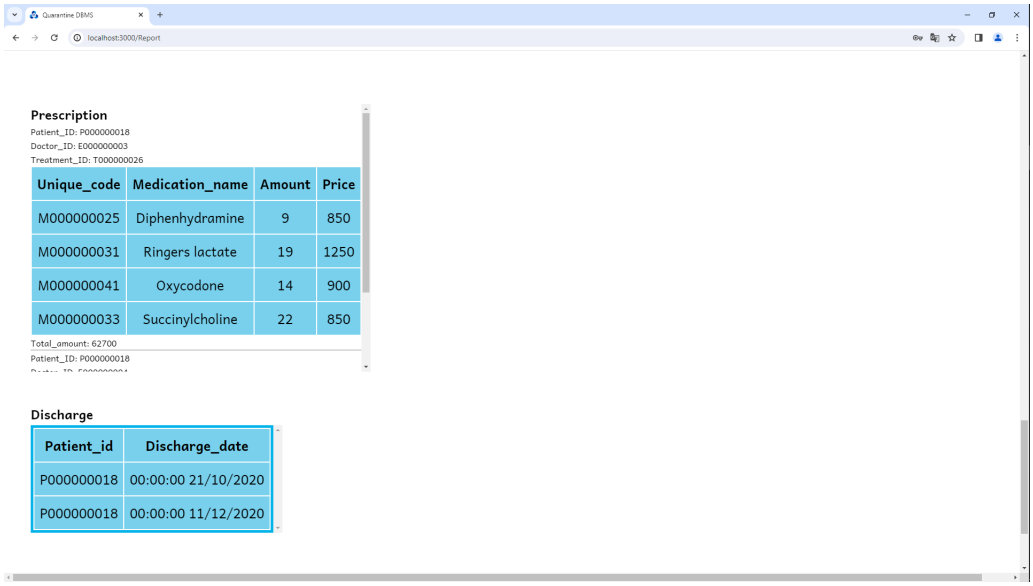


Figure 18: Patient report

5 Database management

5.1 Indexing efficiency

At the beginning, we created table PATIENT as following:

```
CREATE TABLE Patient(
    patient_id          CHAR(10)    NOT NULL,
    identity_number     CHAR(12)    PRIMARY KEY,
    patient_full_name   CHAR(30)    NOT NULL,
    phone               CHAR(10)    NOT NULL,
    gender              CHAR(5)     NOT NULL,
    address             CHAR(100)   NOT NULL,
    warning             CHAR(1)     NULL    CHECK (warning IN ('Y', 'N'))
);
```

In this table, identity_number is set as table primary key. However, in this project, especially **Part 3: Building application**, the most frequently used attribute for most of the queries is *patient_id* in the **patient** table. Users are required to insert *patient_id* in almost every case of searching data, so this attribute should be optimized for querying and further data scaling. One solution we are about to use is indexing.

Let consider this query:

```
SELECT
    Patient.patient_full_name,
    Patient.phone,
    Admission.admission_date,
    Treatment.initiation_date,
    People.first_name AS doctor_first_name,
    People.last_name AS doctor_last_name,
    LocationHistory.building,
    LocationHistory.floor,
    LocationHistory.room_number
```

```
FROM Patient
JOIN Admission ON Patient.patient_id = Admission.patient_id
JOIN Treat ON Patient.patient_id = Treat.patient_id
JOIN Treatment ON Treat.treatment_id = Treatment.treatment_id
JOIN People ON Treat.doctor_id = People.person_id
LEFT JOIN LocationHistory ON Patient.patient_id = LocationHistory.patient_id
WHERE Patient.patient_id = 'P000000069';
```

The query above contains a lot of join operations on attribute patient_id while there is no index created on column patient_id. As we just have a small amount of data (about 100 records in table PATIENT), so we create another 10 million records in table PATIENT to test the query performance intuitively.

