Healthcare Organization Database Project

author: **Dylan Kayyem**

sid: **dyga6971**

github: **dylankayyem**

course: **3287**

Link to Video: https://cuboulder.zoom.us/rec/share/C_TLcr30-BSfq8WHxopJ2FDLfY2gv0bdyabBgZnnpZz7OsoLk-gfvnAHsYOdprSf.gzfTGEpLU4EtBu1g?startTime=1702343187000

Passcode: s#Zu@*7i

Project Description:

Any new Patient is first registered in their database before meeting the doctor. The Doctor can update the data related to the patient upon diagnosis, including the disease diagnosed and prescription. This organization also provides rooms facility for admitting the patient who is critical. Apart from doctors, this organization has nurses and “ward boy”. Each nurse and “ward boy” is assigned to a doctor. They can be assigned to patients, to take care of them. The bill is paid by the patient with two payment options: Cash or E-Banking. A record of each payment made is also maintained by the organization. The record of each call received to provide help and support to its existing person is also maintained.

The healthcare management system is designed to streamline the operations of a medical facility. The SQL database at its core manages intricate relationships between patients, doctors, and administrative staff. This system ensures data integrity, facilitates complex queries, and handles sensitive data with appropriate security measures.

The healthcare industry requires robust data management systems to handle complex and sensitive patient data. Our SQL project introduces a comprehensive database designed to manage patient records, staff assignments, appointments, and financial transactions. The system's flexibility allows for easy expansion and modification to meet evolving healthcare needs.

The database schema is designed to reflect the essential entities of a healthcare system and their interrelations. It consists of tables for patients, doctors, appointments, room assignments, and more, each tailored to store and provide access to specific data types efficiently.

Project Required Items:

1. Multiple Table
2. Relationships between table items (foreign keys)
3. Show SQL statements (and any accompanying code) for all table creation, insertion of initial data, updates, and queries.
4. Table Creation
5. Constraints
6. Indexes
7. Triggers
8. Queries
9. Joins between tables
10. Grouping Results
11. Updates (show triggers being executed)
12. Deleting items that are foreign keys in other tables (show triggers being executed)

Project Objective:

My healthcare organization database is designed to streamline the operations of a medical facility. The SQL database at its core manages intricate relationships between patients, doctors, and administrative staff. This system ensures data integrity, facilitates complex queries, and handles sensitive data with appropriate security measures.

Database Management System (DBMS):

```
phpMyAdmin is the chosen DBMS for its robustness, widespread use,
and support for complex transactions and security features.
```

Database Interface:

```
To interact with the database from the Jupyter Notebook, we'll use
the ‘mysql.connector’ Python module in combination with
‘configparser’ to read database configuration from a file.
```

Data Aggregation and Reporting:

```
Pandas will be used within the Jupyter Notebook for data
aggregation tasks.
```

Demonstration of Concepts:

```
This Jupyter Notebook will serve as a platform to demonstrate
database concepts. Code cells will be used to establish and test
database connections, create tables, insert and query data.
```

In [ ]:

```python
import os
import configparser
mysqlcfg = configparser.ConfigParser()
mysqlcfg.read("../mysql.cfg") # Load the database configuration from the config fil
user, passwd = mysqlcfg['mysql']['user'], mysqlcfg['mysql']['passwd']
dburl = f"mysql://{user}:{passwd}@applied-sql.cs.colorado.edu:3306/{user}"
print (f"mysql://{user}:xxxx@applied-sql.cs.colorado.edu:3306/{user}")
os.environ['DATABASE_URL'] = dburl # define this env. var for sqlmagic
```

```
mysql://dyga6971:xxxx@applied-sql.cs.colorado.edu:3306/dyga6971
```

Load the database configuration from the config file and check it was loaded correctly...

In [ ]:

```python
%reload_ext sql
print ("get version...")
%sql SELECT version()
```

```
get version...
1 rows affected.
```

Out[ ]:

version()
8.0.33

1. Multiple Tables

The healthcare industry requires robust data management systems to handle complex and sensitive patient data. Our SQL project introduces a comprehensive database designed to manage patient records, staff assignments, appointments, and financial transactions. The system's flexibility allows for easy expansion and modification to meet evolving healthcare needs.

The database schema is designed to reflect the essential entities of a healthcare system and their interrelations. It consists of tables for patients, doctors, appointments, room assignments, and care_assignments, call_records, care_provider_table, patient_audit_table, and diagnosis_change_log…

In [ ]:

```
%%sql

DROP TABLE IF EXISTS appointment_table;
DROP TABLE IF EXISTS care_assignment_table;
DROP TABLE IF EXISTS call_record_table;
DROP TABLE IF EXISTS payment_table;
DROP TABLE IF EXISTS room_table;
DROP TABLE IF EXISTS care_provider_table;
DROP TABLE IF EXISTS patient_audit_table;
DROP TABLE IF EXISTS diagnosis_change_log;
DROP TABLE IF EXISTS doctor_table;
DROP TABLE IF EXISTS diagnosis_change_log;
DROP TABLE IF EXISTS patient_table;

# Patient Table:
CREATE TABLE patient_table (
    Patient_ID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Address TEXT,
    Phone VARCHAR(30),
    Email VARCHAR(255) UNIQUE,
    Date_Of_Registration DATE,
    Disease_Diagnosed TEXT,
    Prescription TEXT
);

# Doctor Table:
CREATE TABLE doctor_table (
    Doctor_ID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Phone VARCHAR(30),
    Email VARCHAR(255) UNIQUE,
    Specialization VARCHAR(255),
    CHECK (Specialization IN ('Cardiology', 'Neurology', 'Pediatrics','Oncology','O
    Availability TINYINT
);
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
```

```
Out[ ]: []
```

---

2. Relationships between Table Items (foreign keys) & Table Creation

First, let's take a closer look at the relational structure of our healthcare database, focusing on the importance of foreign keys and their relationships.

The patient_table and doctor_table are foundational tables, each with a unique primary key. These keys are the essential links for connecting data across our database.

The care_provider_table includes a foreign key that references the doctor_table. This means each care provider is associated with a specific doctor, and if a doctor's record is deleted, the corresponding care providers are also automatically removed. This relationship is governed by the 'ON DELETE CASCADE' clause, ensuring no orphan records are left behind.

Similarly, the room_table, payment_table, call_record_table, and appointment_table all have foreign keys linked to the patient_table. The cascade effect applies here too, meaning that if a patient is deleted, all their associated records across these tables will be deleted as well.

The patient_audit_table is slightly different. It's there to keep a log of all deletions, providing a historical record whenever a patient's information is removed from the patient_table.

Lastly, the diagnosis_change_log table tracks any updates to a patient's diagnosis, with a foreign key link back to the patient_table. Like the audit table, it's a way of keeping a historical record, but specifically for changes in patient diagnoses.

