Final Project Report

Hospital Management System - SQL

Tanisha Lohchab

Phase 1:

Developing a thorough business plan to improve hospital management with a SQL-based system was the first step in the Hospital Management System (HMS) project. The primary objective of this plan was to enhance the effectiveness of patient care and administrative processes by creating an extensive database that could manage patient data, appointments, medical records, invoicing, and inventories. In order to ensure proper record-keeping and efficient healthcare delivery, the phase also required curating a dataset with linked tables to contain data on patients, doctors, rooms, prescriptions, nursing staff, and more. The goal of this work was to improve the healthcare industry by improving data management and automating processes.

1. **Description of Project:**
   In the realm of healthcare, where hospitals play a pivotal role in delivering quality medical services to individuals facing diverse health challenges, the need for efficient management and record-keeping is paramount. Hospitals, amidst the complexities of daily operations, from patient care to administrative tasks, require a robust database system to maintain accurate and accessible records. The Hospital Management System (HMS) is designed to address the intricate needs of healthcare institutions, encompassing hospitals and clinics alike. This SQL-based model serves as the digital backbone, empowering hospitals to seamlessly streamline and automate both administrative and clinical processes. Within this comprehensive platform, patient data, appointment scheduling, medical records, billing, and inventory management are seamlessly orchestrated. Furthermore, the HMS extends its capabilities to encompass staff management, efficiently handling the roles and responsibilities of doctors, nurses, ward boys, and administrative personnel. It also includes advanced features for tracking hospital admissions and generating comprehensive discharge summaries. In essence, the HMS revolutionizes the healthcare landscape, enhancing the overall patient care experience through efficient data management and automation.

2. **Description of the Data:**
   The dataset associated with the HMS project comprises several interconnected tables, each serving a specific purpose. We generated the data for the tables ourselves. These tables include:

   Patients: This table stores comprehensive patient information, including personal details, medical history, and insurance information.

   Physician: Information about healthcare professionals, their specialties, contact details, and schedules are stored here.

Rooms: This table manages details related to hospital rooms, such as room numbers, availability, and patient assignments.

Medication: Tracks the inventory of medical supplies and equipment, including stock levels, reorder points, and supplier information.

Nurses: Contains information about nursing staff, their roles, and responsibilities within the healthcare facility.

Appointment: Manages appointments, including unique appointment IDs, patient IDs, prep nurse IDs, physician IDs, scheduled start and end times, and examination room assignments.

Hospital Stay: Records patient admissions, including admission dates, discharge dates, and reasons for admission.

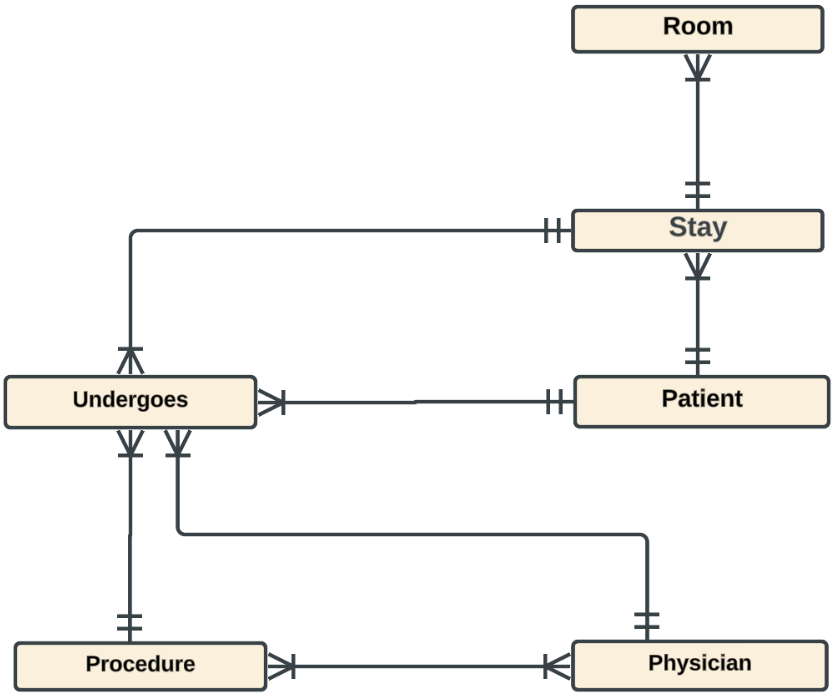
Undergoes: Records patient medical procedures, including patient ID, procedure ID, admission ID, date, physician ID, and assisting nurse ID.

Procedure: Records medical procedures, including unique procedure codes, procedure names, and associated costs.