Figure 19: Execution time without indexing on column patient_id

To query all the information, it took DBMS 1.284 seconds to complete the query, but why it took so long? Let see what happened by using the PLAN_TABLE. PLAN_TABLE is the default sample output table into which the EXPLAIN PLAN statement inserts rows describing execution plans.

```
EXPLAN PLAN FOR
--query
```

```
PLAN_TABLE_OUTPUT
Plan hash value: 2094577857

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                      |  2786 |   650K| 68670   (1)| 00:00:03 |
|*  1 |  HASH JOIN RIGHT OUTER        |                      |  2786 |   650K| 68670   (1)| 00:00:03 |
|*  2 |   INDEX SKIP SCAN             | LOCATION_HISTORY_KEY |     1 |    49 |     2   (0)| 00:00:01 |
|*  3 |   HASH JOIN                   |                      |  2786 |   516K| 68668   (1)| 00:00:03 |
|*  4 |    TABLE ACCESS FULL          | ADMISSION            |     1 |    21 |     3   (0)| 00:00:01 |
|*  5 |    HASH JOIN                  |                      |  2786 |   459K| 68665   (1)| 00:00:03 |
|   6 |     NESTED LOOPS              |                      |     1 |   113 |     4   (0)| 00:00:01 |
|   7 |      NESTED LOOPS             |                      |     1 |   113 |     4   (0)| 00:00:01 |
|   8 |       NESTED LOOPS            |                      |     1 |    57 |     3   (0)| 00:00:01 |
|*  9 |        INDEX RANGE SCAN       | TREAT_KEY            |     1 |    36 |     2   (0)| 00:00:01 |
|  10 |        TABLE ACCESS BY INDEX ROWID| TREATMENT        |     1 |    21 |     1   (0)| 00:00:01 |
|* 11 |         INDEX UNIQUE SCAN     | SYS_C008101          |     1 |       |     0   (0)| 00:00:01 |
|* 12 |       INDEX UNIQUE SCAN       | SYS_C008067          |     1 |       |     0   (0)| 00:00:01 |
|  13 |      TABLE ACCESS BY INDEX ROWID | PEOPLE            |     1 |    56 |     1   (0)| 00:00:01 |
|* 14 |     TABLE ACCESS FULL         | PATIENT              |  2786 |   152K| 68661   (1)| 00:00:03 |
-----------------------------------------------------------------------------------------------------------
```

Figure 20: Execution plan without indexing on column patient_id

After issuing the EXPLAIN PLAN statement, use a script or package provided by Oracle Database to display the most recent plan table output.

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));
```

By default, the query has to go through every records of table patient to search for a specific patient_id in each join operation and especially WHERE statement at the end need a "FULL TABLE ACCESS" as well. So, patient_id is the main problem in this situation and leads to waste of CPU consumption which is costly is every JOIN and WHERE operations include patient_id.

To speed up the query time and improve CPU consumption, we apply index on column patient_id as follow :

