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

DATABASE SYSTEMS (CO2013)

Assignment 2

Quarantine Camp Database

Advisor: Phan Trong Nhan
Class: CC03
Students: Phan Thanh Thong - 2153846
Phan Le Tien Thuan - 2153013
Nguyen Chanh Tin - 2153043
Cao Chanh Tri - 2153917
Tran Hoang Khoi Tuan - 2012359

HO CHI MINH CITY, NOVEMBER 2023

Contents

1 Physical Database Design

1.1 Implementing the database

For this assignment, we decided to choose PostgreSQL as our DBMS (Database Management System). The tables below lists the data type, length of each attribute as well as the constraints associated to them.

- **Patient Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	PRIMARY KEY	Each patient is assigned with a unique patient number
identity_number	VARCHAR(12)	UNIQUE	Social identity numbers must be unique but cannot be null, since a patient might be admitted many times to the camp, and cannot ensure the patient has identity number.
fullname	VARCHAR(50)	NOT NULL	Patients must provide their fullname
phone	VARCHAR(11)		Not every patient has phone number
gender	CHAR(1)	NOT NULL	Gender as Male (M) or Female (F)
address	VARCHAR(50)		Patients might not have a place called home
discharge_date	DATE		

Bảng 1: Patient Table

- **Patient comorbidity**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	NOT NULL	Part of primary key
comorbidity	VARCHAR(50)	NOT NULL	Part of primary key

Bảng 2: Patient comorbidity table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	patient_number, comorbidity	Composite primary key constraint
FOREIGN KEY	patient_number	References column patient_number of Patient table

Bảng 3: Patient Comorbidity Key Constraint

- **Symptom Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	NOT NULL	Part of primary key
date_time	TIMESTAMP	NOT NULL	Part of primary key
description	TEXT		

Bảng 4: Symptom Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	patient_number, date_time	Composite primary key constraint
FOREIGN KEY	patient_number	References column patient_number of Patient table

Bảng 5: Respiratory_Test Key Constraint

- **Personnel Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
unique_code	INT	PRIMARY KEY	Each personnel is assigned with a unique code
dob	DATE	NOT NULL	All values related to personnel is not null
fullname	VARCHAR(50)	NOT NULL	All values related to personnel is not null
phone	VARCHAR(12)	NOT NULL	All values related to personnel is not null
gender	CHAR(1)	NOT NULL	Gender as Male (M) or Female (F)
address	VARCHAR(50)	NOT NULL	All values related to personnel is not null

Bảng 6: Personnel Table

Manager, Volunteer, Nurse, Staff, and Doctor table have manager_code, volunteer_code, nurse_code, staff, and doctor_code referencing personnel unique_code, respectively.

- **Testing Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
test_id	INT	PRIMARY KEY	Every test is created with an id
date	DATE	NOT NULL	All values related to test is not null
time	TIME	NOT NULL	All values related to test is not null
patient_number	INT	NOT NULL	All values related to test is not null

Bảng 7: Testing Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	patient_number	References patient_number from Patient Table

Bảng 8: Testing Key Constraint

- **PCR_Test Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
test_id	INT	PRIMARY KEY	Every test is created with an id
ct_value	INT		
result	CHAR(1)	NOT NULL	Whether positive (+) or negative (-)

Bảng 9: PCR_Test Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	test_id	References column test_id of Testing table

Bảng 10: PCR_Test Key Constraint

- **Quick_Test Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
test_id	INT	PRIMARY KEY	Every test is created with an id
ct_value	INT		
result	CHAR(1)	NOT NULL	Whether positive (+) or negative (-)

Bảng 11: Quick_Test Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	test_id	References column test_id of Testing table

Bảng 12: Quick_Test Key Constraint

- **Respiratory_Test Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
test_id	INT	PRIMARY KEY	Every test is created with an id
breaths_ per_minute	INT	NOT NULL	Test result cannot be null

Bảng 13: Respiratory_Test Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	test_id	References column test_id of Testing table

Bảng 14: Respiratory_Test Key Constraint

- **SPO_Test Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
test_id	INT	PRIMARY KEY	Every test is created with an id
blood_ oxygen_level	INT	NOT NULL	Test result cannot be null

Bảng 15: SPO_Test Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	test_id	References column test_id of Testing table

Bảng 16: SPO_Test Key Constraint

- **Building Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
building_id	INT	PRIMARY KEY	Every building has an id

Bảng 17: Building Table

- **Floor Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
building_id	INT	NOT NULL	Part of primary key
floor_number	INT	NOT NULL	Part of primary key

Bảng 18: Floor Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	building_id, floor_number	Composite primary key constraint
FOREIGN KEY	building_id	References column building_id of Building table

Bảng 19: Floor Key Constraint

- **Room Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
building_id	INT	NOT NULL	Part of primary key
floor_number	INT	NOT NULL	Part of primary key
room_number	INT	NOT NULL	Part of primary key
capacity_number	INT	NOT NULL	Always keep track of the room capacity

Bảng 20: Room Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	building_id, floor_number, room_number	Composite primary key constraint
FOREIGN KEY	building_id	References column building_id of Floor table
FOREIGN KEY	floor_number	References column floor_number of Floor table

Bảng 21: Room Key Constraint

- **Location_history Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	NOT NULL	Part of primary key
date_time	TIMESTAMP	NOT NULL	Part of primary key
building_id	INT	NOT NULL	Always keep track of patient location
floor_number	INT	NOT NULL	Always keep track of patient location
room_number	INT	NOT NULL	Always keep track of patient location

Bảng 22: Location_history Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	patient_number, date_time,	Composite primary key constraint
FOREIGN KEY	building_id	References column building_id of Room table
FOREIGN KEY	floor_number	References column floor_number of Room table
FOREIGN KEY	room_number	References column room_number of Room table

Bảng 23: Location_history Key Constraint

- **Admit Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	PRIMARY KEY	patient_number as a primary key
staff_code	INT	NOT NULL	staff_code cannot be null
admission_date	DATE	NOT NULL	admission_date cannot be null
from	VARCHAR(50)		

Bảng 24: Admit Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	patient_number	References column patient_number of Patient table
FOREIGN KEY	staff_code	References column staff_code of Staff table

Bảng 25: Admit Key Constraint

- **Take_care_of Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	PRIMARY KEY	patient_number as a primary key
nurse_code	INT	NOT NULL	nurse_code cannot be null
start_date	DATE	NOT NULL	start_date cannot be null
end_date	DATE		taking care of patient might not have an end_date yet

Bảng 26: Take_care_of Table

TYPE	COLUMNS	EXPLANATION
FOREIGN KEY	patient_number	References column patient_number of Patient table
FOREIGN KEY	nurse_code	References column nurse_code of Nurse table

Bảng 27: Take_care_of Key Constraint

- **Treatment Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
patient_number	INT	NOT NULL	Part of primary key
start_date	DATE	NOT NULL	Part of primary key
end_date	DATE	NOT NULL	Part of primary key
result	TEXT		

Bảng 28: Treatment Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	patient_number, start_date, end_date,	Composite primary key constraint
FOREIGN KEY	patient_number	References column patient_number of Patient table