In [ ]:

```
%%sql

# Care Provider Table:
CREATE TABLE care_provider_table (
    CareProvider_ID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Phone VARCHAR(30),
    Email VARCHAR(255) UNIQUE,
    Type ENUM('Nurse', 'WardBoy') NOT NULL,
    Doctor_ID INT,
    FOREIGN KEY (Doctor_ID) REFERENCES doctor_table(Doctor_ID) ON DELETE CASCADE
);
# Room Table:
CREATE TABLE room_table (
    Room_Number INT PRIMARY KEY,
    Type VARCHAR(50),
    Date_Of_Status DATE,
    Patient_ID INT,
    FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID) ON DELETE CASCADE
);
# Payment Table:
CREATE TABLE payment_table (
    Payment_ID INT AUTO_INCREMENT PRIMARY KEY,
    Patient_ID INT,
    Amount DECIMAL(10, 2),
    Date_Of_Payment DATE,
    Payment_Method ENUM('Cash', 'E-banking') NOT NULL,
    FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID) ON DELETE CASCADE
);
# Call Record Table:
CREATE TABLE call_record_table (
    Call_ID INT AUTO_INCREMENT PRIMARY KEY,
    Patient_ID INT,
    Caller_Phone VARCHAR(30) NOT NULL,
    Date_Time_Of_Call DATETIME NOT NULL,
    Purpose ENUM('Support', 'Help') NOT NULL,
    Description TEXT,
    FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID) ON DELETE CASCADE
);
# Care Assignment Table:
CREATE TABLE care_assignment_table (
    Care_ID INT AUTO_INCREMENT PRIMARY KEY,
    CareProvider_ID INT,
    Patient_ID INT,
    Start_Date DATE,
    End_Date DATE,
    FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID) ON DELETE CASCADE
    FOREIGN KEY (CareProvider_ID) REFERENCES care_provider_table(CareProvider_ID) O
);
# Appointment Table:
```

```
CREATE TABLE appointment_table (
    Appointment_ID INT AUTO_INCREMENT PRIMARY KEY,
    Patient_ID INT,
    Doctor_ID INT,
    Appointment_DateTime DATETIME NOT NULL,
    Reason TEXT,
    FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID) ON DELETE CASCADE
    FOREIGN KEY (Doctor_ID) REFERENCES doctor_table(Doctor_ID) ON DELETE CASCADE
);
# Patient audit table
CREATE TABLE patient_audit_table (
    Audit_ID INT AUTO_INCREMENT PRIMARY KEY,
    Patient_ID INT,
    Name VARCHAR(255),
    Address TEXT,
    Phone VARCHAR(30),
    Email VARCHAR(255) UNIQUE,
    Date_Of_Registration DATE,
    Disease_Diagnosed TEXT,
    Prescription TEXT,
    Deleted_At DATETIME DEFAULT CURRENT_TIMESTAMP
);
# Diagnosis change log table
CREATE TABLE diagnosis_change_log (
    Log_ID INT AUTO_INCREMENT PRIMARY KEY,
    Patient_ID INT NOT NULL,
    Old_Diagnosis TEXT,
    New_Diagnosis TEXT,
    Change_Date DATETIME,
    FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID) ON DELETE CASCADE
);
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
```

```
Out[ ]: []
```

3. Constraints

Now that we have created the tables, let's dive into the fundamental building blocks of our SQL databases - the constraints.

The PRIMARY KEY constraint of the patient table, for instance, uses an AUTO_INCREMENT PRIMARY KEY, meaning that each new patient gets a unique, automatically incremented identifier, which simplifies record management significantly.

The FOREIGN KEY constraint connects the dots, linking related data across different tables. Take the care provider table, for example, where the Doctor_ID field references the doctor_table. This ensures that every care provider is associated with a valid, existing doctor.

The NOT NULL constraint comes next, enforcing the rule that certain fields must always contain valid data. Fields like a patient's name cannot be left empty, as this information is essential for every record.

Unique values in our database are upheld by the UNIQUE constraint. While the primary key is inherently unique, the UNIQUE constraint allows for other columns, such as an email address, to also maintain uniqueness within the table.

Lastly, we'll examine the FOREIGN KEY constraint equipped with a CASCADE delete action. This powerful feature ensures that when a patient's record is removed from our database, all associated entries, such as their care assignments, are also automatically deleted, maintaining the cleanliness of our data.

1. **PRIMARY KEY Constraint:**

   Ensures each row in a table has a unique identifier. No two rows can have the same primary key value, and a primary key column cannot contain NULL values.

   Example:

   ```
   CREATE TABLE patient_table (
       Patient_ID INT AUTO_INCREMENT PRIMARY KEY,
       -- other columns
   );
   ```

   In this example, Patient_ID is a unique identifier for each patient record in the patient_table.

2. **FOREIGN KEY Constraint:**

   Ensures referential integrity of the data in one table to match values in another table. It constrains data to only allow values that are present in the referenced primary key column of another table.

Example:

```
CREATE TABLE care_provider_table (
    CareProvider_ID INT AUTO_INCREMENT PRIMARY KEY,
    Doctor_ID INT,
    FOREIGN KEY (Doctor_ID) REFERENCES doctor_table(Doctor_ID)
    -- other columns
);
```

3. **NOT NULL Constraint:**

   Specifies that a column cannot store NULL value. This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

   Example:

   ```
   CREATE TABLE patient_table (
       Name VARCHAR(255) NOT NULL,
       -- other columns
   );
   ```

4. **UNIQUE Constraint:**

   Ensures all values in a column are unique. Unlike the primary key, a unique constraint can accept multiple NULL values and it can be imposed on multiple columns.

   Example:

   ```
   CREATE TABLE patient_table (
       Email VARCHAR(255) UNIQUE,
       -- other columns
   );
   ```

5. **Foreign Key Constraint with Cascading Delete:**

   Ensures referential integrity and also specifies what happens to dependent records in the referencing table when a record is deleted from the referenced table.

   Example:

   ```
    CREATE TABLE care_assignment_table (
        -- other columns
        FOREIGN KEY (Patient_ID) REFERENCES patient_table(Patient_ID)
   ON DELETE CASCADE,
        FOREIGN KEY (CareProvider_ID) REFERENCES
   care_provider_table(CareProvider_ID) ON DELETE CASCADE
    );
   ```

   - If a Patient_ID is deleted from the patient_table, all related care assignments in care_assignment_table will also be deleted due to the ON DELETE CASCADE action.

---

4. Show SQL statements (and any accompanying code) for all table creation, insertion of initial data, updates, and queries

At the heart of this operation are our functions designed to generate randomized, yet realistic, data for our 8 database tables. Each table in our database has a dedicated function, specifically crafted to reflect the unique nature of the data it holds.

For example, the generate_patient_data function mimics a mini-universe, populating the patient_table with diverse patient profiles, complete with names, addresses, diagnosis, and prescriptions – all produced by the faker and random library, ensuring that each entry is as unique as the individuals they represent.

Once our data is generated, it's converted into a panda dataframe, and then a csv file. The insertion process is carried out by another set of functions that takes the generated csv file as input, each tailored to the table it serves.