```
CREATE INDEX patient_idx ON Patient(patient_id);
```

PLAN_TABLE_OUTPUT

```
Plan hash value: 942560876

-------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                     |     1 |   239 |    11   (0)| 00:00:01 |
|   1 |  MERGE JOIN OUTER                        |                     |     1 |   239 |    11   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                           |                     |     1 |   190 |     9   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                          |                     |     1 |   190 |     9   (0)| 00:00:01 |
|   4 |     NESTED LOOPS                         |                     |     1 |   134 |     8   (0)| 00:00:01 |
|   5 |      MERGE JOIN CARTESIAN                |                     |     1 |   113 |     7   (0)| 00:00:01 |
|   6 |       MERGE JOIN CARTESIAN               |                     |     1 |    77 |     6   (0)| 00:00:01 |
|*  7 |        TABLE ACCESS FULL                 | ADMISSION           |     1 |    21 |     3   (0)| 00:00:01 |
|   8 |        BUFFER SORT                       |                     |     1 |    56 |     3   (0)| 00:00:01 |
|   9 |         TABLE ACCESS BY INDEX ROWID BATCHED| PATIENT           |     1 |    56 |     3   (0)| 00:00:01 |
|* 10 |          INDEX RANGE SCAN                | PATIENT_IDX         |     1 |       |     2   (0)| 00:00:01 |
|  11 |       BUFFER SORT                        |                     |     1 |    36 |     4   (0)| 00:00:01 |
|* 12 |        INDEX RANGE SCAN                  | TREAT_KEY           |     1 |    36 |     1   (0)| 00:00:01 |
|  13 |      TABLE ACCESS BY INDEX ROWID         | TREATMENT           |     1 |    21 |     1   (0)| 00:00:01 |
|* 14 |       INDEX UNIQUE SCAN                  | SYS_C008101         |     1 |       |     0   (0)| 00:00:01 |
|* 15 |     INDEX UNIQUE SCAN                    | SYS_C008067         |     1 |       |     0   (0)| 00:00:01 |
|  16 |    TABLE ACCESS BY INDEX ROWID           | PEOPLE              |     1 |    56 |     1   (0)| 00:00:01 |
|  17 |   BUFFER SORT                            |                     |     1 |    49 |    10   (0)| 00:00:01 |
|* 18 |    INDEX SKIP SCAN                       | LOCATION_HISTORY_KEY |     1 |    49 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------
```

Figure 21: Execution plan with indexing on column patient_id

5.1.1 Time efficiency

After having indexed on column patient_id, let see how long does it take the DBMS to complete the query :

Figure 22: Execution time with indexing on column patient_id

As displayed on figure above, the time for DBMS to execute the query now is 0.003 second, which is about 400x faster compared to 1.284 seconds.

5.1.2 Execution plan

Now, we dive into execution plan and consider how indexing work in a complex query with multiple JOIN. In the plan table, there are several columns :

- **ID:** A number assigned to each step in the execution plan.
- **OPERATION:** Name of the internal operation performed in this step. In the first row generated for a statement, the column contains one of the following values: DELETE , INSERT , SELECT , UPDATE, HASH, JOIN, other...
- **OBJECT_NAME:** Name of the table or index.
- **BYTES:** Estimate by the query optimization approach of the number of bytes that the operation accessed.
- **COST:** Cost of the operation as estimated by the optimizer. Cost is not determined for table access operations. The value of this column does not have any particular unit of measurement; it is a weighted value used to compare costs of execution plans. The value of this column is a function of the CPU_COST and IO_COST columns.
- **TIME:** Elapsed time in seconds of the operation as estimated by query optimization. For statements that use the rule-based approach, this column is null.

As mentioned before, when ever the query operate the JOIN or WHERE operation such as HASH JOIN (Operation joining two sets of rows and returning the result) and TABLE ACCESS FULL (Retrieval of all rows from a table), the DBMS has to search row by row in table PATIENT to find a specific record. That is the reason why, these operations are costly (in %CPU and Time). Especially, when the DBMS touch table PATIENT (Id 14) it is also costly and time consuming.

After having indexed on columns patient_id, there is a huge improvement in CPU cost and Time.

We can see the improvement on SELECT STATEMENT (63k to 11 CPU cost and lower bytes accessed) and HASH JOIN operations are no longer appear. Instead, there are some addition statements with better cost:

- **MERGE JOIN:** Operation accepting two sets of rows, each sorted by a value, combining each row from one set with the matching rows from the other, and returning the result.
- **TABLE ACCESS BY INDEX ROWID BATCHED** If the table is nonpartitioned and rows are located using index(es).
- **INDEX RANGE SCAN:** Retrieval of one or more rowids from an index. Indexed values are scanned in ascending order.

5.2 Database security - SQL Injection

Our demo application suffers from a critical security vulnerability known as SQL injection (SQLi). This attack allows malicious actors to inject harmful SQL code into the application, granting them access to the backend database. With this access, attackers can bypass security measures, gain unauthorized access to sensitive data, or even manipulate the database itself. For instance, they could access the entire database content, circumvent authentication and authorization controls, or even add, modify, or delete sensitive records.

There are a few key reasons why SQL injection occurs:

- Improper input validation and sanitization: if the application does not properly validate and sanitize the user input, malicious SQL code can be injected into the application's input fields.
- Use of dynamic SQL queries: Dynamic SQL queries are SQL queries that are constructed at runtime based on user input. This can be a powerful feature, but it also introduces the risk of SQL injection if the user input is not properly sanitized.
- Use of legacy database technologies: Some legacy database technologies are more vulnerable to SQL injection than modern database technologies. For example, some legacy database technologies do not support prepared statements, which can be used to prevent SQL injection.

1. **OR Injection**: This vulnerability occurs when the application fails to properly validate user input before using it in a SQL query. An attacker can exploit this by injecting malicious SQL code into the username or password field. In the case of OR injection, the attacker can use the following syntax to bypass authentication:

```
' or 1=1 --

--Which translate to the actual sql query.