<u>Phase 2:</u>

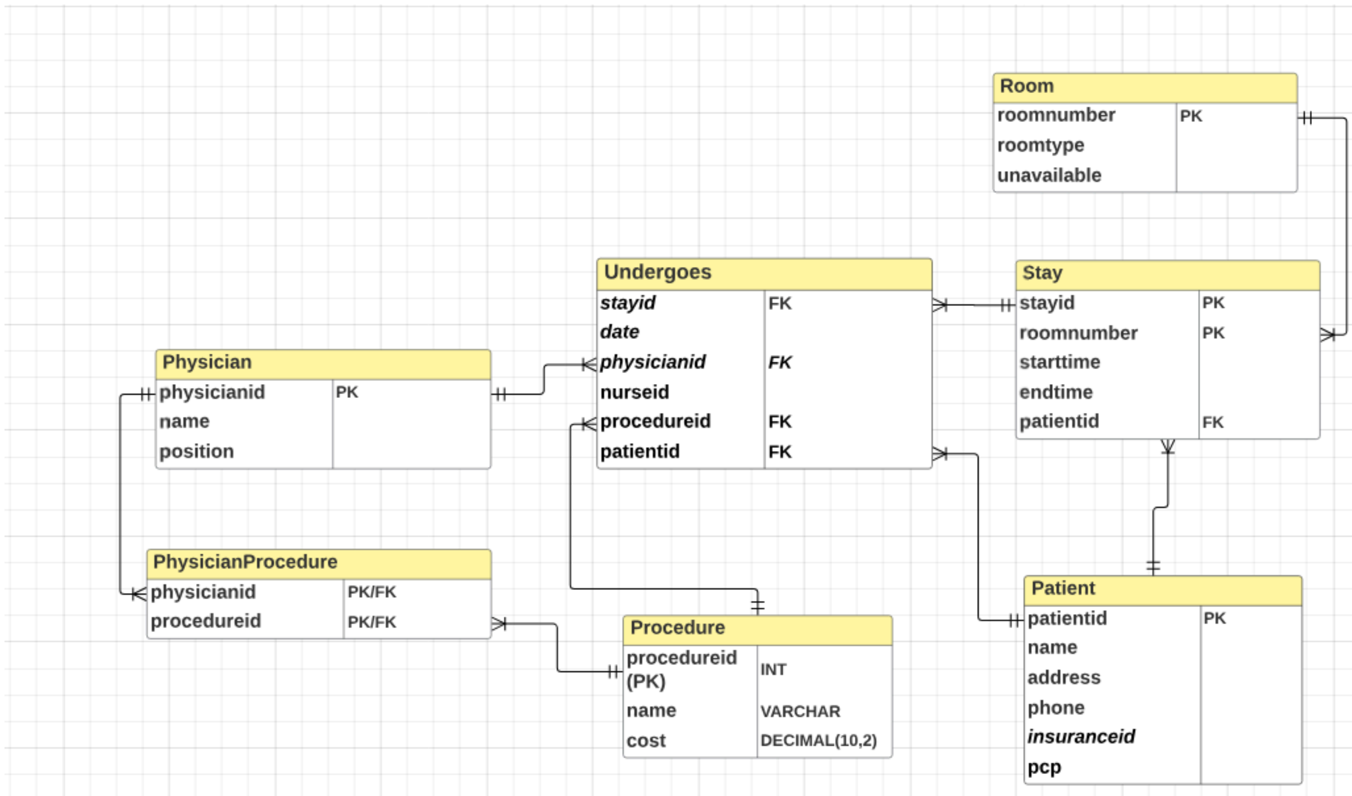
In a conceptual model, we narrowed down on the essential tables required for a Hospital Management System to mainly 6 tables namely Patient, Physician, Room, Stay, Undergoes and Procedure and established relationships between these unique tables. In the logical model we analyzed the columns for each table and established the Primary and Foreign keys. We also used a junction table between Physician and Procedure tables to reduce the many-to-many relationships between them.

3. Develop a Conceptual Model. Consider 4 or 5 entities. Make sure you have at least one many-to-many relationship. Explain with data why it's a many-to-many relationship.



Patient - Stay (One-to-Many): Each patient can have multiple stays in the hospital.
Stay-Room (Many-to-One): Each stay is assigned to one room, but each room can have multiple stays over time.
Patient - Undergoes (One-to-Many): Each patient can undergo multiple procedures (as recorded in the Undergoes table).
Undergoes - Procedure (Many-to-One): Each procedure can be undergone by multiple patients, and each instance of undergoing a procedure is a unique event.
Procedure - Physician (Many-to-Many): Physicians can perform multiple procedures, and each procedure can be performed by multiple physicians.
Physician - Undergoes (One-to-Many): Each physician can be involved in multiple procedure instances.

4. Develop a Logical Model using the Conceptual Model. Make sure you come up with a junction entity to resolve the many-to-many relationship.

