Name: Saher Saeed

Student ID: 23095056

Github :https://github.com/saeedsahar/Data-Mining.git

Database Overview Report

1. Introduction

The database is a structured relational database containing medical records related to patients, doctors, and their appointments. It consists of three key tables: **Patient**, **Doctor**, and **Appointment**. Each table is interconnected through **primary keys (PKs)** and **foreign keys (FKs)**, ensuring data integrity and logical relationships.

Database Record Summary

- **Total Patients:** 500
- **Total Doctors:** 100
- **Total Appointments:** 1,000

Table	Column Name	Type of Data	Unique Value	Nature
Patient Table				
	Patient_ID	Nominal (Unique ID)	Unique for each patient	Qualitative
	First_Name	Nominal (Name)	Various common names	Qualitative
	Middle_Name	Nominal (Optional)	Some have, some don't	Qualitative
	Last_Name	Nominal (Surname)	Common last names	Qualitative
	Gender	Nominal (Male/Female/Other)	Male, Female, Other	Qualitative
	DateOfBirth	Interval (Date of Birth)	Valid birth dates	Quantitative
	Blood_Type	Nominal (Blood Group)	A+, B-, O+, etc.	Qualitative
	Insurance_Type	Nominal (Insurance)	Private, Public, None	Qualitative
	Postal_Code	Nominal (Geographical)	Valid UK postal codes	Qualitative
Doctor Table				
	Doctor_ID	Nominal (Unique ID)	Unique for each doctor	Qualitative
	First_Name	Nominal (Name)	Various common names	Qualitative
	Middle_Name	Nominal (Optional)	Some have, some don't	Qualitative
	Last_Name	Nominal (Surname)	Common last names	Qualitative
	Specialization	Nominal (Medical Field)	Cardiologist, Surgeon, etc.	Qualitative
	Experience_Years	Ordinal (Years of Practice)	Range 1-40	Quantitative
	Consultation_Fee	Ratio (Fee in £)	Ranges from 10 to 200	Quantitative
Appointment Table				
	Appointment_ID	Nominal (Unique ID)	Unique for each appointment	Qualitative
	Patient_ID	Nominal (Foreign Key)	References Patient_ID	Qualitative
	Doctor_ID	Nominal (Foreign Key)	References Doctor_ID	Qualitative
	Severity_Level	Ordinal (Severity)	Mild, Moderate, Severe	Qualitative
	Duration_Minutes	Ratio (Minutes)	Fixed at 20 min	Quantitative
	Cost	Ratio (Fee in £)	Ranges from 10 to 200	Quantitative
	Payment_Status	Nominal (Payment Status)	Paid, Pending, Not Required	Qualitative
	Consultation_Type	Nominal (Type)	In-Person, Telemedicine	Qualitative
	ScheduleTime_Start	Interval (Start Time)	Appointment start time	Quantitative
	ScheduleTime_End	Interval (End Time)	Appointment end time	Quantitative