Bảng 29: Treatment Key Constraint

- **Treat Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
doctor_code	INT	NOT NULL	Part of primary key
patient_number	INT	NOT NULL	Part of primary key
start_date	DATE	NOT NULL	Part of primary key
end_date	DATE	NOT NULL	Part of primary key

Bảng 30: Treat Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	doctor_code, patient_number, start_date, end_date,	Composite primary key constraint
FOREIGN KEY	doctor_code	References column doctor_code of Doctor table
FOREIGN KEY	patient_number	References column patient_number of Treatment table
FOREIGN KEY	start_date	References column start_date of Treatment table
FOREIGN KEY	end_date	References column end_date of Treatment table

Bảng 31: Treat Key Constraint

- **Medication Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
med_code	INT	PRIMARY KEY	med_code as primary key
expiration_date	DATE		some medication might not have expiration date
name	VARCHAR(50)	NOT NULL	name cannot be null
price	INT	NOT NULL	price cannot be null
effects	TEXT		

Bảng 32: Medication Table

- **Use Table**

NAME	DATA TYPE	CONSTRAINT	EXPLANATION
med_code	INT	NOT NULL	Part of primary key
patient_number	INT	NOT NULL	Part of primary key
start_date	DATE	NOT NULL	Part of primary key
end_date	DATE	NOT NULL	Part of primary key

Bảng 33: Use Table

TYPE	COLUMNS	EXPLANATION
PRIMARY KEY	med_code, patient_number, start_date, end_date,	Composite primary key constraint
FOREIGN KEY	med_code	References column med_code of Medication table
FOREIGN KEY	patient_number	References column patient_number of Treatment table
FOREIGN KEY	start_date	References column start_date of Treatment table
FOREIGN KEY	end_date	References column end_date of Treatment table

Bảng 34: Use Key Constraint

1.2 Insert data

- Insert data for all tables in the database.
- Requirements: The data in the tables must be meaningful, and each table has at least 4 rows.

To expand and diversify our database, we've created additional SQL insert statements.

```
INSERT INTO patient (identity_number, fullname, phone, gender, address,
    discharge_date)
VALUES
('121268','Nguyen Van A','387376','M','424 Thanh Thai','2020/12/3'),
('140403','Tran Anh Tuan','683267','M','15 Su Van Hanh','2020/12/26'),
('030276','Nguyen Hua Minh','319749','M','248 Ly Thuong Kiet','2020/11/30'),
('180901','Vu Nguyen','345685','M','23/3 Cao Thang','2021/2/20'),
('150803','Nguyen Van A','214429','F','134 Huynh Van Banh','2021/1/1');
```

Insert into patient table

```
INSERT INTO symptom VALUES
(1,'2020/12/1','Muscle or body aches'),
(2,'2020/12/14','New loss of taste or smell'),
(3,'2020/11/12','Shortness of breath or difficulty breathing'),
(4,'2021/2/2','Congestion or runny nose'),
(5,'2020/12/21','Diarrhea'),
(5,'2020/12/22','New loss of taste or smell');
```

Insert into symptom table

```
INSERT INTO patient_comorbidity VALUES
(1,'Cancer'),
(2,'Chronic lung diseases'),
(3,'Diabetes'),
(4,'None'),
(5,'Immunocompromised state');
```

Insert into patient.comorbidity table

```
SELECT * FROM building;
INSERT INTO public.building DEFAULT VALUES;
INSERT INTO public.building DEFAULT VALUES;
INSERT INTO public.building DEFAULT VALUES;
INSERT INTO public.building DEFAULT VALUES;
INSERT INTO public.building DEFAULT VALUES;
```

Insert into building table

```
INSERT INTO floor VALUES (1,1),(2,1),(3,1),(4,1),(5,1);
```

Insert into floor table

```
INSERT INTO room VALUES
(1,1,101,100),(2,1,202,200),(3,1,303,300),(4,1,404,400),(5,1,505,500);
```

Insert into room table

```
INSERT INTO personnel (dob, fullname, phone, gender, address)
VALUES
('1967/04/03','Truong Tuan Tu','685824','M','14 Pasteur'),
('1968/02/02','Nguyen Tien Thai','424802','M','235 Hoa Hao'),
('1969/01/08','Nguyen Thi Thuy Diem','917693','F','34 Truong Dinh'),
('1969/02/10','Dao Thi Minh Chau','663033','F','45 Quang Trung'),
('1971/02/08','Nguyen Thi Phuong','235893','F','434 Nguyen Hue'),
('1971/07/22','Nguyen Chanh Tin','837017','M','32 Nguyen Tri Phuong'),
('1974/05/30','Pham Hoang Minh','522321','M','358/23/24 Cong Hoa'),
('1975/03/13','Truong Anh Quan','261454','M','6969 3/2'),
('1975/04/22','Le Trong Kien','784831','M','352/3/2/4 CMT8'),
('1976/08/06','Hoang Dai Nam','141724','M','43/7 To Hien Thanh'),
('1977/09/28','Tran Nguyen Gia Phat','178932','M','437 Tran Hung Dao'),
('1978/05/12','Phan Thanh Thong','314822','M','561 Hoa Binh'),
('1978/10/08','Nguyen Tuong Vy','674012','F','13/4 Tran Quoc Thao'),
('1978/11/29','Tran Diem My','324096','F','74 Tran Nhat Duat'),
('1979/03/16','Le Quoc Khai','844076','M','90 Ho Bieu Chanh'),
('1979/03/17','Duong Qua','315159','M','49 Luy Ban Bich'),
('1980/06/09','Co Long','288260','F','49 Luy Ban Bich'),
('1981/06/04','Phan Dat','482645','M','66 Ong Ich Khiem'),
('1982/06/30','Tran Anh Tu','774197','M','7749 Ta Quang Buu'),
('1983/04/20','Phan Tan Trung','213902','F','132 Ly Thai To'),
('1983/06/13','Phan Le Tien Thuan','140403','M','1 Nguyen Hue');
```

Insert into personnel table

```
INSERT INTO doctor VALUES (1),(2),(3),(4),(5),(21);
```

Insert into doctor table

```
INSERT INTO volunteer VALUES (6),(7),(8),(9),(10);
```

Insert into volunteer table

```
INSERT INTO nurse VALUES (11),(12),(13),(14),(15);
```

Insert into nurse table

```
INSERT INTO staff VALUES (16),(17),(18),(19),(20);
```

Insert into staff table

```
INSERT INTO manager VALUES (1),(2),(3),(4),(5),(21);
```

Insert into manager table

```
INSERT INTO medication (expiration_date, name, price, effects)
VALUES
('2025/1/1','Remdesivir',2000000,'An antiviral medication'),
('2026/1/1','Dexamethasone',1490000,'A corticosteroid with anti-inflammatory
    properties'),
('2026/1/1','Tocilizumab',1200000,'Inhibits the activity of interleukin-6 (IL-6)')
    ,
('2025/1/1','Convalescent Plasm',2500000,'Provide passive immunity to the
    recipient, helping their immune system fight the infection'),
('2027/1/1','Monoclonal Antibodies',3600000,'Neutralize the virus and reduce the
    severity of symptoms'),
('2026/1/1','Molnupiravir',2400000,'Reduce the viral load and limit the
    progression of COVID-19');
```