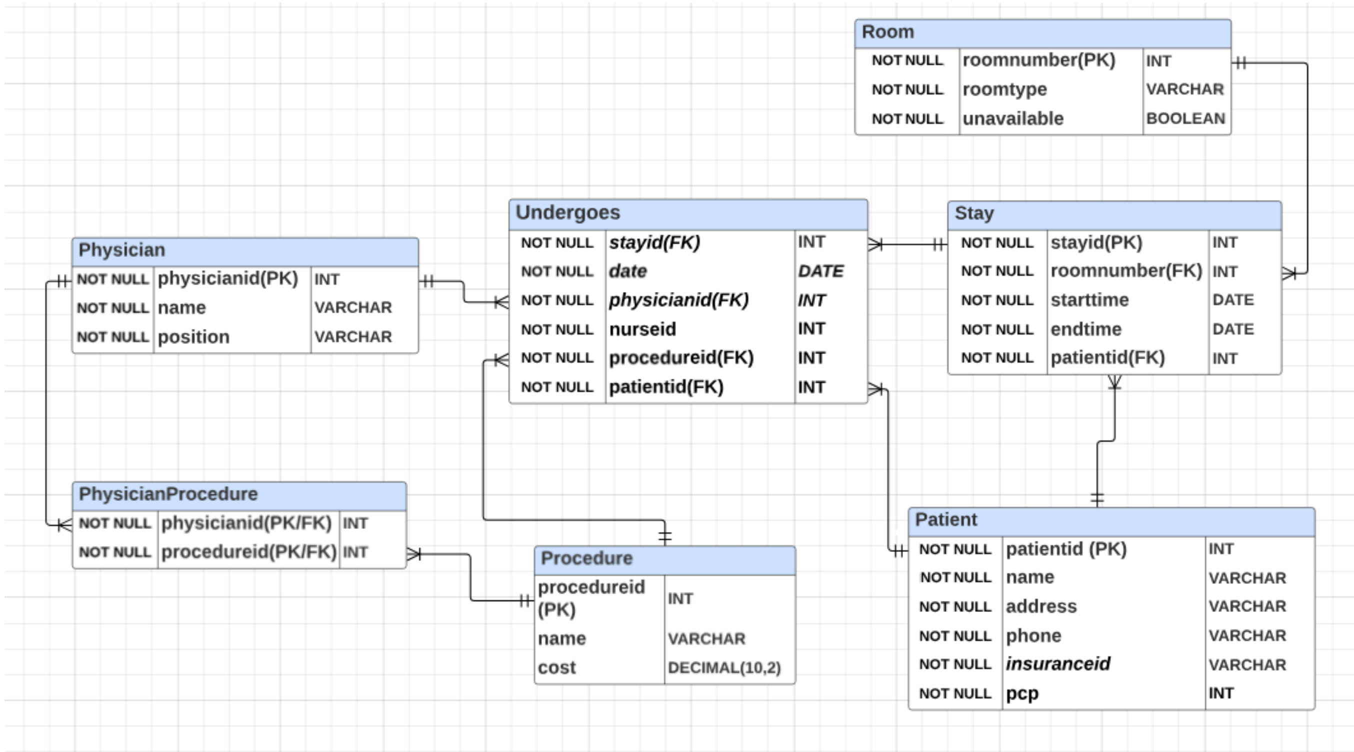


We used a junction table PhysicianProcedure to resolve the many-to-many relationship between Physician and Procedure.
Physician - PhysicianProcedure (One-to-many) : Each physician can take part in multiple procedures.
PhysicianProcedure - Procedure (Many-to-one) : Many physicians take part in a procedure.

Phase 3:

We developed a physical model using the previously developed logical model as a reference. In the physical model, we mentioned the data type and null/ not null aspect for each column. We executed multi-table joins, subqueries, aggregation, and advanced operators like NOT EXISTS and CASE statements to enhance the scalability of the Hospital Management System.

5. Develop the physical model based on the Logical Model



6. Create tables using a database system. Insert data into the database tables. You must provide the DDL (CREATE TABLE statements), INSERT statements, and SELECT statements.

```python
# Create Patient table
cursor.execute('''
CREATE TABLE Patient (
    patientid INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    address TEXT NOT NULL,
    phone TEXT NOT NULL,
    insuranceid TEXT NOT NULL,
    pcp INTEGER NOT NULL,
    FOREIGN KEY (pcp) REFERENCES Physician(physicianid)
)
''')

# Create Procedure table
cursor.execute('''
CREATE TABLE Procedure (
    procedureid INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    cost REAL NOT NULL
)
''')

# Create Room table
cursor.execute('''
CREATE TABLE Room (
    roomnumber INTEGER PRIMARY KEY,
    roomtype TEXT NOT NULL,
    unavailable BOOLEAN NOT NULL
)
''')

# Create Physician table
cursor.execute('''
CREATE TABLE Physician (
    physicianid INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    position TEXT NOT NULL
)
''')

# Create Undergoes table
cursor.execute('''
CREATE TABLE Undergoes (
    patientid INTEGER NOT NULL,
    procedureid INTEGER NOT NULL,
    stayid INTEGER NOT NULL,
    date TEXT NOT NULL,
    PRIMARY KEY (patientid, procedureid, stayid),
    FOREIGN KEY (patientid) REFERENCES Patient(patientid),
    FOREIGN KEY (procedureid) REFERENCES Procedure(procedureid),
    FOREIGN KEY (stayid) REFERENCES Stay(stayid)
)
''')
```

```python
# Create PhysicianProcedure table
cursor.execute('''
CREATE TABLE PhysicianProcedure (
    physicianid INTEGER NOT NULL,
    procedureid INTEGER NOT NULL,
    PRIMARY KEY (physicianid, procedureid),
    FOREIGN KEY (physicianid) REFERENCES Physician(physicianid),
    FOREIGN KEY (procedureid) REFERENCES Procedure(procedureid)
)
''')

# Create Stay table
cursor.execute('''
CREATE TABLE Stay (
    stayid INTEGER PRIMARY KEY,
    patientid INTEGER NOT NULL,
    start_time TEXT NOT NULL,
    end_time TEXT NOT NULL,
    roomnumber INTEGER NOT NULL,
    FOREIGN KEY (patientid) REFERENCES Patient(patientid),
    FOREIGN KEY (roomnumber) REFERENCES Room(roomnumber)
)
''')

# Commit the transaction
conn.commit()
```

✓ 0.0s

```python
cursor.execute("""SELECT name FROM sqlite_master WHERE type='table';""")
print('List of Tables present in the Database')
table_list = [table[0] for table in cursor.fetchall()]
table_list
```