SELECT user_id, username, role FROM user_account WHERE username = '' or 1=1
    -- AND user_password = 'password';
```

Let's break down the syntax:

- ': This single quote is used to open a string literal in SQL.
- **OR**: This keyword indicates that a statement is true if at least one of the conditions it connects is true.
- **1=1**: This statement is always true, regardless of the data it compares.
- –: This is a comment symbol in SQL. Anything that follows the "–" symbol is ignored by the SQL parser.

This statement evaluates to true regardless of the actual username or password, allowing the attacker to gain unauthorized access, as shown in Figure 23 and 24.

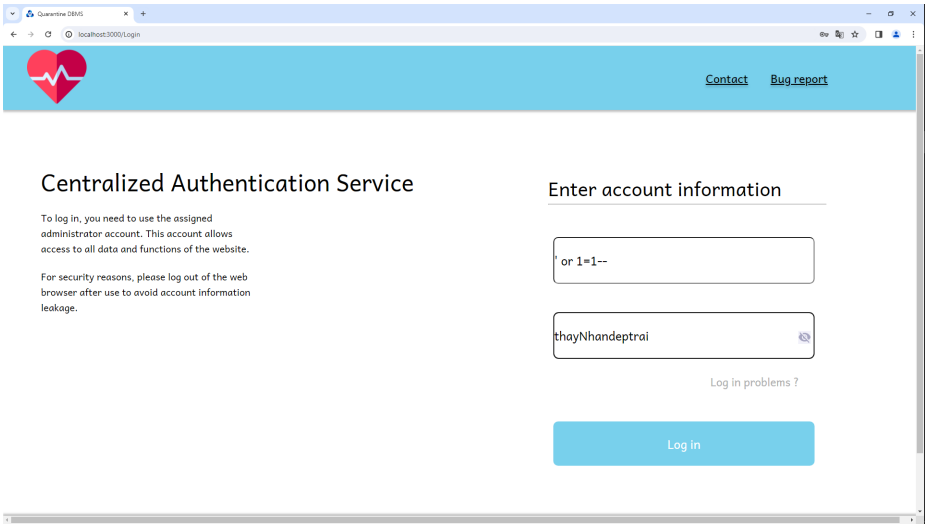


Figure 23: Entering OR injection in the login screen

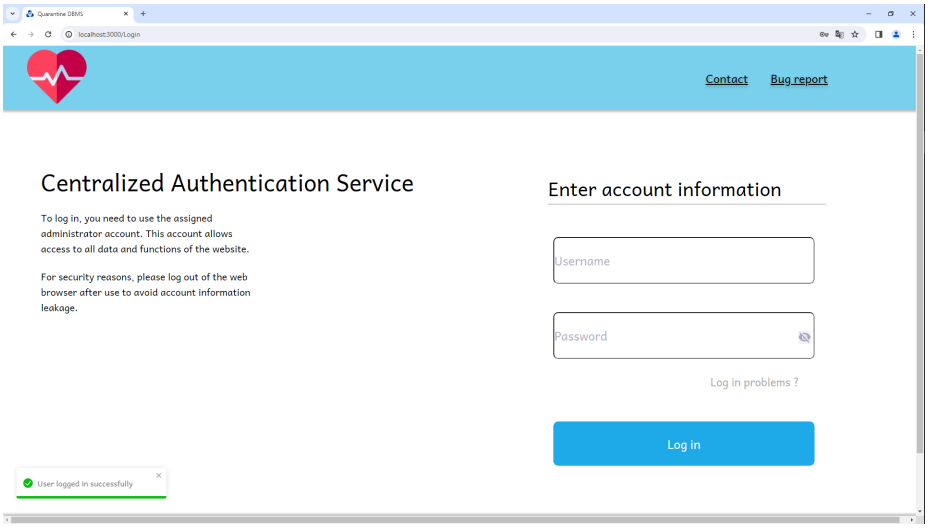


Figure 24: After or injection in login screen

2. **UNION Injection**: This vulnerability allows attackers to retrieve sensitive data from the database by injecting a malicious SQL statement containing the UNION operator. This technique is particularly dangerous if the application is vulnerable to blind SQL injection, where the attacker can infer information based on the application's response.

   An example is shown in **Figure 25**. The attacker request the patient id data but instead of the id, the attacker use **' UNION user_password from user_account** –, which return the password of all user in the database.

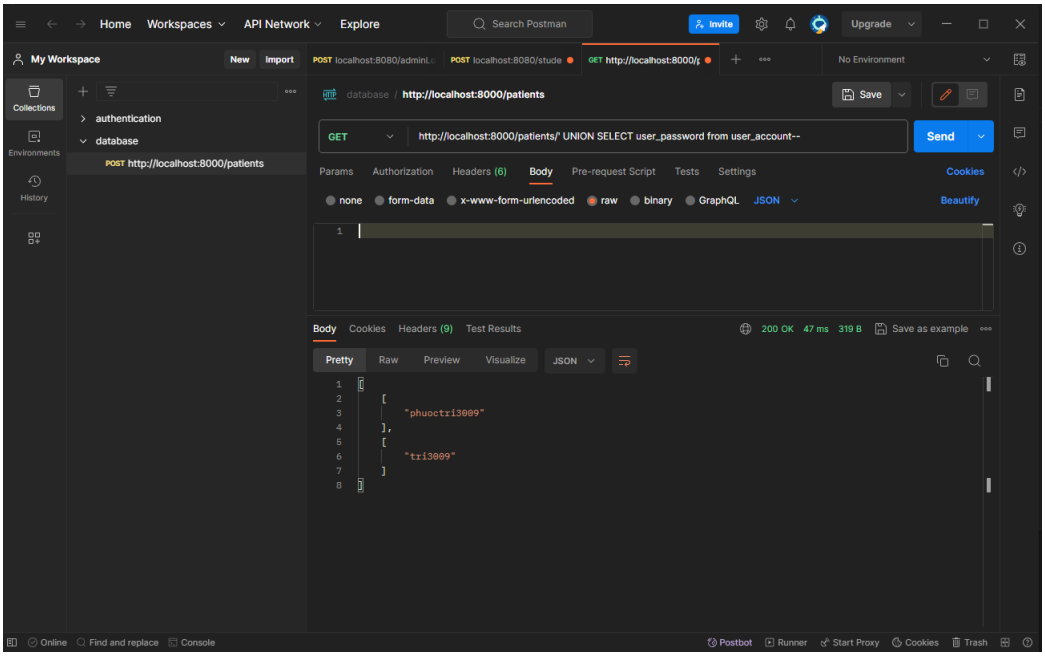


Figure 25: Using postman to inject union to the server API

Let's dive into the backend code. The SQL is construct by concatenating username and password in the middle of the string, which is vulnurable to SQL Injection.