Insert into medication table

```
INSERT INTO admit VALUES
(1,16,'2020/10/24','Hung Vuong Hospital'),
(2,17,'2020/11/15','15 Su Van Hanh'),
(3,18,'2020/10/19','Cho Ray Hospital'),
(4,19,'2020/01/01','23/3 Cao Thang'),
(5,20,'2020/01/01','134 Huynh Van Banh');
```

Insert into admit table

```
INSERT INTO take_care_of VALUES
(1,15,'2020/10/25','2020/12/1'),
(2,15,'2020/11/17','2020/12/20'),
(3,13,'2020/10/23','2020/11/25'),
(4,12,'2021/1/4','2021/2/12'),
(5,11,'2020/11/27','2020/12/28');
```

Insert into take.care.of table

```
INSERT INTO treatment VALUES
(1,'2020/10/26','2020/11/13','Fine'),
(2,'2020/11/19','2020/12/12','Fine'),
(3,'2020/10/27','2020/11/10','Fine'),
(4,'2021/1/4','2021/1/23','Not Fine'),
(5,'2020/11/27','2020/12/28','Fine'),
(4,'2021/1/27','2021/2/8','Fine');
```

Insert into treatment table

```
INSERT INTO treat VALUES
(1,1,'2020/10/26','2020/11/13'),
(2,2,'2020/11/19','2020/12/12'),
(3,3,'2020/10/27','2020/11/10'),
(4,4,'2021/1/4','2021/1/23'),
(5,5,'2020/11/27','2020/12/28'),
(21,4,'2021/1/27','2021/2/8'),
(21,4,'2021/1/4','2021/1/23'),
(4,4,'2021/1/27','2021/2/8');
```

Insert into treat table

```
INSERT INTO use VALUES
(3,1,'2020/10/26','2020/11/13'),
(2,2,'2020/11/19','2020/12/12'),
(5,3,'2020/10/27','2020/11/10'),
(1,4,'2021/1/4','2021/1/23'),
(1,5,'2020/11/27','2020/12/28'),
(4,4,'2021/1/27','2021/2/8');
```

Insert into use table

```
INSERT INTO location_history VALUES
(1,'2020/10/26',2,1,202),
(2,'2020/11/19',1,1,101),
(3,'2020/10/27',3,1,303),
(4,'2021/1/4',1,1,101),
(5,'2020/11/27',5,1,505),
(4,'2021/1/27',4,1,404);
```

Insert into location.history table

```
INSERT INTO testing ("date", "time", patient_number)
VALUES
('2020/10/25','9:00',1),('2020/11/16','9:00',2),
('2020/10/20','9:00',3),('2021/1/4','9:00',4),
('2020/10/24','9:00',5),('2020/10/31','10:00',1),
('2020/11/23','10:00',2),('2020/10/27','10:00',3),
('2021/1/11','10:00',4),('2020/11/1','10:00',5),
('2020/11/7','7:00',1),('2020/11/30','7:00',2),
('2020/11/3','7:00',3),('2021/1/18','7:00',4),
('2020/11/8','7:00',5),('2020/11/14','8:00',1),
('2020/12/7','8:00',2),('2020/11/10','8:00',3),
('2021/1/25','8:00',4),('2020/11/15','8:00',5);
```

Insert into testing table

```
INSERT INTO quick_test
VALUES (1,27,'+'),(2,25,'-'),(3,24,'+'),(4,22,'+'),(5,29,'+');
```

Insert into quick.test table

```
INSERT INTO pcr_test
VALUES (6,25,'+'),(7,26,'-'),(8,26,'+'),(9,21,'-'),(10,30,'-');
```

Insert into pcr.test table

```
INSERT INTO respiratory_test
VALUES (11,15),(12,17),(13,16),(14,13),(15,18);
```

Insert into respiratory.test table

```
INSERT INTO spo_test
VALUES (16,93),(17,94),(18,92),(19,90),(20,92);
```

Insert into spo.test table

2 Store Procedure / Function / SQL

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

In order to update the PCR test results for patients admitted from 01/09/2020, we used UPDATE statement. The process involves targeting the pcr_test table, specifically the result and ct_value columns, and modifying them accordingly. To identify the relevant test IDs for the update, we JOINED the testing and admit tables, ensuring a match on the patient numbers, and filtering records based on the admission date being greater than or equal to '2020-09-01'. After that, the SET clause is employed to change the result to '+' (indicating a positive test result) and the ct_value to NULL.

```
UPDATE pcr_test
SET result = '+', ct_value = NULL
WHERE test_id IN (
  SELECT t.test_id
  FROM testing t
  JOIN admit a ON t.patient_number = a.patient_number
  WHERE a.admission_date >= '2020-09-01'
);
```

a. Update patient PCR test

We used SELECT patient, LEFT JOIN with admit, JOIN with testing and pcr_test to see the result:

```
SELECT p.patient_number, p.identity_number, p.fullname, ad.admission_date, pt.
    test_id, pt.ct_value, pt.result
FROM patient p
LEFT JOIN
  admit ad on ad.patient_number=p.patient_number
JOIN
    testing t ON p.patient_number = t.patient_number
JOIN
  pcr_test pt on t.test_id = pt.test_id
order by p.patient_number asc;
```

Before UPDATE:

	patient_number integer	identity_number character varying (10)	fullname character varying (20)	admission_date date	test_id integer	ct_value integer	result character
1	1	121268	Nguyen Van A	2020-10-24	6	25	+
2	2	140403	Tran Anh Tuan	2020-11-15	7	26	-
3	3	030276	Nguyen Hua Minh	2020-10-19	8	26	+
4	4	180901	Vu Nguyen	2020-01-01	9	21	-
5	5	150803	Nguyen Van A	2020-01-01	10	30	-

After UPDATE:

	patient_number integer	identity_number character varying (10)	fullname character varying (20)	admission_date date	test_id integer	ct_value integer	result character
1	1	121268	Nguyen Van A	2020-10-24	6	[null]	+
2	2	140403	Tran Anh Tuan	2020-11-15	7	[null]	+
3	3	030276	Nguyen Hua Minh	2020-10-19	8	[null]	+
4	4	180901	Vu Nguyen	2020-01-01	9	21	-
5	5	150803	Nguyen Van A	2020-01-01	10	30	-

We can see that patients whose admission date is from 01/09/2020 have ct_value = null and result = +.