✓ 0.0s

```
List of Tables present in the Database

['Patient',
 'Procedure',
 'Room',
 'Physician',
 'Undergoes',
 'PhysicianProcedure',
 'Stay']
```

```python
# Insert data into Undergoes table
undergoes = [
    (100000001, 6, 3215, '2008-02-05'),
    (100000001, 2, 3215, '2008-03-05'),
    (100000004, 1, 3217, '2008-07-05'),
    (100000004, 5, 3217, '2008-09-05'),
    (100000001, 7, 3215, '2008-10-05'),
    (100000004, 4, 3217, '2008-13-05'),  # This date is invalid, assuming a typo
]
cursor.executemany('INSERT INTO Undergoes VALUES (?,?,?,?)', undergoes)

# Insert data into PhysicianProcedure table
physician_procedures = [
    (3, 6),
    (7, 2),
    (3, 1),
    (6, 5),
    (7, 7),
    (3, 4),
]
cursor.executemany('INSERT INTO PhysicianProcedure VALUES (?,?)', physician_procedures)

# Insert data into Stay table
stays = [
    (3215, 100000001, '2008-01-05', '2008-04-05', 111),
    (3216, 100000003, '2008-03-05', '2008-14-05', 123),  # This end date is invalid, assuming a typo
    (3217, 100000004, '2008-02-05', '2008-03-05', 112),
]
cursor.executemany('INSERT INTO Stay VALUES (?,?,?,?,?)', stays)
```

```python
# Insert data into Patient table
patients = [
    (100000001, 'John Smith', '42 Foobar Lane', '555-0256', '68476213', 1),
    (100000002, 'Grace Ritchie', '37 Snafu Drive', '555-0512', '36546321', 2),
    (100000003, 'Random J. Patient', '101 Omgbbq Street', '555-1204', '65465421', 2),
    (100000004, 'Dennis Doe', '1100 Foobaz Avenue', '555-2048', '68421879', 3),
]
cursor.executemany('INSERT INTO Patient VALUES (?,?,?,?,?,?)', patients)

# Insert data into Procedure table
procedures = [
    (1, 'Reverse Rhinopodoplasty', 1500.00),
    (2, 'Obtuse Pyloric Recombobulation', 3750.00),
    (3, 'Folded Demiophtalmectomy', 4500.00),
    (4, 'Complete Walletectomy', 10000.00),
    (5, 'Obfuscated Dermogastrotomy', 4899.00),
    (6, 'Reversible Pancreomyoplasty', 5600.00),
    (7, 'Follicular Demiectomy', 25.00),
]
cursor.executemany('INSERT INTO Procedure VALUES (?,?,?)', procedures)

# Insert data into Room table
rooms = [
    (101, 'Single', 0),
    (102, 'Single', 1),
    (103, 'Double', 0),
    (111, 'Single', 0),
    (112, 'Single', 1),
    (113, 'Double', 0),
    (121, 'Single', 0),
    (122, 'Single', 0),
    (123, 'Single', 0),
]
cursor.executemany('INSERT INTO Room VALUES (?,?,?)', rooms)

# Insert data into Physician table
physicians = [
    (1, 'John Dorian', 'Staff Internist'),
    (2, 'Elliot Reid', 'Attending Physician'),
    (3, 'Christopher Turk', 'Surgical Attending Physician'),
    (4, 'Percival Cox', 'Senior Attending Physician'),
    (5, 'Bob Kelso', 'Head Chief of Medicine'),
    (6, 'Todd Quinlan', 'Surgical Attending Physician'),
    (7, 'John Wen', 'Surgical Attending Physician'),
    (8, 'Keith Dudemeister', 'MD Resident'),
    (9, 'Molly Clock', 'Attending Psychiatrist'),
]
cursor.executemany('INSERT INTO Physician VALUES (?,?,?)', physicians)
```

```python
pd.read_sql("""SELECT * FROM PhysicianProcedure ORDER BY physicianid, procedureid;""", conn)
```

✓ 0.0s

	physicianid	procedureid
0	3	1
1	3	4
2	3	6
3	6	5
4	7	2
5	7	7

```python
pd.read_sql("""SELECT * FROM Stay ORDER BY stayid;""", conn)
```

✓ 0.0s

	stayid	patientid	start_time	end_time	roomnumber
0	3215	100000001	2008-01-05	2008-04-05	111
1	3216	100000003	2008-03-05	2008-14-05	123
2	3217	100000004	2008-02-05	2008-03-05	112

\

```python
pd.read_sql("""SELECT * FROM Room ORDER BY roomnumber;""", conn)
```

✓ 0.0s

	roomnumber	roomtype	unavailable
0	101	Single	0
1	102	Single	1
2	103	Double	0
3	111	Single	0
4	112	Single	1
5	113	Double	0
6	121	Single	0
7	122	Single	0
8	123	Single	0

```python
pd.read_sql("""SELECT * FROM Physician ORDER BY physicianid;""", conn)
```

✓ 0.0s

	physicianid	name	position
0	1	John Dorian	Staff Internist
1	2	Elliot Reid	Attending Physician
2	3	Christopher Turk	Surgical Attending Physician
3	4	Percival Cox	Senior Attending Physician
4	5	Bob Kelso	Head Chief of Medicine
5	6	Todd Quinlan	Surgical Attending Physician
6	7	John Wen	Surgical Attending Physician
7	8	Keith Dudemeister	MD Resident
8	9	Molly Clock	Attending Psychiatrist

```python
pd.read_sql("""SELECT * FROM Undergoes ORDER BY patientid, procedureid, stayid;""", conn)
```

✓ 0.0s

	patientid	procedureid	stayid	date
0	100000001	2	3215	2008-03-05
1	100000001	6	3215	2008-02-05
2	100000001	7	3215	2008-10-05
3	100000004	1	3217	2008-07-05
4	100000004	4	3217	2008-13-05
5	100000004	5	3217	2008-09-05

```python
#sqlite_master is a table with database schema
pd.read_sql(""" SELECT *
                FROM sqlite_master
                WHERE type='table';""",
            conn)
```

✓ 0.1s

	type	name	tbl_name	rootpage	sql
0	table	Patient	Patient	2	CREATE TABLE Patient (\n patientid INTEGER ...
1	table	Procedure	Procedure	3	CREATE TABLE Procedure (\n procedureid INTE...
2	table	Room	Room	4	CREATE TABLE Room (\n roomnumber INTEGER PR...
3	table	Physician	Physician	5	CREATE TABLE Physician (\n physicianid INTE...
4	table	Undergoes	Undergoes	6	CREATE TABLE Undergoes (\n patientid INTEGE...
5	table	PhysicianProcedure	PhysicianProcedure	8	CREATE TABLE PhysicianProcedure (\n physici...
6	table	Stay	Stay	10	CREATE TABLE Stay (\n stayid INTEGER PRIMAR...

```python
pd.read_sql("""SELECT * FROM Patient ORDER BY patientid;""", conn)
```

✓ 0.0s

	patientid	name	address	phone	insuranceid	pcp
0	100000001	John Smith	42 Foobar Lane	555-0256	68476213	1
1	100000002	Grace Ritchie	37 Snafu Drive	555-0512	36546321	2
2	100000003	Random J. Patient	101 Omgbbq Street	555-1204	65465421	2
3	100000004	Dennis Doe	1100 Foobaz Avenue	555-2048	68421879	3

```python
pd.read_sql("""SELECT * FROM Procedure ORDER BY procedureid;""", conn)
```

✓ 0.0s

	procedureid	name	cost
0	1	Reverse Rhinopodoplasty	1500.0
1	2	Obtuse Pyloric Recombobulation	3750.0
2	3	Folded Demiophtalmectomy	4500.0
3	4	Complete Walletectomy	10000.0
4	5	Obfuscated Dermogastrotomy	4899.0
5	6	Reversible Pancreomyoplasty	5600.0
6	7	Follicular Demiectomy	25.0

7. Create a variety of SQL queries to retrieve data from one or many tables:

1. Create the tables that you have come up with (the table must be based on the Physical Model).

(a) Columns, Primary Key (PK), Data Type and length, and NULL/NOT NULL need to be implemented, per the Physical Model.