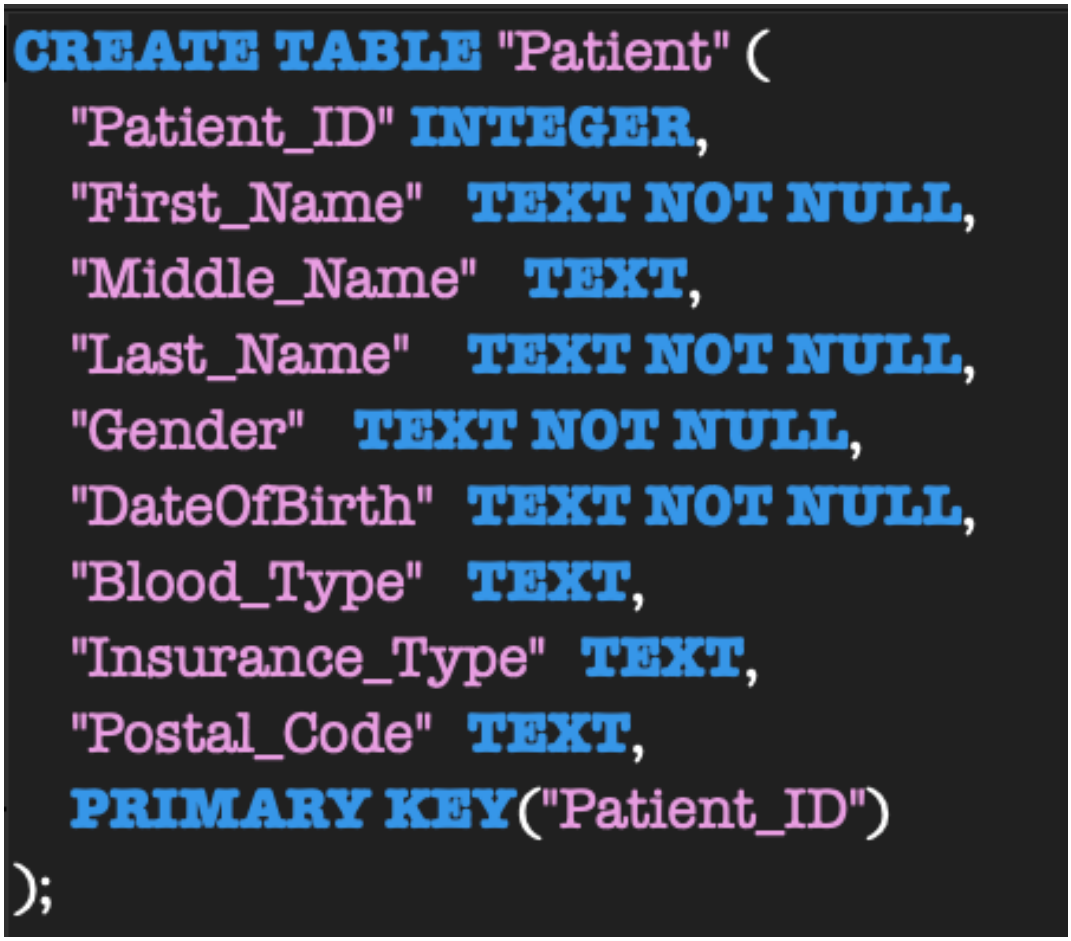
Patient Data Generation via Script

```python
def generate_patient_data(num_patients=500):

    np.random.seed(42)

    patient_ids = [i for i in range(100000, 100000 + num_patients)]

    uk_postcodes = [
        "SW1A 1AA", "EC1A 1BB", "W1A 0AX", "M1 1AE", "B33 8TH",
        "CR2 6XH", "DN55 1PT", "GIR 0AA", "L1 8JQ", "RH10 1AA"
    ]

    current_year = datetime.now().year
    birth_years = np.random.randint(current_year - 90, current_year - 18, num_patients)

    birth_dates = [datetime(year, np.random.randint(1, 13), np.random.randint(1, 29)) for year in birth_years]

    first_names = ["John", "Emily", "Michael", "Sarah", "David", "Jessica", "James", "Emma", "Robert", "Olivia",
                   "Daniel", "Sophia", "Matthew", "Isabella", "Andrew", "Mia", "Ethan", "Charlotte", "Joseph", "Amelia"]

    middle_names = ["Alexander", "Marie", "William", "Grace", "Edward", "Louise", "Henry", "Rose", "Joseph", "Anne",
                    "Thomas", "Victoria", "George", "Elizabeth", "Samuel", "Nicole", "Christopher", "Catherine", "Nathan", "Eleanor"]

    last_names = ["Smith", "Johnson", "Brown", "Williams", "Jones", "Davis", "Miller", "Wilson", "Anderson", "Taylor",
                  "Thomas", "Harris", "Martin", "Thompson", "White", "Garcia", "Martinez", "Clark", "Rodriguez", "Lewis"]

    first_name_list = [random.choice(first_names) for _ in range(num_patients)]
    middle_name_list = [random.choice(middle_names) if random.random() > 0.6 else "" for _ in range(num_patients)]
    last_name_list = [random.choice(last_names) for _ in range(num_patients)]

    return pd.DataFrame({
        "Patient_ID": patient_ids,
        "First_Name": first_name_list,
        "Middle_Name": middle_name_list,
        "Last_Name": last_name_list,
        "Gender": np.random.choice(["Male", "Female", "Other"], num_patients),
        "DateOfBirth": birth_dates,
        "Blood_Type": np.random.choice(["A+", "A-", "B+", "B-", "AB+", "AB-", "O+", "O-"], num_patients),
        "Insurance_Type": np.random.choice(["Private", "Public", "None"], num_patients),
        "Postal_Code": np.random.choice(uk_postcodes, num_patients)
    })
```