This meticulous process is mirrored for each table in our system, ensuring that our database is not only a repository of information but a dynamic reflection of a living, breathing healthcare ecosystem.

In [ ]:

```
from datetime import datetime
import mysql.connector
import pandas as pd
import random
import faker
import csv
fake = faker.Faker()
```

In [ ]:

```
# Function to create a database connection
def create_db_connection(config_path='../mysql.cfg'):
    mysqlcfg = configparser.ConfigParser()
    mysqlcfg.read(config_path)
    db_config = mysqlcfg['mysql']
    db = mysql.connector.connect(
        host="applied-sql.cs.colorado.edu",
        user=db_config['user'],
        passwd=db_config['passwd'],
        database=db_config['user']
    )
    cursor = db.cursor()
    return db, cursor
```

1. patient_table

In [ ]:

```
# Function to generate patient_data(100 patients):
def generate_patient_data(num_patient):
    diseases = ['Arthritis', 'Epilepsy', 'Anemia', 'Diabetes', 'Allergies', 'Migrai
    prescriptions = ['Levothyroxine', 'Atorvastatin', 'Metroprolol', 'Gabapentin',
    patient_data = []
    for i in range(1, num_patient + 1):
        patient = {
            'Patient_ID': i,
            'Name': fake.name(),
            'Address': fake.address(),
            'Phone': fake.phone_number(),
            'Email': fake.email(),
            'Date_Of_Registration': fake.date_between(start_date="-1y", end_date="t
            'Disease_Diagnosed': random.choice(diseases),
            'Prescription': random.choice(prescriptions)
        }
        patient_data.append(patient)
    print("[patient_data] successfully generated!")
    return patient_data
num_patients = 100
patient_data = generate_patient_data(num_patients)
df_patient = pd.DataFrame(patient_data, columns=["Patient_ID", "Name", "Address", "
csv_file_path_draft = '../Project/sample_data/patient_data.csv'
df_patient.to_csv(csv_file_path_draft, index=False)
csv_file_path_draft
```

```
[patient_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/patient_data.csv'
```

In [ ]:

```
# Insert patient_data into patient_table:
db, cursor = create_db_connection("../mysql.cfg")
csv_file_path = '../Project/sample_data/patient_data.csv'
try:
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute(
                "INSERT INTO patient_table (Name, Address, Phone, Email, Date_Of_Re
                row[1:]
            )
    db.commit()
    print("[patient_data] successfully inserted into [patient_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    cursor.close()
    db.close()
```

```
[patient_data] successfully inserted into [patient_table]!
```

2. doctor_table

In [ ]:

```
# Generating random data for doctor_table(20 doctors):
def generate_doctor_data(num_doctors):
    doctor_data = []
    specializations = ['Cardiology', 'Neurology', 'Pediatrics', 'Orthopedics', 'Onc
    for i in range(1, num_doctors+1):
        doctor = {
            'Doctor_ID': i,
            'Name': fake.name(),
            'Phone': fake.phone_number(),
            'Email': fake.email(),
            'Specialization': random.choice(specializations),
            'Availability': random.randint(0, 1)
        }
        doctor_data.append(doctor)
    print("[doctor_data] successfully generated!")
    return doctor_data
num_doctors = 20
doctor_data = generate_doctor_data(num_doctors)
df_doctors = pd.DataFrame(doctor_data, columns=["Doctor_ID", "Name", "Phone", "Emai
csv_doctor_file_path = '../Project/sample_data/doctor_data.csv'
df_doctors.to_csv(csv_doctor_file_path, index=False)
csv_doctor_file_path
```

```
[doctor_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/doctor_data.csv'
```

In [ ]:

```
# Insert doctor_data into doctor_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/doctor_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute(
                "INSERT INTO doctor_table (Name, Phone, Email, Specialization, Avai
                row[1:]
            )
    db.commit()
    print("[doctor_data] successfully inserted into [doctor_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    cursor.close()
    db.close()
```

```
[doctor_data] successfully inserted into [doctor_table]!
```

3. care_provider_table

In [ ]:

```
# Function to generate care_provider_data(100 care providers):
def generate_care_provider_data(num_care_provider):
    db, cursor = create_db_connection("../mysql.cfg")
    care_providers = []
    # doctor_ids is set to doctor IDs from table:
    cursor.execute("SELECT Doctor_ID FROM doctor_table")
    doctor_ids = [doctor_id[0] for doctor_id in cursor.fetchall()]
    for i in range(1, num_care_provider + 1):
        care_provider = {
            'CareProvider_ID': i,
            'Name': fake.name(),
            'Phone':  fake.phone_number(),
            'Email': fake.email(),
            'Type': random.choice(['Nurse','WardBoy']),
            'Doctor_ID': random.choice(doctor_ids),
        }
        care_providers.append(care_provider)
    print("[care_provider_data] successfully generated!")
    return care_providers
num_care_provider = 100
care_provider_data = generate_care_provider_data(num_care_provider)
df_care_provider = pd.DataFrame(care_provider_data, columns=["CareProvider_ID", "Na
csv_nurse_file_path = '../Project/sample_data/care_provider_data.csv'
df_care_provider.to_csv(csv_nurse_file_path, index=False)
csv_nurse_file_path
#df_care_provider.head(5)
```

```
[care_provider_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/care_provider_data.csv'
```

In [ ]:

```
# Insert care_provider_data into care_provider_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/care_provider_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute("""
                INSERT INTO care_provider_table (CareProvider_ID, Name, Phone, Emai
                VALUES (%s, %s, %s, %s, %s, %s)
                """, row)
    db.commit()
    print("[care_provider_data] successfully inserted into [care_provider_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    cursor.close()
    db.close()
```

```
[care_provider_data] successfully inserted into [care_provider_table]!
```

4. care_assignment_table

In [ ]:

```python
# Function to generate care_assignment_data(100 care assignments):
def generate_care_assignment_data(num_assignment):
    db, cursor = create_db_connection("../mysql.cfg")
    care_assignments = []
    cursor.execute("SELECT Patient_ID FROM patient_table")
    patients_id = [patient_id[0] for patient_id in cursor.fetchall()]
    cursor.execute("SELECT CareProvider_ID FROM care_provider_table")
    careproviders_ids = [careprovider_id[0] for careprovider_id in cursor.fetchall(
    for i in range(1, num_assignment + 1):
        care_assignment = {
            'Care_ID': i,
            'CareProvider_ID': random.choice(careproviders_ids),
            'Patient_ID': random.choice(patients_id),
            'Start_Date': fake.date_between(start_date="-1y", end_date="today"),
            'End_Date': fake.date_between(start_date="today", end_date="+1y"),
        }
        care_assignments.append(care_assignment)
    print("[care_assignment_data] successfully generated!")
    return care_assignments
num_assignments = 100
care_assignment_data = generate_care_assignment_data(num_assignments)
df_care_assignments = pd.DataFrame(care_assignment_data, columns=['Care_ID', 'CareP
csv_care_assignment_file_path = '../Project/sample_data/care_assignment_data.csv'
df_care_assignments.to_csv(csv_care_assignment_file_path, index=False)
csv_care_assignment_file_path
```

```
[care_assignment_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/care_assignment_data.csv'
```