(b) Show the table definition (DDL) that you implemented (not in a graphical view).

```
# Create Patient table
cursor.execute('''
CREATE TABLE Patient (
    patientid INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    address TEXT NOT NULL,
    phone TEXT NOT NULL,
    insuranceid TEXT NOT NULL,
    pcp INTEGER NOT NULL,
    FOREIGN KEY (pcp) REFERENCES Physician(physicianid)
)
''')

# Create Procedure table
cursor.execute('''
CREATE TABLE Procedure (
    procedureid INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    cost REAL NOT NULL
)
''')

# Create Room table
cursor.execute('''
CREATE TABLE Room (
    roomnumber INTEGER PRIMARY KEY,
    roomtype TEXT NOT NULL,
    unavailable BOOLEAN NOT NULL
)
''')

# Create Physician table
cursor.execute('''
CREATE TABLE Physician (
    physicianid INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    position TEXT NOT NULL
)
''')

# Create Undergoes table
cursor.execute('''
CREATE TABLE Undergoes (
    patientid INTEGER NOT NULL,
    procedureid INTEGER NOT NULL,
    stayid INTEGER NOT NULL,
    date TEXT NOT NULL,
    PRIMARY KEY (patientid, procedureid, stayid),
    FOREIGN KEY (patientid) REFERENCES Patient(patientid),
    FOREIGN KEY (procedureid) REFERENCES Procedure(procedureid),
    FOREIGN KEY (stayid) REFERENCES Stay(stayid)
)
''')
```

```python
# Create PhysicianProcedure table
cursor.execute('''
CREATE TABLE PhysicianProcedure (
    physicianid INTEGER NOT NULL,
    procedureid INTEGER NOT NULL,
    PRIMARY KEY (physicianid, procedureid),
    FOREIGN KEY (physicianid) REFERENCES Physician(physicianid),
    FOREIGN KEY (procedureid) REFERENCES Procedure(procedureid)
)
''')

# Create Stay table
cursor.execute('''
CREATE TABLE Stay (
    stayid INTEGER PRIMARY KEY,
    patientid INTEGER NOT NULL,
    start_time TEXT NOT NULL,
    end_time TEXT NOT NULL,
    roomnumber INTEGER NOT NULL,
    FOREIGN KEY (patientid) REFERENCES Patient(patientid),
    FOREIGN KEY (roomnumber) REFERENCES Room(roomnumber)
)
''')

# Commit the transaction
conn.commit()
```

✓ 0.0s

```python
cursor.execute("""SELECT name FROM sqlite_master WHERE type='table';""")
print('List of Tables present in the Database')
table_list = [table[0] for table in cursor.fetchall()]
table_list
```

✓ 0.0s

```
List of Tables present in the Database

['Patient',
 'Procedure',
 'Room',
 'Physician',
 'Undergoes',
 'PhysicianProcedure',
 'Stay']
```

(c) Insert the complete set of data that you have come up with and show the insert statements used.

```
# Insert data into Undergoes table
undergoes = [
    (100000001, 6, 3215, '2008-02-05'),
    (100000001, 2, 3215, '2008-03-05'),
    (100000004, 1, 3217, '2008-07-05'),
    (100000004, 5, 3217, '2008-09-05'),
    (100000001, 7, 3215, '2008-10-05'),
    (100000004, 4, 3217, '2008-13-05'),  # This date is invalid, assuming a typo
]
cursor.executemany('INSERT INTO Undergoes VALUES (?,?,?,?)', undergoes)

# Insert data into PhysicianProcedure table
physician_procedures = [
    (3, 6),
    (7, 2),
    (3, 1),
    (6, 5),
    (7, 7),
    (3, 4),
]
cursor.executemany('INSERT INTO PhysicianProcedure VALUES (?,?)', physician_procedures)

# Insert data into Stay table
stays = [
    (3215, 100000001, '2008-01-05', '2008-04-05', 111),
    (3216, 100000003, '2008-03-05', '2008-14-05', 123),  # This end date is invalid, assuming a typo
    (3217, 100000004, '2008-02-05', '2008-03-05', 112),
]
cursor.executemany('INSERT INTO Stay VALUES (?,?,?,?,?)', stays)
```

```python
# Insert data into Patient table
patients = [
    (100000001, 'John Smith', '42 Foobar Lane', '555-0256', '68476213', 1),
    (100000002, 'Grace Ritchie', '37 Snafu Drive', '555-0512', '36546321', 2),
    (100000003, 'Random J. Patient', '101 Omgbbq Street', '555-1204', '65465421', 2),
    (100000004, 'Dennis Doe', '1100 Foobaz Avenue', '555-2048', '68421879', 3),
]
cursor.executemany('INSERT INTO Patient VALUES (?,?,?,?,?,?)', patients)

# Insert data into Procedure table
procedures = [
    (1, 'Reverse Rhinopodoplasty', 1500.00),
    (2, 'Obtuse Pyloric Recombobulation', 3750.00),
    (3, 'Folded Demiophtalmectomy', 4500.00),
    (4, 'Complete Walletectomy', 10000.00),
    (5, 'Obfuscated Dermogastrotomy', 4899.00),
    (6, 'Reversible Pancreomyoplasty', 5600.00),
    (7, 'Follicular Demiectomy', 25.00),
]
cursor.executemany('INSERT INTO Procedure VALUES (?,?,?)', procedures)

# Insert data into Room table
rooms = [
    (101, 'Single', 0),
    (102, 'Single', 1),
    (103, 'Double', 0),
    (111, 'Single', 0),
    (112, 'Single', 1),
    (113, 'Double', 0),
    (121, 'Single', 0),
    (122, 'Single', 0),
    (123, 'Single', 0),
]
cursor.executemany('INSERT INTO Room VALUES (?,?,?)', rooms)

# Insert data into Physician table
physicians = [
    (1, 'John Dorian', 'Staff Internist'),
    (2, 'Elliot Reid', 'Attending Physician'),
    (3, 'Christopher Turk', 'Surgical Attending Physician'),
    (4, 'Percival Cox', 'Senior Attending Physician'),
    (5, 'Bob Kelso', 'Head Chief of Medicine'),
    (6, 'Todd Quinlan', 'Surgical Attending Physician'),
    (7, 'John Wen', 'Surgical Attending Physician'),
    (8, 'Keith Dudemeister', 'MD Resident'),
    (9, 'Molly Clock', 'Attending Psychiatrist'),
]
cursor.executemany('INSERT INTO Physician VALUES (?,?,?)', physicians)
```