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

We retrieves comprehensive information for patients with the name 'Nguyen Van A' by combining data from multiple tables. The patient table is the primary source of patient information, and the query includes columns such as patient_number, identity_number, fullname, phone, gender, address, and discharge_date. Furthermore, the query incorporates left joins with the symptom and patient_comorbidity tables to capture details about symptoms experienced by the patient and any comorbidities they may have. The STRING_AGG function is employed to concatenate distinct symptom descriptions and comorbidities into a single column, separated by newline characters and commas, respectively.

```
SELECT
    p.patient_number,
    p.identity_number,
    p.fullname,
    p.phone,
    p.gender,
    p.address,
    p.discharge_date,
  lc.room_number,
  lc.floor_number,
  lc.building_id,
    STRING_AGG(DISTINCT CONCAT(s.date_time, ': ', s.description), E'\n') AS
    symptoms,
  STRING_AGG(DISTINCT CONCAT(pc.comorbidity), E', ') AS comorbidity
FROM
    patient p
LEFT JOIN
  public.location_history lc ON p.patient_number = lc.patient_number
LEFT JOIN
    public.symptom s ON p.patient_number = s.patient_number
LEFT JOIN
  public.patient_comorbidity pc ON p.patient_number = pc.patient_number
WHERE
    p.fullname = 'Nguyen Van A'
GROUP BY
    p.patient_number, p.identity_number, p.fullname, p.phone, p.gender, p.address,
     p.discharge_date,
  lc.room_number, lc.floor_number, lc.building_id;
```

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

The results:

patient_number integer	identity_number character varying (10)	fullname character varying (20)	phone character varying (10)	gender character	address character varying (20)	discharge_date date
1	121268	Nguyen Van A	387376	M	424 Thanh Thai	2020-12-03
5	150803	Nguyen Van A	214429	F	134 Huynh Van Banh	2020-01-01

room_number integer	floor_number integer	building_id integer	symptoms text	comorbidity text
202	1	2	2020-12-01 00:00:00: Muscle or body aches	Cancer
505	1	5	2020-12-21 00:00:00: Diarrhea 2020-12-22 00:00:00: New loss of taste or smell	Immunocompromised state

× Cancel

2.3 Write a function to calculate the testing for each patient

Input: Patient ID Output: A list of testing In this task, We create a PostgreSQL function, named get_testing_for_patient, designed to retrieve testing information for a specific patient identified by their identity_number. The function returns a table with columns such as test_id, test_date, test_time, pcr_test_result, quick_test_result, spo2_test_result, and respiratory_test_result. The function uses a RETURN QUERY statement combined with a SQL SELECT query to fetch data from the testing table. It employs LEFT JOINs with the pcr_test, quick_test, spo_test, and respiratory_test tables to gather details about different types of tests associated with each testing event.

```
CREATE OR REPLACE FUNCTION get_testing_for_patient(identity_number character
    varying(10))
RETURNS TABLE (
    test_id integer,
    test_date date,
    test_time time without time zone,
    pcr_test_result integer,
    quick_test_result integer,
    spo2_test_result integer,
    respiratory_test_result integer
)
AS $$
BEGIN
    RETURN QUERY
    SELECT
        t.test_id,
        t.date,
        t.time,
    pcr.ct_value,
    q.ct_value,
    spo.blood_oxygen_level,
    res.breaths_per_minute
    FROM
        testing t
  LEFT JOIN pcr_test pcr ON pcr.test_id = t.test_id
  LEFT JOIN quick_test q ON q.test_id = t.test_id
```

```
  LEFT JOIN spo_test spo ON spo.test_id = t.test_id
  LEFT JOIN respiratory_test res ON res.test_id = t.test_id
    WHERE
        t.patient_number = (SELECT patient_number FROM patient p WHERE p.
    identity_number = get_testing_for_patient.identity_number);
END;
$$ LANGUAGE plpgsql;
```

c. Write a function to calculate the testing for each patient

We call the function by input a patient with identify_number = '150803'

```
SELECT * FROM get_testing_for_patient('150803');
```

The result:

test_id integer	test_date date	test_time time without time zone	pcr_test_result integer	quick_test_result integer	spo2_test_result integer	respiratory_test_result integer
5	2020-10-24	09:00:00	[null]	29	[null]	[null]
10	2020-11-01	10:00:00	30	[null]	[null]	[null]
15	2020-11-08	07:00:00	[null]	[null]	[null]	18
20	2020-11-15	08:00:00	[null]	[null]	92	[null]

2.4 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.

We create a Postgres function named sort_nurses_by_patient_count. The function takes two input parameters, namely start_date and end_date, representing the beginning and end of the time range under consideration. The function utilizes a RETURN QUERY statement along with a SQL SELECT query to extract relevant information from the database. The SELECT query employs LEFT JOINs with the take_care_of, nurse, personnel, and patient tables to establish connections between nurses, patients, and their care-taking periods.

```
CREATE OR REPLACE FUNCTION sort_nurses_by_patient_count(start_date date, end_date
    date)
RETURNS TABLE (nurse_code integer, nurse_name varchar, patient_count bigint)
AS $$
BEGIN
    RETURN QUERY
    SELECT
        tc.nurse_code,
        person.fullname AS nurse_name,
        COUNT(DISTINCT tc.patient_number) AS patient_count
    FROM
        public.take_care_of tc
    LEFT JOIN
        public.nurse n ON tc.nurse_code = n.nurse_code
  LEFT JOIN
      public.personnel person ON person.unique_code = n.nurse_code
    LEFT JOIN
        public.patient p ON tc.patient_number = p.patient_number
    WHERE
```

```
        tc.start_date >= sort_nurses_by_patient_count.start_date
        AND (tc.end_date IS NULL OR tc.end_date <= sort_nurses_by_patient_count.
    end_date)
    GROUP BY
        tc.nurse_code, person.fullname
    ORDER BY
        patient_count DESC;
END;
$$ LANGUAGE plpgsql;
```

d. Write a procedure to sort the nurses in decreasing number of patients

The result is a table with columns for nurse_code, nurse_name, and patient_count, which represents the unique nurse identifier, the nurse's name, and the count of patients they have taken care of, respectively.
The result:

nurse_code integer	nurse_name character varying	patient_count bigint
15	Le Quoc Khai	2
11	Tran Nguyen Gia Phat	1
12	Phan Thanh Thong	1
13	Nguyen Tuong Vy	1

3 Building Applications

3.1 Create User

Creating a user with DBA privileges involves logging into the database using administrator-level credentials, such as SYS or SYSTEM. In this scenario, a new user named "andy@email.com" is established to manage database resources and operations. The DBA (Database Administrator) grants this user all access rights, including privileges to execute administrative commands, create and modify database objects, and manipulate data. Assigning comprehensive access rights to the "Andy" user empowers them with the authority to oversee and administer the database efficiently. It is essential to exercise caution and adhere to security best practices when creating users with elevated privileges, ensuring the integrity and confidentiality of the database are maintained. This user creation process is fundamental for facilitating effective database management and ensuring that authorized personnel have the necessary permissions to carry out administrative tasks.