In [ ]:

```python
# Insert care_assignment_data into care_assignment_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/care_assignment_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute("""
                INSERT INTO care_assignment_table (Care_ID, CareProvider_ID, Patien
                VALUES (%s, %s, %s, %s, %s)
                """, row)
    db.commit()
    print("[care_assignment_data] successfully inserted into [care_assignment_table
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    cursor.close()
    db.close()
```

```
[care_assignment_data] successfully inserted into [care_assignment_table]!
```

5. room_table

In [ ]:

```python
# Function to generate room_data(50 rooms)
def generate_room_data(num_rooms):
    room_types = ['Single', 'Double', 'Deluxe', 'Suite']
    rooms = []
    for i in range(1, num_rooms + 1):
        room_number = i
        room_type = random.choice(room_types)
        patient_id = random.choice([1, random.randint(1, 100)])
        date_of_status = fake.date_between(start_date="-1y", end_date="today")
        rooms.append({
            'Room_Number': room_number,
            'Type': room_type,
            'Date_Of_Status': date_of_status,
            'Patient_ID': patient_id
        })
    print("[room_data] successfully generated!")
    return rooms
num_rooms = 50
room_data = generate_room_data(num_rooms)
df_room = pd.DataFrame(room_data, columns=["Room_Number", "Type", "Date_Of_Status",
csv_room_file_path = '../Project/sample_data/room_data.csv'
df_room.to_csv(csv_room_file_path, index=False)
csv_room_file_path
```

```
[room_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/room_data.csv'
```

In [ ]:

```python
# Insert room_data into room_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/room_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute("""
                INSERT INTO room_table (Room_Number, Type, Date_Of_Status, Patient_
                VALUES (%s, %s, %s, %s)
                """, row)
    db.commit()
    print("[room_data] successfully inserted into [room_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    if cursor:
        cursor.close()
    if db:
        db.close()
```

```
[room_data] successfully inserted into [room_table]!
```

6. payment_table

In [ ]:

```
# Function to generate payment_data(100 payments and 100 patients):
def generate_payment_data(num_payment, num_patient):
    data = []
    for _ in range(num_payment):
        payment_id = _ + 1
        patient_id = random.randint(1, num_patient)
        amount = round(random.uniform(100, 1000), 2)
        date_of_payment = fake.date_between_dates(date_start=datetime(2020, 1, 1),
        payment_method = random.choice(['Cash', 'E-banking'])
        data.append([payment_id, patient_id, amount, date_of_payment, payment_metho
    print("[payment_data] successfully generated!")
    return data
num_payments = 100
num_patients = 100
payment_data = generate_payment_data(num_payments, num_patients)
df_payment = pd.DataFrame(payment_data, columns=["Payment_ID", "Patient_ID", "Amoun
csv_payment_file_path = '../Project/sample_data/payment_data.csv'
df_payment.to_csv(csv_payment_file_path, index=False)
csv_payment_file_path
```

```
[payment_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/payment_data.csv'
```

In [ ]:

```
# Insert payment_data into payment_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/payment_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute("""
                INSERT INTO payment_table (Patient_ID, Amount, Date_Of_Payment, Pay
                VALUES (%s, %s, %s, %s)
            """, row[1:])
    db.commit()
    print("[payment_data] successfully inserted into [payment_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    if cursor:
        cursor.close()
    if db:
        db.close()
```

```
[payment_data] successfully inserted into [payment_table]!
```

7. call_record_table

In [ ]:

```
# Function to generate call_record_data(100 records for 100 patients):
def generate_call_record_data(num_record, num_patient):
    data = []
    for _ in range(num_record):
        call_id = _ + 1
        patient_id = random.randint(1, num_patient)
        caller_phone = fake.phone_number()
        date_time_of_call = fake.date_time_between(start_date="-2y", end_date="now"
        purpose = random.choice(['Support', 'Help'])
        description = fake.text(max_nb_chars=200)
        data.append([call_id, patient_id, caller_phone, date_time_of_call, purpose,
    print("[call_record_data] successfully generated!")
    return data
num_records = 100
num_patients = 100
call_record_data = generate_call_record_data(num_records, num_patients)
df_call_record = pd.DataFrame(call_record_data, columns=["Call_ID", "Patient_ID", "
csv_call_record_file_path = '../Project/sample_data/call_record_data.csv'
df_call_record.to_csv(csv_call_record_file_path, index=False)
csv_call_record_file_path
```

```
[call_record_data] successfully generated!
```

Out[ ]:

```
'../Project/sample_data/call_record_data.csv'
```

In [ ]:

```
# Insert call_record_data into call_record_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/call_record_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute("""
                INSERT INTO call_record_table (Patient_ID, Caller_Phone, Date_Time_
                VALUES (%s, %s, %s, %s, %s)
            """, row[1:])
    db.commit()
    print("[call_record_data] successfully inserted into [call_record_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    if cursor:
        cursor.close()
    if db:
        db.close()
```

```
[call_record_data] successfully inserted into [call_record_table]!
```

8. appointment_table

In [ ]:

```
# Function to generate appointment_data(250 appointments):
def generate_appointment_data(num_appointment):
    data = []
    num_patient = 100
    num_doctor = 20
    for i in range (1, num_appointment + 1):
        data.append({
            'Appointment_ID': i,
            'Patient_ID': random.randint(1, num_patient),
            'Doctor_ID':random.randint(1, num_doctor),
            'Appointment_DateTime': fake.date_between(start_date="today", end_date=
            'Reason': random.choices(['Checkup', 'Consultation', 'Follow-up', 'Emer
        })
    print("[appointment_data] successfully generated!")
    return data
num_appointments = 250
data = generate_appointment_data(num_appointments)
df_appointment = pd.DataFrame(data, columns=["Appointment_ID", "Patient_ID", "Docto
df_appointment.to_csv('../Project/sample_data/appointment_data.csv', index=False)
```

```
[appointment_data] successfully generated!
```

In [ ]:

```
# Insert appointment_data into appointment_table:
try:
    db, cursor = create_db_connection("../mysql.cfg")
    csv_file_path = '../Project/sample_data/appointment_data.csv'
    with open(csv_file_path, mode='r', encoding='utf-8-sig') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)
        for row in csv_reader:
            cursor.execute("""
                INSERT INTO appointment_table (Patient_ID, Doctor_ID, Appointment_D
                VALUES ( %s, %s, %s, %s)
            """, row[1:])
    db.commit()
    print("[appointment_data] successfully inserted into [appointment_table]!")
except mysql.connector.Error as e:
    print(f"Error: {e}")
    db.rollback()
finally:
    if cursor:
        cursor.close()
    if db:
        db.close()
```

```
[appointment_data] successfully inserted into [appointment_table]!
```