1. **Random Seed:** Ensures reproducibility by generating the same dataset every time.
2. **Patient_ID:** Sequential unique IDs starting from 100000 to prevent duplication.
3. **Postal Code:** Randomly assigned from a list of valid UK postal codes to simulate geographic distribution.
4. **Birth Year:** Randomly generated to ensure patients are between **18 and 90 years old**.
5. **Date of Birth:** A combination of **random month and day**, ensuring valid birth dates.
6. **First Name:** Randomly selected from a list of common UK first names.
7. **Middle Name:** 40% chance of being assigned; otherwise, left empty for realism.
8. **Last Name:** Randomly chosen from a list of common UK surnames.
9. **Gender:** Assigned randomly as **Male, Female, or Other** for inclusivity.
10. **Blood Type:** Randomly chosen from the 8 standard blood groups (A+, B-, O+, etc.).
11. **Insurance Type:** Assigned as **Private, Public, or None**, reflecting real-world coverage.
12. **Final DataFrame:** Compiles all generated data into a structured, tabular format for analysis.

This logic ensures **realistic, diverse, and structured patient data** for healthcare applications.

Patient Table (Patient Demographics and Health Information)



```
CREATE TABLE "Patient" (
    "Patient_ID" INTEGER,
    "First_Name"   TEXT NOT NULL,
    "Middle_Name"   TEXT,
    "Last_Name"   TEXT NOT NULL,
    "Gender"   TEXT NOT NULL,
    "DateOfBirth"  TEXT NOT NULL,
    "Blood_Type"   TEXT,
    "Insurance_Type"  TEXT,
    "Postal_Code"  TEXT,
    PRIMARY KEY("Patient_ID")
);
```

ThePatient Table Schema (Summary)

13. **Patient_ID (PK):** Unique identifier for each patient.
14. **First_Name, Last_Name:** Required patient name fields.
15. **Middle_Name:** Optional middle name.
16. **Gender:** Specifies patient's gender (**Male, Female, Other**).
17. **DateOfBirth:** Patient's birth date for age tracking.
18. **Blood_Type:** Records blood group (**A+, B-, O+**, etc.).
19. **Insurance_Type:** Defines insurance coverage (**Private, Public, None**).
20. **Postal_Code:** Represents patient's geographical location.

Constraints:

- **PK:** Patient_ID ensures each record is unique.
- **NOT NULL:** Required fields (**First Name, Last Name, Gender, DOB**) ensure complete records.

This schema ensures **structured, reliable patient data storage** for hospital systems.