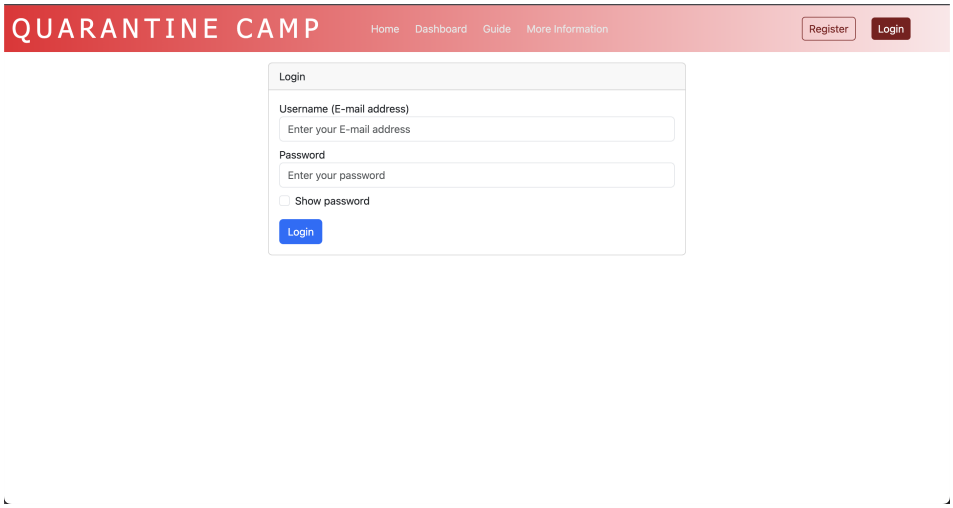

Figure 1. Login

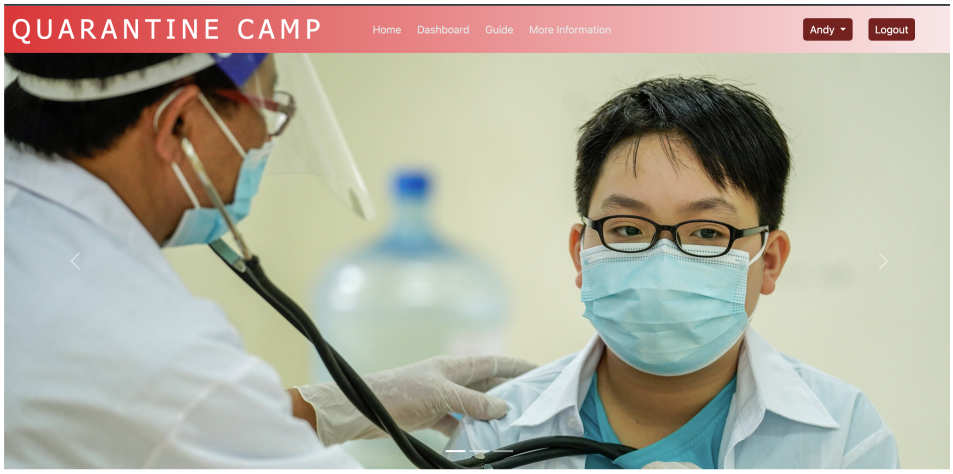

Figure 2. Logined

3.2 Requirement Functions

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

The patient information search functionality provides a comprehensive view of individuals within the system, offering key details such as their full name, contact phone number, and pertinent information about any existing comorbidities. This feature serves as a vital tool in accessing and managing patient records efficiently. Users can input various search parameters, including full names, phone numbers, admission dates, identity numbers, and gender, to pinpoint specific individuals within the database. The search results not only display core patient details but also incorporate information related to comorbidities, providing a holistic overview of the individual's health profile. This comprehensive approach enhances the ability of healthcare professionals to retrieve accurate and relevant patient data swiftly, facilitating informed decision-making and ensuring the delivery of precise and tailored medical care.

Selecting specific fields, such as full name, phone number, admission date, identity number, and gender, for patient information search is a strategic decision aimed at optimizing the search process and ensuring relevant and accurate results. These fields represent essential and distinctive aspects of an individual's identity and medical history. Full name serves as a primary identifier, phone number facilitates contact, admission date provides temporal context, identity number ensures unique patient identification, and gender offers a crucial demographic detail. By focusing on these key fields, the search functionality is streamlined to gather pertinent information efficiently. This targeted approach not only expedites the retrieval of specific patient records but also reduces the likelihood of irrelevant or duplicate results. It aligns with the practical needs of healthcare professionals, allowing them to swiftly access and assess critical patient data for informed decision-making and tailored medical care.

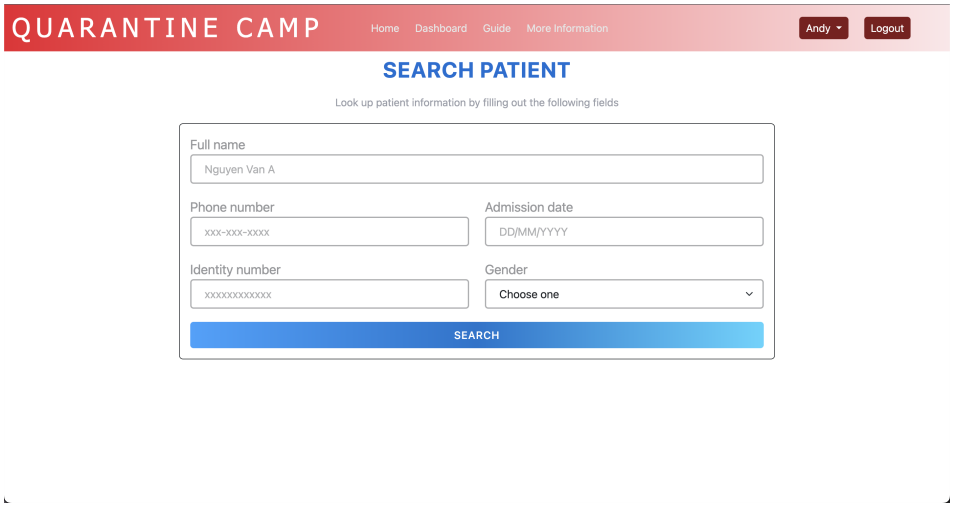

Figure 3. Search form

You can search by entering values into each field, or you can skip fields that do not have values. The system will automatically search and provide the most accurate results.