Printing Tables After Creating Tables & Inserting Data

> Once we run the functions and have inserted the data, we can see the randomized tables as follows.
>
> The head is set to 5 just to show what the tables look like populated...

```
In [ ]: df_patient.head(5)
```

Out[ ]:

	Patient_ID	Name	Address	Phone	Email	Date_Of_Regis
0	1	Eric Gibson	1034 Crystal Squares Apt. 544\nLake Cindyfort,...	742-596-9950x0900	christinekramer@example.net	202
1	2	Sherri Oneal	353 Susan Key Suite 421\nPotterberg, GU 38658	001-224-560-3250x23307	garciaseth@example.org	202
2	3	Jennifer Flores	4953 Smith Ramp Suite 149\nSouth Richard, MS 3...	412-346-5620	riveraashley@example.net	202
3	4	Michael Hernandez	761 Perez Circle Apt. 691\nCrystalbury, KS 47600	(859)843-5317x211	ernest31@example.com	202
4	5	Rodney Ramirez	0025 Jones Cliffs Suite 684\nNew Matthewboroug...	+1-876-341-2466	spencertimothy@example.com	202

```
In [ ]: df_doctors.head(5)
```

Out[ ]:

	Doctor_ID	Name	Phone	Email	Specialization	Availability
0	1	Heather Valdez	(658)870-2533	heidicooper@example.com	Neurology	0
1	2	Larry Fisher	(676)833-7413	john20@example.net	Cardiology	0
2	3	Julie Villegas	(657)469-7093	chunt@example.org	Orthopedics	1
3	4	Richard Lara	001-249-573-1320	christopherhenson@example.org	Cardiology	1
4	5	Gabriela Gonzalez	236.926.2292	zramirez@example.org	Neurology	0

```
In [ ]: df_care_provider.head(5)
```

Out[ ]:

	CareProvider_ID	Name	Phone	Email	Type	Doctor_ID
0	1	Andres Hill	(469)828-3583	briana22@example.com	Nurse	11
1	2	Leslie Thomas	001-620-584-7867x07643	rodriguezrobert@example.org	WardBoy	3
2	3	Joseph Kirk	+1-537-700-9262x0076	alisonjenkins@example.org	WardBoy	11
3	4	Kim Blackwell	+1-200-331-5262x8525	timothy71@example.com	Nurse	11
4	5	Ashley Moore	+1-906-840-1497x6632	pmann@example.net	WardBoy	5

In [ ]:

```
df_care_assignments.head(5)
```

Out[ ]:

	Care_ID	CareProvider_ID	Patient_ID	Start_Date	End_Date
0	1	73	36	2023-12-03	2024-10-31
1	2	62	87	2022-12-22	2024-01-10
2	3	91	54	2023-07-13	2024-03-30
3	4	85	5	2023-03-08	2023-12-12
4	5	98	93	2023-06-11	2023-12-16

In [ ]:

```
df_room.head(5)
```

Out[ ]:

	Room_Number	Type	Date_Of_Status	Patient_ID
0	1	Deluxe	2023-10-21	2
1	2	Single	2023-07-01	40
2	3	Double	2023-09-28	1
3	4	Suite	2023-10-26	1
4	5	Suite	2023-09-21	45

In [ ]:

```
df_payment.head(5)
```

Out[ ]:

	Payment_ID	Patient_ID	Amount	Date_Of_Payment	Payment_Method
0	1	4	312.26	2020-05-04	E-banking
1	2	43	349.18	2022-01-23	E-banking
2	3	74	672.49	2020-02-07	E-banking
3	4	11	747.60	2020-09-10	Cash
4	5	98	759.81	2021-11-17	E-banking

In [ ]:

```
df_call_record.head(5)
```

Out[ ]:

	Call_ID	Patient_ID	Caller_Phone	Date_Time_Of_Call	Purpose	Description
0	1	55	909.670.0030x971	2023-09-17 07:14:40	Help	Talk property several dog citizen sure south e...
1	2	96	001-893-636-1715	2023-11-18 12:02:31	Support	Next according drop study from. Officer but mi...
2	3	20	001-706-705-7361	2022-11-09 15:55:36	Support	Think production thus participant actually ind...
3	4	20	+1-862-829-5950	2022-10-10 01:19:21	Help	Law opportunity consumer.\nView at artist impa...
4	5	67	919-514-0239x272	2023-02-04 14:47:57	Help	Want sound else beat choice for six. Stop as s...

In [ ]:

```
df_appointment.head(5)
```

Out[ ]:

	Appointment_ID	Patient_ID	Doctor_ID	Appointment_DateTime	Reason
0	1	44	9	2024-07-27	[Follow-up]
1	2	74	14	2024-02-26	[Consultation]
2	3	42	9	2024-08-09	[Emergency]
3	4	52	4	2024-08-27	[Consultation]
4	5	55	16	2024-09-12	[Emergency]

---

5. Indexes

As we dive into the topic of indexing in SQL databases, it's essential to understand that indexes are a critical component for optimizing performance.

Let's begin with the patient_table. We have an index created on the Name column, which is often a target for WHERE clause searches and JOIN operations. The underlying B-tree structure of this index allows for logarithmic time complexity in search operations, as opposed to a linear search through potentially millions of rows. It's this logarithmic efficiency that drastically reduces the time it takes to locate a patient by name.

Switching our focus to the doctor_table, we applied the same indexing strategy to the Name column. For a database that regularly processes queries for doctor information, this index isn't just a convenience—it's a necessity. It mitigates the overhead associated with table scans and, when paired with join operations that leverage the doctor's name, it ensures that our joins are using the most efficient path to retrieve related data.

When it comes to the care_provider_table, the indexing of the Name column serves a dual purpose. It accelerates direct searches for care providers and enhances the performance of complex queries where the care provider's name is a part of a filter or join condition. It's especially helpful for nested loop joins, where the indexed column can significantly speed up the nested iterations.

1. Index on patient_table(Name):

An index on the Name column of the patient_table helps to speed up queries that search for patients by name. Since names are often used in WHERE clauses or as part of a JOIN condition, this index can greatly reduce the search time by allowing the database to quickly locate rows by name instead of scanning the entire table. If the Name column is used frequently to retrieve patient information, this index can be particularly valuable.

In [ ]:

```
%%sql

CREATE INDEX idx_patient_name ON patient_table(Name);
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]:

```
[]
```

2. Index on doctor_table(Name):

Similar to the patient name index, an index on the Name column of the doctor_table optimizes searches for doctors by name. If you frequently run reports or queries that involve looking up doctors by their names, such an index would help improve the performance of those queries. Additionally, if there are often joins between the doctor_table and other tables using the doctor's name, this index can also expedite those operations.

In [ ]:

```
%%sql

CREATE INDEX idx_doctor_name ON doctor_table(Name);
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]: []

3. Index on care_provider_table(Name):

An index on the Name column of the care_provider_table would enhance search operations for care providers. It would be particularly useful in queries where care providers need to be matched with other entities in the database, such as patients or doctors, especially if those queries filter or join on the care provider's name.