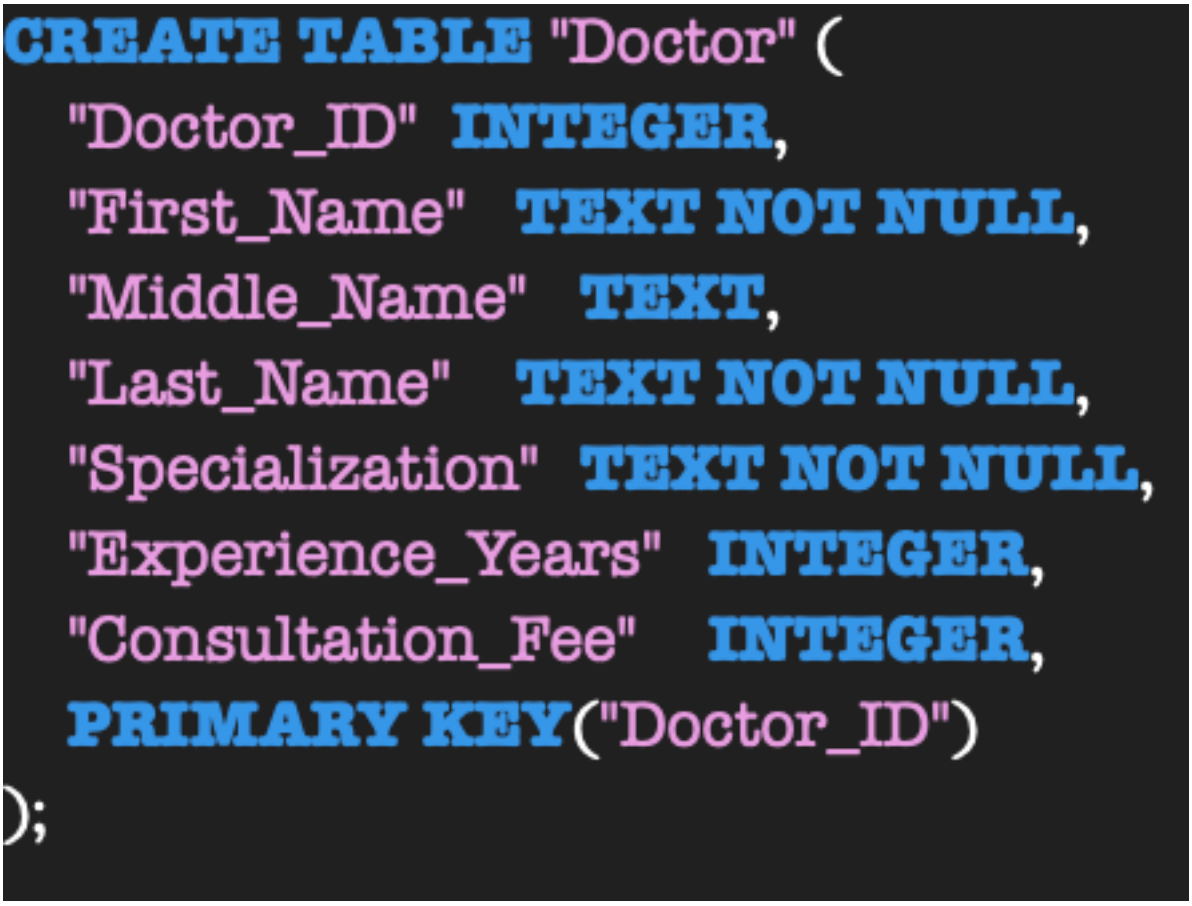
Doctor Data Generation via Script

```python
def generate_doctor_data(num_doctors=100):

    np.random.seed(42)

    doctor_ids = [i for i in range(200000, 200000 + num_doctors)]

    specializations = [
        "Cardiologist", "Neurologist", "General Physician", "Pediatrician",
        "Surgeon", "Dermatologist", "Oncologist", "Orthopedic", "Psychiatrist", "ENT Specialist"
    ]

    first_names = ["John", "Emily", "Michael", "Sarah", "David", "Jessica", "James", "Emma", "Robert", "Olivia",
                   "Daniel", "Sophia", "Matthew", "Isabella", "Andrew", "Mia", "Ethan", "Charlotte", "Joseph", "Amelia"]

    middle_names = ["Alexander", "Marie", "William", "Grace", "Edward", "Louise", "Henry", "Rose", "Joseph", "Anne",
                    "Thomas", "Victoria", "George", "Elizabeth", "Samuel", "Nicole", "Christopher", "Catherine", "Nathan", "Eleanor"]

    last_names = ["Smith", "Johnson", "Brown", "Williams", "Jones", "Davis", "Miller", "Wilson", "Anderson", "Taylor",
                  "Thomas", "Harris", "Martin", "Thompson", "White", "Garcia", "Martinez", "Clark", "Rodriguez", "Lewis"]

    first_name_list = [random.choice(first_names) for _ in range(num_doctors)]
    middle_name_list = [random.choice(middle_names) if random.random() > 0.6 else "" for _ in range(num_doctors)]
    last_name_list = [random.choice(last_names) for _ in range(num_doctors)]

    return pd.DataFrame({
        "Doctor_ID": doctor_ids,
        "First_Name": first_name_list,
        "Middle_Name": middle_name_list,
        "Last_Name": last_name_list,
        "Specialization": np.random.choice(specializations, num_doctors),
        "Experience_Years": np.random.randint(1, 40, num_doctors),
        "Consultation_Fee": np.random.choice(range(10, 210, 10), num_doctors)
    })
```