(d) Retrieve the data from each table by using the SELECT * statement and order by PK column(s).

```
pd.read_sql("""SELECT * FROM PhysicianProcedure ORDER BY physicianid, procedureid;""", conn)
```

✓ 0.0s

	physicianid	procedureid
0	3	1
1	3	4
2	3	6
3	6	5
4	7	2
5	7	7

```
pd.read_sql("""SELECT * FROM Stay ORDER BY stayid;""", conn)
```

✓ 0.0s

	stayid	patientid	start_time	end_time	roomnumber
0	3215	100000001	2008-01-05	2008-04-05	111
1	3216	100000003	2008-03-05	2008-14-05	123
2	3217	100000004	2008-02-05	2008-03-05	112

```python
pd.read_sql("""SELECT * FROM Room ORDER BY roomnumber;""", conn)
```

✓ 0.0s

	roomnumber	roomtype	unavailable
0	101	Single	0
1	102	Single	1
2	103	Double	0
3	111	Single	0
4	112	Single	1
5	113	Double	0
6	121	Single	0
7	122	Single	0
8	123	Single	0

```python
pd.read_sql("""SELECT * FROM Physician ORDER BY physicianid;""", conn)
```

✓ 0.0s

	physicianid	name	position
0	1	John Dorian	Staff Internist
1	2	Elliot Reid	Attending Physician
2	3	Christopher Turk	Surgical Attending Physician
3	4	Percival Cox	Senior Attending Physician
4	5	Bob Kelso	Head Chief of Medicine
5	6	Todd Quinlan	Surgical Attending Physician
6	7	John Wen	Surgical Attending Physician
7	8	Keith Dudemeister	MD Resident
8	9	Molly Clock	Attending Psychiatrist

```python
pd.read_sql("""SELECT * FROM Undergoes ORDER BY patientid, procedureid, stayid;""", conn)
```

✓ 0.0s

	patientid	procedureid	stayid	date
0	100000001	2	3215	2008-03-05
1	100000001	6	3215	2008-02-05
2	100000001	7	3215	2008-10-05
3	100000004	1	3217	2008-07-05
4	100000004	4	3217	2008-13-05
5	100000004	5	3217	2008-09-05

```python
#sqlite_master is a table with database schema
pd.read_sql(""" SELECT *
                FROM sqlite_master
                WHERE type='table';""",
            conn)
```

✓ 0.1s

	type	name	tbl_name	rootpage	sql
0	table	Patient	Patient	2	CREATE TABLE Patient (\n patientid INTEGER ...
1	table	Procedure	Procedure	3	CREATE TABLE Procedure (\n procedureid INTE...
2	table	Room	Room	4	CREATE TABLE Room (\n roomnumber INTEGER PR...
3	table	Physician	Physician	5	CREATE TABLE Physician (\n physicianid INTE...
4	table	Undergoes	Undergoes	6	CREATE TABLE Undergoes (\n patientid INTEGE...
5	table	PhysicianProcedure	PhysicianProcedure	8	CREATE TABLE PhysicianProcedure (\n physici...
6	table	Stay	Stay	10	CREATE TABLE Stay (\n stayid INTEGER PRIMAR...

```python
pd.read_sql("""SELECT * FROM Patient ORDER BY patientid;""", conn)
```

✓ 0.0s

	patientid	name	address	phone	insuranceid	pcp
0	100000001	John Smith	42 Foobar Lane	555-0256	68476213	1
1	100000002	Grace Ritchie	37 Snafu Drive	555-0512	36546321	2
2	100000003	Random J. Patient	101 Omgbbq Street	555-1204	65465421	2
3	100000004	Dennis Doe	1100 Foobaz Avenue	555-2048	68421879	3

```python
pd.read_sql("""SELECT * FROM Procedure ORDER BY procedureid;""", conn)
```

✓ 0.0s

	procedureid	name	cost
0	1	Reverse Rhinopodoplasty	1500.0
1	2	Obtuse Pyloric Recombobulation	3750.0
2	3	Folded Demiophtalmectomy	4500.0
3	4	Complete Walletectomy	10000.0
4	5	Obfuscated Dermogastrotomy	4899.0
5	6	Reversible Pancreomyoplasty	5600.0
6	7	Follicular Demiectomy	25.0

2. Write an SQL involving the junction table and two other related tables. You must use the INNER JOIN to connect with all three tables. The database that you created must be included in your SQL queries.

```
conn = sqlite3.connect('/Users/tanishalohchab/Documents/SQL Database/hmsdatabase.sqlite')
cursor = conn.cursor()
```