```
// Login api
app.post("/login", async (req, res, next) => {
  const { username, password } = req.body;

  // Perform authentication against the Oracle database
  try {
    const connection = await pool.getConnection();
    let result; // Define result outside the try block

    try {
      result = await connection.execute(
        `SELECT user_id, username,role FROM user_account WHERE username = ${
    username} AND user_password = ${password}`
        );
      console.log(result);
    } catch (error) {
      console.error("Error executing query:", error);
      throw error; // Re-throw the error to propagate it to the outer catch block
    } finally {
      // Close the connection in the finally block
      connection.close();
    }
}

// The rest of the code
```

By using placeholders and then bind objects, now query prevent injection by having the bind data as a whole object instead of string to query the database.

```
// Login api
app.post("/login", async (req, res, next) => {
  const { username, password } = req.body;

  // Perform authentication against the Oracle database
  try {
    const connection = await pool.getConnection();
    let result; // Define result outside the try block

    try {
      result = await connection.execute(
        `SELECT user_id, username,role FROM user_account WHERE username = :username AND user_password = :password`,
        { username, password }
      );
      console.log(result);
    } catch (error) {
      console.error("Error executing query:", error);
      throw error; // Re-throw the error to propagate it to the outer catch block
    } finally {
      // Close the connection in the finally block
      connection.close();
    }
}

// The rest of the code
```

6 Source code

All code and SQL statements are stored on Github at this link : https://github.com/Yukinaa1q/databasemanagement.