1. **Function Definition:** Defines generate_doctor_data(num_doctors=100), which generates **100 doctors by default**.
2. **Random Seed:** Ensures **consistent random outputs** each time the function is run.
3. **Doctor_IDs:** Assigns **unique IDs starting from 200000** to avoid duplication.
4. **Specializations:** Randomly selects a **medical specialty** from a predefined list (e.g., **Cardiologist, Surgeon, Pediatrician**).
5. **First Name:** Randomly picks **first names** from a list of **common UK names**.
6. **Middle Name:** 60% probability of being empty; otherwise, randomly selected.
7. **Last Name:** Randomly chosen from a **list of common UK surnames**.
8. **Experience Years:** Randomly assigned **between 1 and 40 years** to reflect varying experience levels.
9. **Consultation Fee:** Chosen from **multiples of 10 (between £10 and £200)**, representing real-world fee structures.
10. **Final DataFrame:** Stores all generated doctor details in a **structured tabular format** for easy processing.

This logic ensures **realistic doctor data generation**, covering **specializations, experience, and consultation fees**, making it useful for hospital management and analysis.

Doctor Table (Medical Professionals and Specializations)



```
CREATE TABLE "Doctor" (
    "Doctor_ID"  INTEGER,
    "First_Name"  TEXT NOT NULL,
    "Middle_Name"  TEXT,
    "Last_Name"  TEXT NOT NULL,
    "Specialization"  TEXT NOT NULL,
    "Experience_Years"  INTEGER,
    "Consultation_Fee"  INTEGER,
    PRIMARY KEY("Doctor_ID")
);
```

Doctor Table Schema (Summary)

1. **Doctor_ID (PK):** Unique identifier for each doctor.
2. **First_Name, Last_Name:** Required fields for doctor's name.
3. **Middle_Name:** Optional middle name.
4. **Specialization:** Defines the doctor's medical field (**Cardiologist, Surgeon, etc.**).
5. **Experience_Years:** Number of years the doctor has practiced.
6. **Consultation_Fee:** The charge for a consultation.

Constraints:

- **PK:** Doctor_ID ensures unique doctor identification.
- **NOT NULL:** Essential fields (**First Name, Last Name, Specialization**) are required for valid records.

This schema ensures **efficient doctor management** in the hospital database.

Appointment Data Generation via Script

```python
def generate_appointment_data(num_appointments=1000, patient_ids=None, doctor_ids=None, num_duplicates=10):

    np.random.seed(42)

    appointment_ids = [i for i in range(300000, 300000 + num_appointments)]

    df = pd.DataFrame({
        "Appointment_ID": appointment_ids,
        "Patient_ID": np.random.choice(patient_ids, num_appointments, replace=True),
        "Doctor_ID": np.random.choice(doctor_ids, num_appointments, replace=True),
        "Severity_Level": np.random.choice(["Mild", "Moderate", "Severe"], num_appointments, p=[0.5, 0.3, 0.2]),
        "Duration_Minutes": 20,  # Fixed duration for all appointments
        "Cost": np.random.choice(range(10, 210, 10), num_appointments),  # Cost in multiples of 10
        "Payment_Status": np.random.choice(["Paid", "Pending", "Not Required"], num_appointments),
        "Consultation_Type": np.random.choice(["In-Person", "Telemedicine"], num_appointments)
    })

    appointment_dates = np.random.choice(pd.date_range("2023-01-01", "2024-12-31", freq="D"), num_appointments)
    appointment_dates = pd.Series(pd.to_datetime(appointment_dates)).dt.strftime("%Y-%m-%d")

    time_slots = pd.date_range("09:00", "16:40", freq="20min").time  # Last slot starts at 16:40
    start_times = np.random.choice(time_slots, num_appointments)

    df["ScheduleTime_Start"] = [datetime.combine(pd.to_datetime(date).date(), time) for date, time in zip(appointment_dates, start_times)]
    df["ScheduleTime_End"] = df["ScheduleTime_Start"] + timedelta(minutes=20)
    df["ScheduleTime_Start"] = df["ScheduleTime_Start"].dt.strftime("%Y-%m-%d %H:%M:%S")
    df["ScheduleTime_End"] = df["ScheduleTime_End"].dt.strftime("%Y-%m-%d %H:%M:%S")

    duplicate_records = df.sample(num_duplicates).copy()
    duplicate_records["Appointment_ID"] = range(300000 + num_appointments, 300000 + num_appointments + num_duplicates)

    df = pd.concat([df, duplicate_records], ignore_index=True)

    return df
```