✓ 0.0s

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    Procedure.name AS ProcedureName,
    Undergoes.date AS ProcedureDate
FROM
    Undergoes
INNER JOIN
    Patient ON Undergoes.patientid = Patient.patientid
INNER JOIN
    Procedure ON Undergoes.procedureid = Procedure.procedureid
ORDER BY
    Undergoes.date;
""", conn)
```

✓ 0.0s

	PatientName	ProcedureName	ProcedureDate
0	John Smith	Reversible Pancreomyoplasty	2008-02-05
1	John Smith	Obtuse Pyloric Recombobulation	2008-03-05
2	Dennis Doe	Reverse Rhinopodoplasty	2008-07-05
3	Dennis Doe	Obfuscated Dermogastrotomy	2008-09-05
4	John Smith	Follicular Demiectomy	2008-10-05
5	Dennis Doe	Complete Walletectomy	2008-13-05

We are using the hmsdatabase for this query.

3. Write an SQL by including two or more tables and using the LEFT OUTER JOIN. Show the results and sort the results by key field(s). Interpret the results compared to what an INNER JOIN does.

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    Patient.patientid AS PatientID,
    Physician.name AS PhysicianName,
    Physician.position AS PhysicianPosition
FROM
    Patient
LEFT OUTER JOIN
    Physician ON Patient.pcp = Physician.physicianid
ORDER BY
    Patient.patientid;
""", conn)
```

✓ 0.0s

	PatientName	PatientID	PhysicianName	PhysicianPosition
0	John Smith	100000001	John Dorian	Staff Internist
1	Grace Ritchie	100000002	Elliot Reid	Attending Physician
2	Random J. Patient	100000003	Elliot Reid	Attending Physician
3	Dennis Doe	100000004	Christopher Turk	Surgical Attending Physician

The LEFT OUTER JOIN is useful when you want to ensure that all records from the "left" table (Patient in this case) are included in the result set, regardless of whether there is a matching record in the "right" table (Physician). This is particularly useful for identifying records in the primary table that don't have corresponding entries in the related table.

The INNER JOIN would be useful for generating a report of all procedures undergone by patients, including the details of each procedure and the corresponding patient information.

4. Write a single-row subquery. Show the results and sort the results by key field(s). Interpret the output.

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    Undergoes.procedureid,
    Procedure.name AS ProcedureName,
    Procedure.cost
FROM
    Patient
INNER JOIN
    Undergoes ON Patient.patientid = Undergoes.patientid
INNER JOIN
    Procedure ON Undergoes.procedureid = Procedure.procedureid
WHERE
    Procedure.cost = (SELECT MAX(cost) FROM Procedure)
ORDER BY
    Patient.patientid;
""", conn)
```

✓ 0.0s

	PatientName	procedureid	ProcedureName	cost
0	Dennis Doe	4	Complete Walletectomy	10000.0

Interpretation of the results:

This use of a single-row subquery is effective for scenarios where a specific piece of data (in this case, the maximum procedure cost) needs to be integrated into the criteria for a larger query. It's termed "single-row" because the subquery is designed to return only one row (the maximum cost) to be used in the main query's comparison.

Here the output includes only those patients who have undergone the most expensive procedure listed in the Procedure table and details about the procedure (like its name and cost) along with patient information. If no patient has undergone the most expensive procedure, the result set will be empty.

5. Write a multiple-row subquery. Show the results and sort the results by key field(s). Interpret the output.

```
pd.read_sql("""SELECT
    Physician.name AS PhysicianName,
    Physician.position
FROM
    Physician
WHERE
    Physician.physicianid IN (
        SELECT physicianid
        FROM PhysicianProcedure
        GROUP BY physicianid
        HAVING COUNT(procedureid) > 1
    )
ORDER BY
    Physician.physicianid;
""", conn)
```

[33] ✓ 0.0s

	PhysicianName	position
0	Christopher Turk	Surgical Attending Physician
1	John Wen	Surgical Attending Physician

Interpretation of the results:

This approach is often used in scenarios where we need to filter data based on a condition that involves a variable number of related records in another table. It contrasts with a single-row subquery, which is designed to compare against a single, specific value or condition.

The output includes the names and positions of physicians who are qualified to perform multiple types of procedures. If a physician is only associated with one type of procedure or none at all, they will not appear in this list.

6. Write an SQL to aggregate the results by using multiple columns in the SELECT clause. Interpret the output.

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    COUNT(Undergoes.procedureid) AS NumberOfProcedures,
    SUM(Procedure.cost) AS TotalCost
FROM
    Undergoes
INNER JOIN
    Patient ON Undergoes.patientid = Patient.patientid
INNER JOIN
    Procedure ON Undergoes.procedureid = Procedure.procedureid
GROUP BY
    Patient.patientid, Patient.name
ORDER BY
    Patient.patientid;
""", conn)
```

✓ 0.0s

	PatientName	NumberOfProcedures	TotalCost
0	John Smith	3	9375.0
1	Dennis Doe	3	16399.0

Interpretation of the results:

This type of aggregation is useful in scenarios where you need a summarized view of data based on multiple attributes. In this case, it provides a comprehensive overview of each patient's interactions with the hospital system in terms of the number of procedures and the total expenditure on these procedures.

The output includes the name of each patient, the total number of different procedures that each patient has undergone and the total cost incurred by each patient for these procedures.

7. Write a subquery using the NOT IN operator. Show the results and sort the results by key field(s). Interpret the output.

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    Patient.patientid
FROM
    Patient
WHERE
    Patient.patientid NOT IN (
        SELECT Undergoes.patientid
        FROM Undergoes
        INNER JOIN Procedure ON Undergoes.procedureid = Procedure.procedureid
        WHERE Procedure.cost >= (SELECT AVG(cost) FROM Procedure)
    )
ORDER BY
    Patient.patientid;
""", conn)
```

✓ 0.0s

	PatientName	patientid
0	Grace Ritchie	100000002
1	Random J. Patient	100000003

Interpretation of the results:

The use of the NOT IN operator with a subquery is a powerful way to filter out records based on a set of criteria defined in another part of the database. In this case, it helps isolate patients based on the absence of certain high-cost medical procedures in their records.

The output includes the Names and IDs of patients who have not undergone any of the more expensive procedures (cost equal to or above the average cost). This query is useful for identifying patients who have potentially less complex or less expensive medical histories, as indicated by the types of procedures they have not undergone.

8. Write a query using a CASE statement. Show the results and sort the results by key field(s). Interpret the output.

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    Patient.patientid,
    Physician.position AS PhysicianPosition,
    CASE
        WHEN Physician.position LIKE '%Specialist%' THEN 'Specialist'
        WHEN Physician.position LIKE '%General%' THEN 'General'
        ELSE 'Other'
    END AS PhysicianCategory
FROM
    Patient
INNER JOIN
    Physician ON Patient.pcp = Physician.physicianid
ORDER BY
    Patient.patientid;
""", conn)
```

✓ 0.0s

	PatientName	patientid	PhysicianPosition	PhysicianCategory
0	John Smith	100000001	Staff Internist	Other
1	Grace Ritchie	100000002	Attending Physician	Other
2	Random J. Patient	100000003	Attending Physician	Other
3	Dennis Doe	100000004	Surgical Attending Physician	Other

Interpretation of the results:

The CASE statement in SQL is particularly useful for transforming, categorizing, or summarizing data during the retrieval process, especially when such categorization is not explicitly stored in the database.