In [ ]:

```
%%sql

CREATE INDEX idx_care_provider_name ON care_provider_table(Name);
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]: []

---

6. Triggers

Let's take a closer look at the triggers we've implemented in our healthcare database:

First, we have a trigger for the patient_table. When a new patient record is being inserted, it's crucial that we capture the date of registration. Our trigger, set_registration_date, does just that. It checks if the Date_Of_Registration field is NULL, and if so, it automatically sets it to the current date using the CURDATE() function. This ensures that every patient record has a registration date without relying on the user to provide one.

Next, we have the trigger for the doctor_table, designed to update a doctor's availability. In a real-world scenario, this trigger would be more complex, but our simple example, after_appointment_insert, ensures that when a new appointment is scheduled, the associated doctor's availability is set to '0' or unavailable. This is a simple but effective way to prevent double-booking of appointments.

Lastly, there's the trigger for the room_table, before_room_insert, which automatically sets the Date_Of_Status to the current date when a patient is assigned to a room. This field is vital as it tracks when rooms become occupied, which in turn helps in managing room availability. With these triggers, we are effectively making our database smarter and more responsive to the dynamic environment of healthcare management.

1. Trigger to set the Date_Of_Registration for a new patient:

In [ ]:

```
%%sql

CREATE TRIGGER set_registration_date
BEFORE INSERT ON patient_table
FOR EACH ROW
BEGIN
   IF NEW.Date_Of_Registration IS NULL THEN
      SET NEW.Date_Of_Registration = CURDATE();
   END IF;
END;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]: []

2. Trigger for Auto-Updating Doctor Availability:

This trigger could automatically update a doctor's availability status to '0' (unavailable) if they have an appointment scheduled. This is a simple example and in a real-world scenario, you would likely have a more complex logic to determine availability.

In [ ]:

```
%%sql

CREATE TRIGGER after_appointment_insert
AFTER INSERT ON appointment_table
FOR EACH ROW
BEGIN
    UPDATE doctor_table SET Availability = 0 WHERE Doctor_ID = NEW.Doctor_ID;
END;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]: []

3. Trigger for Auto-Setting Room Status:

When a patient is assigned to a room, this trigger could automatically set the Date_Of_Status to the current date to indicate when the room was occupied.

In [ ]:

```
%%sql
```

```
CREATE TRIGGER before_room_insert
BEFORE INSERT ON room_table
FOR EACH ROW
BEGIN
    IF NEW.Patient_ID IS NOT NULL THEN
        SET NEW.Date_Of_Status = CURDATE();
    END IF;
END;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

```
Out[ ]: []
```

7. Queries

Let's walk through a series of queries…

Firstly, consider the scenario where we need to identify all patients diagnosed with a specific disease, such as Diabetes. Our SQL query here leverages the WHERE clause to filter patient records. This precise targeting is crucial for healthcare providers to quickly find and manage patient cases, especially when dealing with prevalent conditions.

Moving on, we address the financial aspect of healthcare management. A query designed to select patients with high payment amounts serves as a critical tool for financial oversight. It not only aggregates payments above a certain threshold but also joins patient information, providing a holistic view of high-cost treatments which can be helpful in financial planning and audits.

Next, we have a query that supports the operational logistics of the hospital. By retrieving a list of all doctors with a specific specialization who are currently available, we facilitate efficient scheduling. This query not only filters by specialization but also ensures that only those who are available are considered, optimizing resource allocation.

Lastly, our attention turns to the management of hospital resources, such as room occupancy. We execute a query that lists all occupied rooms, along with patient information. This is vital for day-to-day operations, as it provides a snapshot of room usage, assisting in both patient coordination and room allocation strategies.

1. Query to select all patients diagnosed with a certain disease:

In [ ]:

```
%%sql

SELECT * FROM patient_table WHERE Disease_Diagnosed = 'Diabetes'

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Patient_ID	Name	Address	Phone	Email	Date_Of_Registrat
12	Kevin Martin	PSC 6892, Box 9914 APO AA 73225	579.487.8608x6827	muellertiffany@example.org	2023-05
19	Amber Johnson	Unit 9150 Box 2265 DPO AA 71751	287-822-1105x512	fvelazquez@example.org	2023-07
23	Julian Tucker	0820 Pennington Avenue Suite 364 New Megan, OK 98672	(577)266-3821x6361	emily92@example.com	2023-07
25	Lisa Johnson	4473 Smith Forest Suite 270 Calvinfort, IA 60891	356.328.8845x30720	brad64@example.com	2023-03
43	David Johnson	8269 James Meadow North Karenburgh, VT 05615	888-309-6090x574	gallagherrobert@example.org	2023-10

2. Query to Select Patients with High Payment Amounts:

```
This query selects all patients who have made payments above a
certain threshold, which might be useful for identifying high-cost
treatments or prioritizing financial audits.
```

In [ ]:

```
%%sql

SELECT p.Patient_ID, p.Name, p.Address, p.Phone, p.Email, p.Date_Of_Registration, p
FROM patient_table p
JOIN payment_table pt ON p.Patient_ID = pt.Patient_ID
```

```
GROUP BY p.Patient_ID
HAVING Total_Paid > 2000;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
4 rows affected.
```

Out[ ]:

Patient_ID	Name	Address	Phone	Email	Date_Of_Regist
70	Vanessa Herrera	989 Roberto Court Suite 782 Guerreroborough, TX 85819	+1-254-370-2077x962	hayesrobert@example.com	2023
88	Michael Zuniga	3146 Katrina Estate Apt. 593 Stephaniefort, VI 63572	335.736.4967x12790	ncurtis@example.com	2023
24	Tyler Richards	9227 Petty Gateway Jacobsmouth, CA 40398	634-431-1614x589	anitakeller@example.net	2023
48	Ricky Bennett	77235 Sharon Heights Apt. 445 New Joycemouth, WI 63212	506.630.1860x3306	yrogers@example.net	2023

3. Query to Find Doctors with Specific Specialization Available:

```
This query can be used to find all doctors who specialize in a
particular field and are currently available, which can be useful
for scheduling purposes.
```

In [ ]:

```
%%sql

SELECT Doctor_ID, Name, Phone, Email, Specialization
FROM doctor_table
WHERE Specialization = 'Cardiology' AND Availability = 0;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
2 rows affected.
```

Out[ ]:

Doctor_ID	Name	Phone	Email	Specialization
2	Larry Fisher	(676)833-7413	john20@example.net	Cardiology
14	Maria Lucas	+1-684-485-7237x18292	nicole13@example.net	Cardiology

4. Query to List All Rooms Currently Occupied:

```
This query gives a list of all rooms that are currently occupied by
patients, providing information on patient allocation within the
hospital.
```

In [ ]:

```
%%sql

SELECT r.Room_Number, r.Type, r.Date_Of_Status, p.Name as Patient_Name
FROM room_table r
JOIN patient_table p ON r.Patient_ID = p.Patient_ID
WHERE r.Patient_ID IS NOT NULL

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Room_Number	Type	Date_Of_Status	Patient_Name
3	Double	2023-09-28	Eric Gibson
4	Suite	2023-10-26	Eric Gibson
7	Suite	2023-10-19	Eric Gibson
8	Double	2023-11-24	Eric Gibson
9	Double	2023-06-22	Eric Gibson

---

8. Using Joins between tables

```
Next, we have the joins. We start with the fundamental join between
our patient_table and care_provider_table. This join is essential
for accessing comprehensive patient care data. Through an INNER
JOIN with the care_assignment_table, we retrieve a list that
connects patient names with their assigned care providers, painting
a full picture of patient care assignments.

Next, we examine how a join between the patient_table and
appointment_table can be utilized to fetch upcoming appointments.
This is particularly useful for front desk operations and patient
notifications, ensuring that everyone is informed of future
engagements. By using a WHERE clause to filter appointments after
the current date, we provide a proactive view into the patient's
schedule.

Then, we shift our focus to the operational side where doctors and
care providers intersect. By joining the doctor_table and
care_provider_table, we list all care providers assigned to each
doctor. This information streamlines the management of staff and
helps in the efficient delegation of tasks.

Finally, we delve into a more complex join that brings together the
patient_table, payment_table, and call_record_table. This multi-
```

table join is not just about linking tables; it's about consolidating a patient's financial and interaction records into one view. It gives us insight into a patient's latest payment and their most recent communication with the facility, information that's vital for both administrative staff and healthcare providers.

1. Joins between patient_table and care_provider_table to get the patient details along with their assigned care provider:

In [ ]:

```
%%sql

SELECT p.Name AS Patient_Name, c.Name AS CareProvider_Name
FROM patient_table p
JOIN care_assignment_table ca ON p.Patient_ID = ca.Patient_ID
JOIN care_provider_table c ON ca.CareProvider_ID = c.CareProvider_ID

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Patient_Name	CareProvider_Name
Kelly Blevins	Frank Keller
Lisa Trevino	Joe Roberts
Jordan Davis	Jeremy Gross
Rodney Ramirez	Bruce White
Tammy Perez	Eric Taylor

2. Join between patient_table and appointment_table to get upcoming appointments for patients:

In [ ]:

```
%%sql

SELECT p.Name AS Patient_Name, d.Name AS Doctor_Name, a.Appointment_DateTime
FROM patient_table p
JOIN appointment_table a ON p.Patient_ID = a.Patient_ID
JOIN doctor_table d ON a.Doctor_ID = d.Doctor_ID
WHERE a.Appointment_DateTime > NOW()

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Patient_Name	Doctor_Name	Appointment_DateTime
Wesley Huang	Allison Weaver	2024-08-10 00:00:00
Tammie Swanson	Allison Weaver	2024-04-14 00:00:00
Misty Smith	Allison Weaver	2024-09-16 00:00:00
Mr. Donald Lloyd	Allison Weaver	2024-11-08 00:00:00
Andrea Lambert	Allison Weaver	2024-04-16 00:00:00

3. Join between doctor_table and care_provider_table to list all care providers assigned to each doctor:

In [ ]:

```
%%sql

SELECT d.Name AS Doctor_Name, cp.Name AS CareProvider_Name, cp.Type
FROM doctor_table d
JOIN care_provider_table cp ON d.Doctor_ID = cp.Doctor_ID

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Doctor_Name	CareProvider_Name	Type
Allison Weaver	Allison Rodriguez	WardBoy
Allison Weaver	Brian Mitchell	Nurse
Allison Weaver	John Underwood	Nurse
Allison Weaver	Rebecca Rodriguez	WardBoy
Brian Jordan	Shawn Holder	WardBoy

4. Join between patient_table, payment_table, and call_record_table to find payment details and last call record for each patient:

In [ ]:

```
%%sql

SELECT p.Name AS Patient_Name, pt.Amount AS Payment_Amount, pt.Date_Of_Payment, cr.
FROM patient_table p
JOIN payment_table pt ON p.Patient_ID = pt.Patient_ID
LEFT JOIN call_record_table cr ON p.Patient_ID = cr.Patient_ID
WHERE pt.Date_Of_Payment = (SELECT MAX(Date_Of_Payment) FROM payment_table WHERE Pa
AND cr.Date_Time_Of_Call = (SELECT MAX(Date_Time_Of_Call) FROM call_record_table WH

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Patient_Name	Payment_Amount	Date_Of_Payment	Caller_Phone	Date_Time_Of_Call
Christopher Santiago	590.65	2021-07-27	909.670.0030x971	2023-09-17 07:14:40
Luis Hubbard	733.83	2020-07-12	001-893-636-1715	2023-11-18 12:02:31
Amy Brown	796.86	2021-05-03	001-706-705-7361	2022-11-09 15:55:36
Michael Howell	611.16	2020-05-13	919-514-0239x272	2023-02-04 14:47:57
Peggy Ferguson DDS	279.75	2021-05-16	349-588-4359x46821	2022-09-27 22:43:09

9. Grouping Results

In the healthcare sector, the ability to group and summarize data is invaluable.

Let’s look at how we can group patients by their attending doctors. The SQL query we use here employs the GROUP BY clause to aggregate patient data by doctor. This not only helps us assess the workload of each doctor but also enables us to manage resources effectively, ensuring that patients receive timely care.

Similarly, when analyzing care assignments, we can group these by each care provider. By aggregating the number of care assignments, we gain clarity on the distribution of tasks among care providers, enabling equitable workload distribution and identifying potential areas where additional resources may be needed.

Turning our attention to financials, we utilize a GROUP BY query to sum up the total payments received per patient. This insight is crucial for the financial department to track revenue streams and to flag any discrepancies in patient billing.

Lastly, we group rooms by their type to count the number of occupied rooms. This information is key for hospital administration to understand room utilization rates, to plan for future patient admissions, and to maintain high standards of patient care.

Grouping highlights patterns that might otherwise be obscured in an unsegmented dataset and provides a strategic advantage in operational decision-making.