1. **Function Definition:** Defines generate_appointment_data(num_appointments=1000, patient_ids=None, doctor_ids=None, num_duplicates=10), which generates **1,000 appointments by default** with optional patient and doctor ID inputs.
2. **Random Seed:** Ensures **consistent random outputs** every time the function runs.
3. **Appointment_IDs:** Assigns **unique IDs starting from 300000** to prevent duplication.
4. **Patient_ID:** Randomly selects a **patient from the provided patient IDs** (ensuring valid references).
5. **Doctor_ID:** Randomly selects a **doctor from the provided doctor IDs** to establish doctor-patient mapping.
6. **Severity Level:** Randomly assigns **Mild, Moderate, or Severe**, with a probability distribution of **50%, 30%, and 20%** respectively.
7. **Duration Minutes:** Fixed at **20 minutes** for all appointments to simplify scheduling.
8. **Cost:** Randomly assigned in **multiples of 10 (between £10 and £200)** to reflect realistic medical service charges.
9. **Payment Status:** Randomly assigned as **Paid, Pending, or Not Required** to simulate financial transactions.
10. **Consultation Type:** Chosen randomly as **In-Person or Telemedicine** to account for both physical and virtual consultations.
11. **Appointment Date:** Randomly picked from a **date range between January 1, 2023, and December 31, 2024** to ensure scheduled appointments fall within a two-year period.
12. **Appointment Time:** Randomly selected from predefined **20-minute slots between 09:00 AM and 16:40 PM** to represent real hospital scheduling constraints.
13. **Start and End Time:**
    - The **start time** is determined by randomly assigning a time slot.
    - The **end time** is computed as **start time + 20 minutes** to maintain consistency.

14. **Duplicate Records:** A small number of **duplicate records (10)** are intentionally added to test data cleaning mechanisms.
15. **Final DataFrame:** Compiles all generated **appointment details into a structured format**, making it suitable for hospital management and analytics.

This structured logic ensures **realistic, well-distributed, and analyzable appointment data**, integrating **patients, doctors, and financial transactions** effectively.

Appointment Table (Medical Consultations & Transactions)

```
CREATE TABLE "Appointment" (
  "Appointment_ID" INTEGER,
  "Patient_ID" INTEGER,
  "Doctor_ID" INTEGER,
  "Severity_Level" TEXT NOT NULL,
  "Duration_Minutes" INTEGER,
  "Cost" INTEGER,
  "Payment_Status" TEXT NOT NULL,
  "Consultation_Type" TEXT NOT NULL,
  "ScheduleTime_Start" TEXT NOT NULL,
  "ScheduleTime_End" TEXT NOT NULL,
  PRIMARY KEY("Appointment_ID"),
  FOREIGN KEY("Doctor_ID") REFERENCES "Doctor"("Doctor_ID") ON DELETE CASCADE,
  FOREIGN KEY("Patient_ID") REFERENCES "Patient"("Patient_ID") ON DELETE CASCADE
);
```

1. **Appointment_ID (PK):** Unique identifier for each appointment.
2. **Patient_ID (FK):** References the **Patient** table (linked patient).
3. **Doctor_ID (FK):** References the **Doctor** table (assigned doctor).
4. **Severity_Level:** Defines case severity (**Mild, Moderate, Severe**).
5. **Duration_Minutes:** Appointment length in minutes.
6. **Cost:** Fee charged for the appointment.
7. **Payment_Status:** Indicates if payment is **Paid, Pending, or Not Required**.
8. **Consultation_Type:** Specifies **In-Person or Telemedicine**.
9. **ScheduleTime_Start:** Start time of the appointment.
10. **ScheduleTime_End:** End time (calculated from start time + duration).

Constraints:

- **PK:** Appointment_ID ensures uniqueness.
- **FK:** Doctor_ID & Patient_ID enforce valid references.
- **ON DELETE CASCADE:** Deletes appointments if the linked patient or doctor is removed.

This schema ensures **efficient hospital scheduling, financial tracking, and data integrity**.