The output includes the name and ID of each patient, The position of their primary care physician and a categorized label for each physician based on their position - whether they are a specialist, a general physician, or neither (Other). This could help in understanding the distribution of patients under different types of physicians and planning resources accordingly.

9. Write a query using the NOT EXISTS operator. Show the results and sort the results by key field(s). Interpret the output.

```
pd.read_sql("""SELECT
    Physician.name AS PhysicianName,
    Physician.physicianid
FROM
    Physician
WHERE NOT EXISTS (
    SELECT 1
    FROM PhysicianProcedure
    WHERE PhysicianProcedure.physicianid = Physician.physicianid
)
ORDER BY
    Physician.physicianid;
""", conn)
```

✓ 0.0s

	PhysicianName	physicianid
0	John Dorian	1
1	Elliot Reid	2
2	Percival Cox	4
3	Bob Kelso	5
4	Keith Dudemeister	8
5	Molly Clock	9

Interpretation of the results:

The NOT EXISTS operator is particularly useful for finding records that do not have a corresponding match in another table. In contrast to NOT IN, NOT EXISTS is often more efficient in SQL, especially with large datasets, as it stops processing as soon as it finds a match.

The output includes names and IDs of physicians who have not been associated with any procedures. This might be useful in a hospital management context to identify physicians who are either new, have administrative roles, or are not currently active in performing procedures.

10. Write a subquery using the NOT NULL operator in the inner query. Show the results and sort the results by key field(s). Interpret the output.

```
pd.read_sql("""SELECT
    Patient.name AS PatientName,
    Patient.patientid,
    Physician.name AS PhysicianName
FROM
    Patient
INNER JOIN
    Physician ON Patient.pcp = Physician.physicianid
WHERE
    Patient.patientid IN (
        SELECT patientid
        FROM Patient
        WHERE pcp IS NOT NULL
    )
ORDER BY
    Patient.patientid;
""", conn)
```

✓ 0.0s

	PatientName	patientid	PhysicianName
0	John Smith	100000001	John Dorian
1	Grace Ritchie	100000002	Elliot Reid
2	Random J. Patient	100000003	Elliot Reid
3	Dennis Doe	100000004	Christopher Turk

Interpretation of the results:

This query is useful in scenarios where the healthcare facility needs to identify patients who are currently under the care of a physician. It helps in understanding patient-physician assignments and ensuring that all patients are adequately covered.

The output includes Names and IDs of patients who have an assigned primary care physician. The name of each patient's primary care physician.

Summary:

The Hospital Management System (HMS) is a platform built on SQL that is intended to increase hospital and clinic productivity. Clinical and administrative procedures like patient data management, appointment scheduling, billing, inventory control, and staff management are automated. The database schema of the Hospital Management System is made up of tables for the following patient, physician, room, stay, undergoes, procedure, and physician procedure. These databases are intended to record and connect multiple aspects of hospital activities. Important patient data is kept in the Patient database, which is connected to the Stay table, which documents specifics about hospital visits. Physician contains information about physicians that is linked to Undergoes, which describes the procedures patients receive, and PhysicianProcedure, a junction table that shows which doctors are qualified to execute specific procedures. treatments lists the many medical treatments that are available, and Room describes the kind and availability of rooms. Through the use of primary and foreign keys, these databases are linked, enabling a relational structure that facilitates complicated queries for hospital administration duties like scheduling and monitoring visits and treatments.

The hospital management database maps relationships where patients may have multiple stays and undergo various procedures, stays are linked to specific rooms, and procedures are linked to both patients and physicians. Each stay is confined to one room, while rooms can host many stays. Procedures are recorded as unique events per patient and are conducted by physicians, with each physician performing numerous procedures across different patients.

The many-to-many relationship between Physician and Procedure is efficiently managed by the database through the use of a junction table called PhysicianProcedure. A physician can be linked to numerous procedures, which are recorded in a one-to-many relationship between the physician and the physician procedure. On the other hand, it also shows how a procedure can have many physicians affiliated with it in a many-to-one relationship from PhysicianProcedure to Procedure. For hospital administration operations, this architecture makes the relational mapping easier to handle and more scalable.

We executed SQL concepts and techniques that can enhance scalability in the Hospital Management System (HMS). Usage of multi-table joins, subqueries, aggregation, and advanced operators like NOT EXISTS and CASE statements encapsulate complex logic within the database layer itself. This allows avoiding redundant data which can cause consistency issues at scale. It also enables creating reusable query templates that provide actionable insights from large volumes of data efficiently, without moving the data across layers. Overall, implementing these database best practices will enable smooth scaling of the HMS database and application layer as the number of patients, procedures and healthcare data continues to grow over time.

With the help of this SQL-based platform, which acts as the digital framework, hospitals may easily automate and streamline both clinical and administrative procedures. Patient information, scheduling, billing, medical records, and inventory management are all smoothly coordinated inside one all-inclusive platform. Additionally, the HMS broadens its scope of functionality to include staff management, effectively managing the duties and obligations of physicians, nurses, ward boys, and administrative staff. Advanced tools for monitoring hospital admissions and creating thorough discharge summaries are also included. The HMS essentially transforms the healthcare industry by improving patient care overall through effective automation and data management.