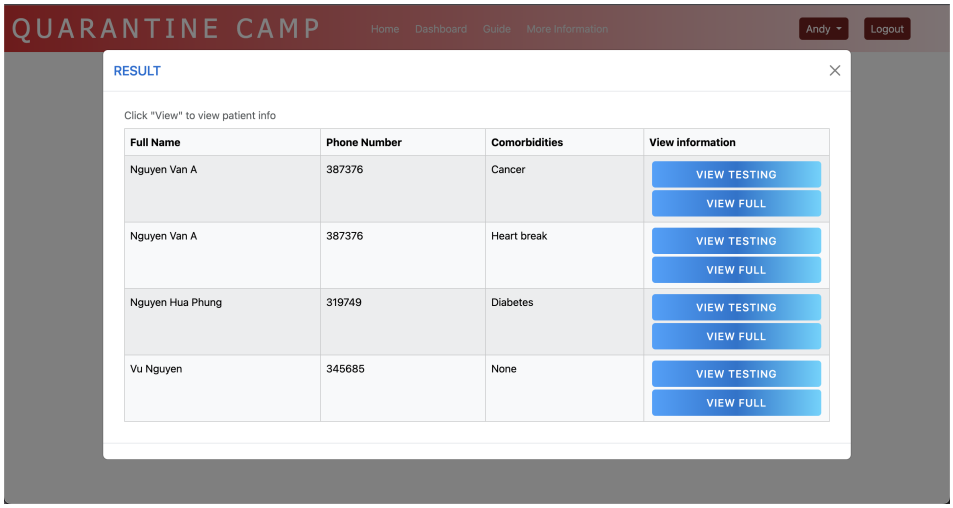

Figure 4. Search result with Full name input is "Nguyen"

3.2.2 Add information for a new patient.

The addition of information for a new patient involves a meticulous process to ensure a comprehensive and accurate representation of their health profile. Beginning with demographic details, such as the patient's full name, contact number, admission date, identity number, and gender, establishes the foundational identity of the individual within the system. Subsequent inclusion of relevant comorbidities, symptoms, and any pre-existing health conditions contributes to a holistic understanding of the patient's medical history. To enhance diagnostic precision, information regarding any testing undergone by the patient is

recorded, encompassing laboratory tests, imaging studies, and other diagnostic procedures. Finally, documenting the details of the treatment plan, whether ongoing or completed, provides a dynamic snapshot of the patient's healthcare journey. This meticulous process not only ensures a thorough and accurate representation of the new patient but also sets the stage for personalized and effective medical care.

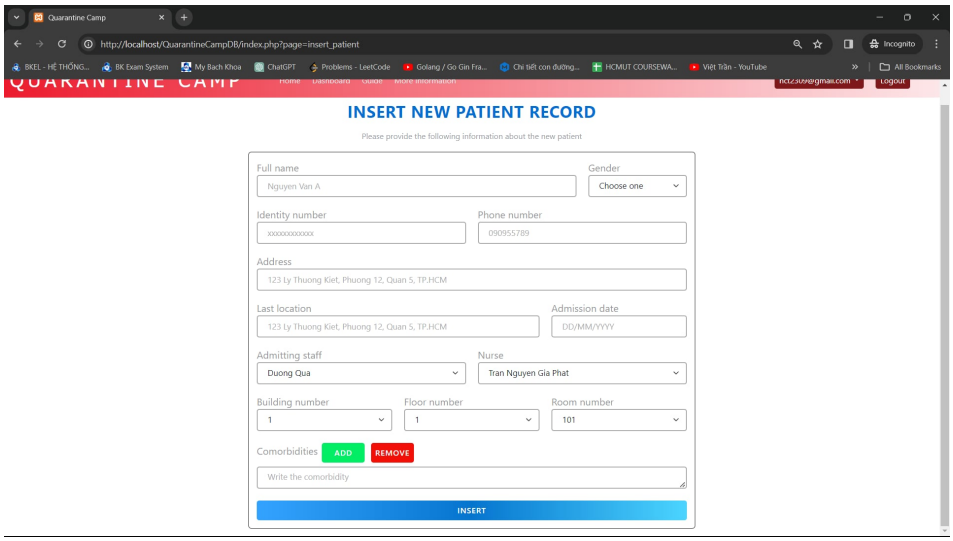

Figure 5. Insert Form

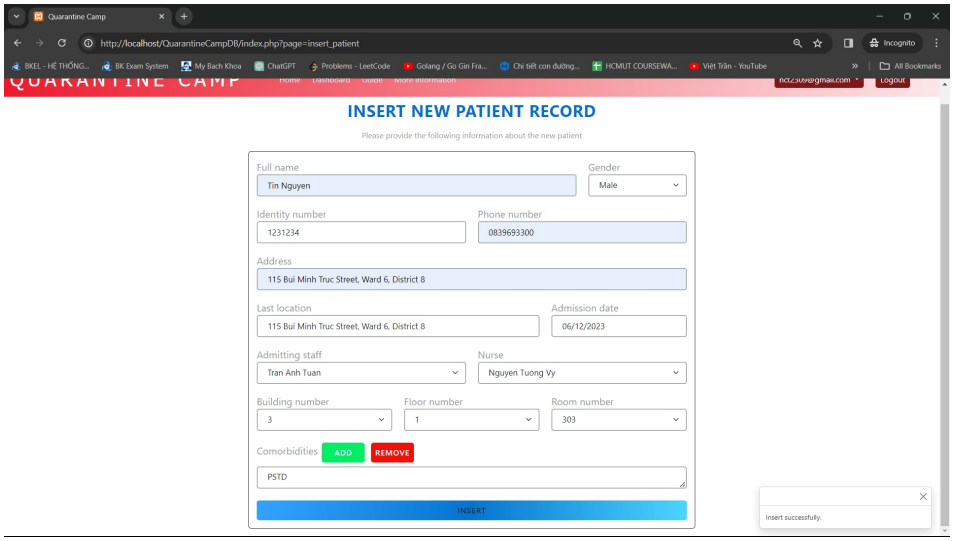

Figure 6. You must input all the fields to insert a patient

3.2.3 List details of all testing which belong to a patient.

The system captures and compiles a comprehensive list of all testing activities associated with a specific patient. This includes an exhaustive record of diagnostic examinations, assessments, and tests that have been conducted for the individual. The details encompass a range of medical evaluations, such as pcr test, quick test, spo test, respiratory test, offering a thorough overview of the patient's medical testing history. This consolidated information not only facilitates efficient tracking of the patient's health assessments but also aids healthcare professionals in making well-informed decisions based on a comprehensive understanding of the individual's diagnostic journey. Access to this detailed testing history is pivotal in ensuring continuity of care, enabling medical practitioners to tailor treatment plans and interventions based on the specific diagnostic data pertinent to each patient.