1. Grouping the number of patients per doctor:

In [ ]:

```
%%sql

SELECT d.Name, COUNT(*) AS Patient_Count
FROM doctor_table d
JOIN appointment_table a ON d.Doctor_ID = a.Doctor_ID
GROUP BY d.Doctor_ID

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Name	Patient_Count
Heather Valdez	11
Larry Fisher	21
Julie Villegas	8
Richard Lara	18
Gabriela Gonzalez	10

2. Grouping the number of care assignments per care provider:

```
This query counts how many care assignments each care provider has
been given.
```

In [ ]:

```
%%sql

SELECT cp.Name, COUNT(*) AS Assignment_Count
FROM care_provider_table cp
JOIN care_assignment_table ca ON cp.CareProvider_ID = ca.CareProvider_ID
GROUP BY cp.CareProvider_ID

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Name	Assignment_Count
Andres Hill	3
Leslie Thomas	1
Joseph Kirk	1
Kim Blackwell	3
Kristin Mccormick	1

3. Grouping the total payments received per patient:

```
This query sums up the total amount of payments each patient has
made to the healthcare organization.
```

In [ ]:

```
%%sql

SELECT p.Name, SUM(pt.Amount) AS Total_Payments
FROM patient_table p
JOIN payment_table pt ON p.Patient_ID = pt.Patient_ID
GROUP BY p.Patient_ID

LIMIT 5;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
5 rows affected.
```

Out[ ]:

Name	Total_Payments
Eric Gibson	163.48
Michael Hernandez	1274.39
Rodney Ramirez	802.57
Andrea Lambert	1124.11
Brandon Bradley	1561.23

4. Group by Room Type to Count Number of Occupied Rooms:

```
This query provides a count of how many rooms of each type are
currently occupied by patients.
```

In [ ]:

```
%%sql

SELECT Type, COUNT(*) AS OccupiedRooms
FROM room_table
WHERE Patient_ID IS NOT NULL
GROUP BY Type;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
4 rows affected.
```

Out[ ]:

Type	OccupiedRooms
Deluxe	4
Single	16
Double	12
Suite	18

10. Updates (with triggers executing)

When we consider the dynamics of a healthcare database, the ability to track changes over time is not just a feature—it's a necessity. That's where our carefully designed triggers come into play.

Let's delve into the first trigger, log_diagnosis_change. The purpose of this trigger is to capture any changes in a patient's diagnosis. It springs into action after an update is made to the patient_table. If the Disease_Diagnosed field is changed, the trigger automatically logs the old and new diagnoses into a dedicated diagnosis_change_log table, along with the time of change. This meticulous recording is crucial for maintaining a historical record, which can be invaluable for tracking the progression of a patient's condition and for legal or auditing purposes.

To illustrate, if we update a patient's diagnosis to 'Type 2 Diabetes' where it was previously different, the trigger executes seamlessly, creating a log entry that captures this significant change.

The second trigger, archive_patient_before_delete, is a proactive archival tool. Prior to any deletion of a patient record, this trigger archives the current state of the record into an audit table. This archival action is a safeguard, preserving the data before any update occurs, which can be a lifesaver in scenarios where data needs to be recovered or reviewed at a later date.

1. Trigger to Log Changes in Patient Diagnosis:

A trigger that logs the old and new diagnoses whenever a patient's Disease_Diagnosed field is updated.

In [ ]:

```
%%sql

CREATE TRIGGER log_diagnosis_change
AFTER UPDATE ON patient_table
FOR EACH ROW
BEGIN
    IF OLD.Disease_Diagnosed <> NEW.Disease_Diagnosed THEN
        INSERT INTO diagnosis_change_log (Patient_ID, Old_Diagnosis, New_Diagnosis,
        VALUES (OLD.Patient_ID, OLD.Disease_Diagnosed, NEW.Disease_Diagnosed, NOW()
    END IF;
END;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]: []

Activate the log_diagnosis_change by updating the diagnosis of a patient...

```
This would activate the log_diagnosis_change trigger if the
original Disease_Diagnosed for Patient_ID 1 is different from 'Type
2 Diabetes'.
```

In [ ]:

```
%%sql

UPDATE patient_table
SET Disease_Diagnosed = 'Type 2 Diabetes'
WHERE Patient_ID = 1 AND Disease_Diagnosed <> 'Type 2 Diabetes';
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
1 rows affected.
```

Out[ ]: []

Show trigger...

In [ ]:

```
%%sql

SELECT * FROM diagnosis_change_log;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
1 rows affected.
```

Out[ ]:

Log_ID	Patient_ID	Old_Diagnosis	New_Diagnosis	Change_Date
1	1	Asthma	Type 2 Diabetes	2023-12-12 00:52:24

In [ ]:

```
%%sql

SELECT * FROM patient_table WHERE Patient_ID = 1;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
1 rows affected.
```

Out[ ]:

Patient_ID	Name	Address	Phone	Email	Date_Of_Registration	Disease
1	Eric Gibson	1034 Crystal Squares Apt. 544 Lake Cindyfort, WI 72686	742-596-9950x0900	christinekramer@example.net	2023-08-26	Typ

2. Trigger to Archive Patient Record Before Update:

```
Before updating a patient's record, this trigger archives the
current state of the record into an audit table. This is useful for
keeping a history of changes over time.
```

In [ ]:

```
%%sql

CREATE TRIGGER archive_patient_before_delete
BEFORE DELETE ON patient_table
FOR EACH ROW
BEGIN
    INSERT INTO patient_audit_table (
        Patient_ID,
        Name,
        Address,
        Phone,
        Email,
        Date_Of_Registration,
        Disease_Diagnosed,
        Prescription
    ) VALUES (
        OLD.Patient_ID,
        OLD.Name,
        OLD.Address,
        OLD.Phone,
        OLD.Email,
        OLD.Date_Of_Registration,
        OLD.Disease_Diagnosed,
        OLD.Prescription
    );
END;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
0 rows affected.
```

Out[ ]:

```
[]
```

---

11. Deleting items that are foreign keys in other tables (show triggers being executed)

> Now, let's consider a few scenarios where these triggers come into play. One example, when we delete a patient from the patient_table, we must also consider the impact this action has on related tables. Our database is designed with an 'ON DELETE CASCADE' clause for foreign key constraints. This means that deleting a patient record will automatically trigger the deletion of all related records in other tables where the patient's ID is a foreign key.
>
> For instance, the patient's appointments, care assignments, and room assignments will also be removed, maintaining the integrity of our database. Let’s check it by running this code block…
>
> While it ensures that no records are left behind, it also means that valuable data related to the deleted patient will be lost

```
unless we have archiving mechanisms in place. This is where our
patient_audit_table comes into play, which, through triggers,
archives patient data before any deletion occurs.
```

When deleting items that are foreign keys in other tables, you must ensure referential integrity. For example, if you delete a patient, you must handle the related appointments:

In [ ]:

```
%%sql

DELETE FROM patient_table WHERE Patient_ID = 1;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
1 rows affected.
```

Out[ ]:

```
[]
```

In [ ]:

```
%%sql

SELECT * FROM patient_audit_table;
```

```
 * mysql://dyga6971:***@applied-sql.cs.colorado.edu:3306/dyga6971
1 rows affected.
```

Out[ ]:

Audit_ID	Patient_ID	Name	Address	Phone	Email	Date_Of_Registratio
1	1	Eric Gibson	1034 Crystal Squares Apt. 544 Lake Cindyfort, WI 72686	742-596-9950x0900	christinekramer@example.net	2023-08-2

```
Thus, even when a patient record is purged from the main table, we
retain their information in an audit trail.

In conclusion, adding these features allows us to keep our active
records clean and relevant, while also preserving the history of
our data for future references.<br>
```