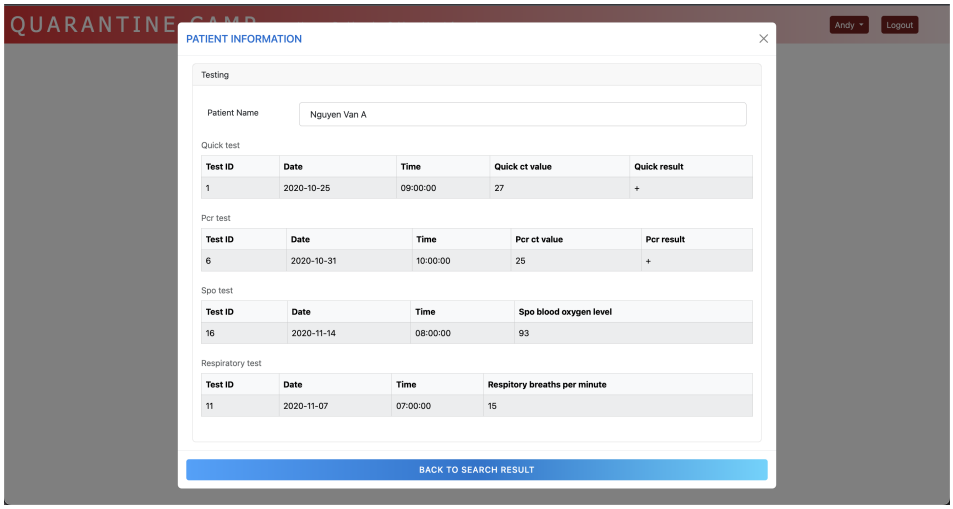

Figure 7. Click button VIEW TESTING to view the testing of specific patient

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

The comprehensive patient report offers a detailed overview of an individual's health journey, encompassing vital categories of information. It begins with demographic details, encapsulating personal information such as the full name, contact number, admission date, identity number, and gender, establishing a foundational understanding of the patient's identity. Moving beyond demographics, the report delves into the realm of comorbidities, shedding light on any pre-existing health conditions that may impact the current medical status. The inclusion of symptoms provides insight into the specific manifestations the patient is experiencing, aiding in a more nuanced diagnosis. Furthermore, the report incorporates a thorough account of testing activities conducted, spanning various diagnostic assessments and examinations. Lastly, details about the ongoing or completed treatment plans contribute to a holistic understanding of the patient's medical history. This multifaceted approach ensures that healthcare professionals have a comprehensive and consolidated view of the patient, fostering more informed decision-making and personalized care interventions.

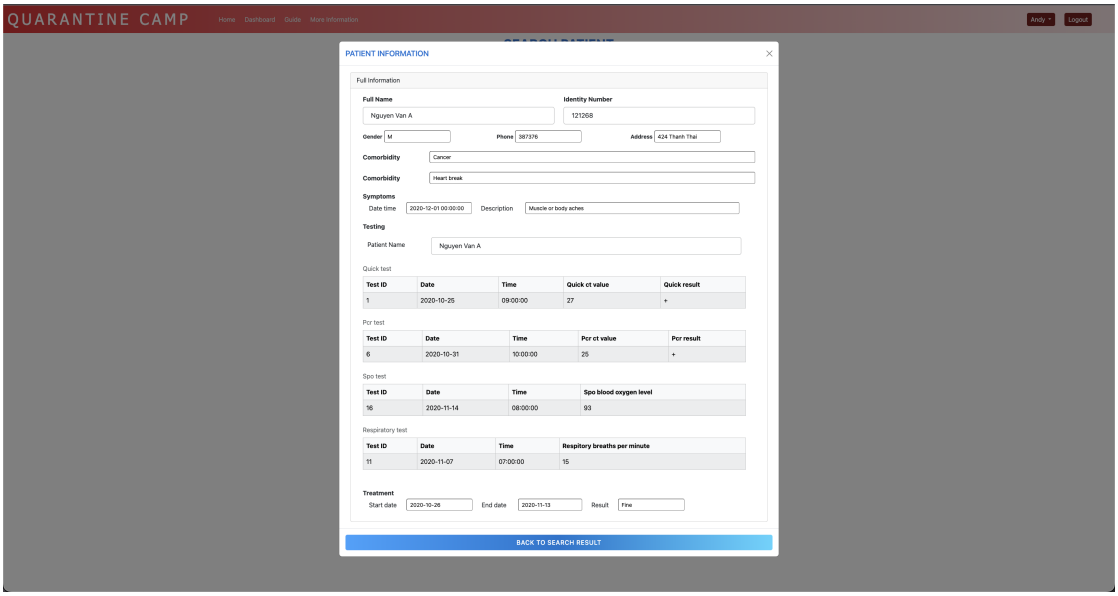

Figure 8. Click button VIEW FULL to view the full information of specific patient

4 DATABASE MANAGEMENT

4.1 Proving one use-case of indexing efficiency in your scenarios

```
-- Generate and insert a million rows into the patient table
CREATE SEQUENCE public.identity_number_seq;

-- Create a sequence to generate unique numeric values for the phone
CREATE SEQUENCE public.phone_seq;

-- Generate and insert a million rows into the patient table
INSERT INTO public.patient (identity_number, fullname, phone, gender, address,
    discharge_date)
SELECT
    LPAD(NEXTVAL('public.identity_number_seq')::text, 9, '0'), -- Ensure 9 digits
    for identity_number
    'Name' || gs,
    LPAD(NEXTVAL('public.phone_seq')::text, 10, '0'), -- Ensure 10 digits for
    phone
    CASE WHEN gs % 2 = 0 THEN 'M' ELSE 'F' END,
    'Address' || gs,
    current_date - (random() * 365)::integer
FROM generate_series(1, 50000000) gs;
```

Generate 50 million records for table Patient

Consider a scenario where the number of records in our quarantine camp's Patient table has surged to **50 million**, and we aim to retrieve information about a specific patient, let's say 'Nguyen Van A.' Unfortunately, the column 'fullname' lacks an index and is not designated as a primary key in this table. Consequently, the Database Management System (DBMS) is compelled to sequentially search for the patient named 'Nguyen Van A,' leading to an execution time that grows proportionally with the number of records. This linear growth in execution time is evident in Figure 9, where the query takes approximately **28616 milliseconds**—an especially sluggish performance, particularly when involving joins with other tables.

```
EXPLAIN ANALYZE SELECT * FROM patient WHERE fullname = 'Name1000000';
```

The query to analyze

	QUERY PLAN text
1	Gather (cost=1000.00..639349.82 rows=62384 width=334) (actual time=159.689..28615.882 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on patient (cost=0.00..632111.42 rows=25993 width=334) (actual time=19020.351..28489.640 rows=0 loop…
5	Filter: ((fullname)::text = 'Name1000000'::text)
6	Rows Removed by Filter: 16666666
7	Planning Time: 0.576 ms
8	JIT:
9	Functions: 6
10	Options: Inlining true, Optimization true, Expressions true, Deforming true
11	Timing: Generation 1.566 ms, Inlining 135.175 ms, Optimization 83.746 ms, Emission 47.179 ms, Total 267.665 ms
12	Execution Time: 28616.491 ms

Figure 9. Query without indexing

Subsequently, upon creating an index for the 'fullname' column, the same query exhibits a significantly enhanced execution time, now only requiring around **2.29 milliseconds** (excluding caching), as shown in Figure 10. This substantial improvement highlights the efficiency gained through the judicious use of indexing, particularly in scenarios involving extensive database and complex query operations.

```
CREATE INDEX IF NOT EXISTS patient_name_index ON patient(fullname);
EXPLAIN ANALYZE SELECT * FROM patient WHERE fullname = 'Name1000000';
```

Create index for column fullname of table patient

	QUERY PLAN text
1	Index Scan using patient_name_index on patient (cost=0.56..8.58 rows=1 width=58) (actual time=1.698..1.706 rows=1 loops=1)
2	Index Cond: ((fullname)::text = 'Name1000000'::text)
3	Planning Time: 5.514 ms
4	Execution Time: 2.290 ms

Figure 10. Query with indexing

4.2 Solving one use-case of database security in your scenarios

Security is a paramount concern in the development and maintenance of any database and website. In this project, the focus is on safeguarding the integrity of both the code and the entire website by adopting principles from the CIA triad: confidentiality, integrity, and availability.

4.2.1 Theory of general security

Confidentiality is a critical aspect of the security framework, emphasizing the protection of data from unauthorized access. The project recognizes the significance of this principle,

particularly in light of regulatory requirements such as HIPAA (Health Insurance Portability and Accountability Act). With patient information being a core component of the database, ensuring that sensitive data is securely stored and remains inaccessible to unauthorized parties is a top priority.

Integrity is another foundational element of the security approach, focusing on the prevention of unintended or unauthorized modifications to the data within the database. The project acknowledges the importance of maintaining data accuracy and reliability, especially in environments where the integrity of information is paramount. Unwanted alterations, whether intentional or accidental, pose a threat to the trustworthiness of the entire system.

Ensuring the continuous availability of both the server and the data is a key objective in the project's security strategy. The availability component of the CIA triad underscores the importance of making resources accessible whenever they are needed. By prioritizing availability, the project aims to minimize downtime and ensure that users can reliably access the website and its associated data at all times.

As part of the project's security demonstration, injection attack methods will be explored. This exploration serves as a means to illustrate potential vulnerabilities and compromises that a website might encounter. By proactively addressing these issues, the project aims to enhance its resilience against various security threats, ultimately providing a robust and secure environment for both data and users. The focus on the CIA triad establishes a solid foundation for comprehensive security measures, although additional considerations may also be incorporated to further fortify the website's defenses against evolving cyber threats.

4.2.2 SQL Injection

SQL Injection (SQLi) is a malicious technique exploiting vulnerabilities in web application databases. It occurs when unfiltered user inputs are improperly handled, allowing attackers to insert or manipulate SQL queries. Understanding SQL Injection involves recognizing the potential risks associated with insufficient input validation and the need to fortify defenses against this pervasive threat. **Genaral types of SQLi**

In-band SQLi: Attacker uses the same communication channel for launching and gathering results, with two sub-variations:

- Error-based SQLi exploits database errors to gather information about its structure.
- Union-based SQLi leverages the UNION SQL operator to obtain a single HTTP response containing exploitable data.

Inferential (Blind) SQLi: Attacker sends data payloads to observe server responses, learning about its structure without direct data transfer. Two types:

- Boolean Blind SQLi prompts true/false results based on query responses, modifying HTTP content accordingly.
- Time-based Blind SQLi delays the database's reaction, with the attacker deducing query truth from response times.

Out-of-band SQLi: Attack relies on features enabled on the web application's database server, serving as an alternative to in-band and inferential techniques. Used when the same channel can't be employed for both attack and information retrieval or when the server's speed hinders such actions. The technique relies on the server's ability to create DNS or HTTP requests to transfer data to the attacker.

We will use the Boolean SQLi for simple demonstration.

4.2.3 How it works

For the login process, the sql is for this is
"SELECT * FROM public.mgr_authentication WHERE username = '$email' AND password ='$password'"
By using the payload: ok' or 1=1– ' The system will show all of the database as the above query will become: "SELECT * FROM public.mgr_authentication WHERE username ='ok' or 1=1– " AND password = '$password'"

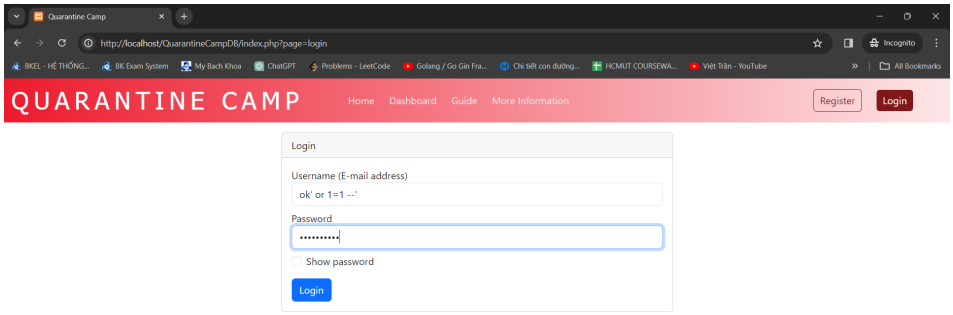

Figure 11. Try to inject sql through the input box.

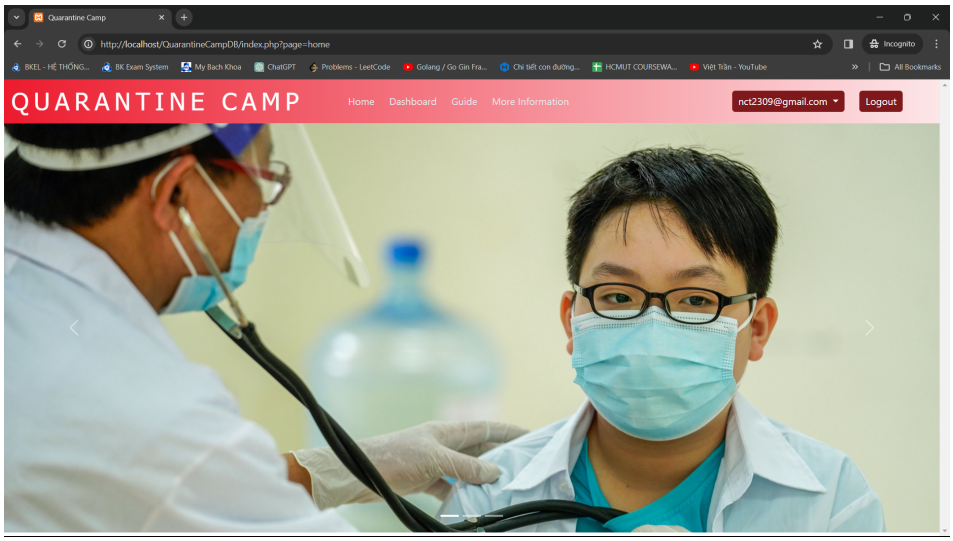

Figure 12. Sql injection successful.

In this query, the system check for 2 condition username Like 'test' or 1 = 1. Because 1 = 1 is true, the system will let the user login even if the password was already hashed.

4.2.4 How to prevent

The 2 ways to prevent Sql injection is input validation and input parameterized.

The input validition can be done both in client and server. There are many to validate by required attribute of "input" tag, write a regex function to validate input. So as in backend.

```
function isEmailValid(email) {
    // Regular expression pattern for a valid email address
    var emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
    return emailPattern.test(email);
}
```

Figure 13. Example for validation.

The next convenient way is to parameterized the input data or use parameterize.

```
$result = pg_query_params($conn, 'SELECT * FROM public.mgr_authentication WHERE username = $1 AND password = $2', array($email, $password));
```

Figure 14. PHP function to bind data into query.

The information provided pertains to a singular security use-case, recognizing the existence of numerous other security challenges. Misconfigurations and the use of inadequate default passwords can potentially give rise to various other issues.