SQLite Data Create and Insert Table Script:

```
def insert_into_sqlite(df_patient, df_doctor, df_appointment):
    try:
        conn = sqlite3.connect("hospital.db", timeout=10)
        cursor = conn.cursor()

        cursor.execute("PRAGMA journal_mode=WAL;")
        cursor.execute("PRAGMA synchronous = NORMAL;")
        cursor.execute("PRAGMA temp_store = MEMORY;")
        cursor.execute("PRAGMA foreign_keys = ON;")  # Ensure foreign key constraints are enforced
        # Drop and recreate the Appointment table to include new columns
        cursor.execute("DROP TABLE IF EXISTS Appointment")
        cursor.execute("""
        CREATE TABLE IF NOT EXISTS Patient (
            Patient_ID INTEGER PRIMARY KEY,
            First_Name TEXT NOT NULL,
            Middle_Name TEXT,
            Last_Name TEXT NOT NULL,
            Gender TEXT NOT NULL,
            DateOfBirth TEXT NOT NULL,
            Blood_Type TEXT,
            Insurance_Type TEXT,
            Postal_Code TEXT
        )""")
        cursor.execute("""
        CREATE TABLE IF NOT EXISTS Doctor (
            Doctor_ID INTEGER PRIMARY KEY,
            First_Name TEXT NOT NULL,
            Middle_Name TEXT,
            Last_Name TEXT NOT NULL,
            Specialization TEXT NOT NULL,
            Experience_Years INTEGER,
            Consultation_Fee INTEGER
        )""")
        cursor.execute("""
        CREATE TABLE IF NOT EXISTS Appointment (
            Appointment_ID INTEGER PRIMARY KEY,
            Patient_ID INTEGER,
            Doctor_ID INTEGER,
            Severity_Level TEXT NOT NULL,
            Duration_Minutes INTEGER,
            Cost INTEGER,
            Payment_Status TEXT NOT NULL,
            Consultation_Type TEXT NOT NULL,
            ScheduleTime_Start TEXT NOT NULL,
            ScheduleTime_End TEXT NOT NULL,
            FOREIGN KEY (Patient_ID) REFERENCES Patient(Patient_ID) ON DELETE CASCADE,
            FOREIGN KEY (Doctor_ID) REFERENCES Doctor(Doctor_ID) ON DELETE CASCADE
        )""")
    # Convert datetime columns to string format before inserting into SQLite
    df_patient["DateOfBirth"] = df_patient["DateOfBirth"].astype(str)
    df_appointment["ScheduleTime_Start"] = df_appointment["ScheduleTime_Start"].astype(str)
    df_appointment["ScheduleTime_End"] = df_appointment["ScheduleTime_End"].astype(str)
    df_patient_records = df_patient.values.tolist()
    df_doctor_records = df_doctor.values.tolist()
    df_appointment_records = df_appointment.values.tolist()
    conn.execute("BEGIN TRANSACTION")

    if df_patient_records:
        cursor.executemany("INSERT OR IGNORE INTO Patient VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)", df_patient_records)

    if df_doctor_records:
        cursor.executemany("INSERT OR IGNORE INTO Doctor VALUES (?, ?, ?, ?, ?, ?, ?)", df_doctor_records)

    if df_appointment_records:
        cursor.executemany("INSERT OR IGNORE INTO Appointment VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)", df_appointment_records)

    conn.commit()  # Commit once to avoid table locking

    print("Data inserted into SQLite successfully.")

except sqlite3.Error as e:
    conn.rollback()  # Rollback changes if an error occurs
    print(f"SQLite Error: {e}")

finally:
    conn.close()  # Ensure connection is closed properly
```

1. Function Definition

- `insert_into_sqlite(df_patient, df_doctor, df_appointment)` initializes database setup.

2. Database Connection

- Connects to `"hospital.db"` with a **10-second timeout**.
- Uses a cursor for SQL execution.

3. Performance Optimizations (`PRAGMA`)

- **WAL mode** for better concurrency.
- **Foreign keys enabled** for referential integrity.
- **Memory storage for temporary tables** to boost speed.

4. Drop Appointment Table

- Deletes `Appointment` table **if it exists** for schema modifications.

5. Create Tables (if not exists)

- **`Patient` Table:** Stores patient details (`Patient_ID`, `Name`, `Gender`, `DOB`, etc.).
- **`Doctor` Table:** Holds doctor info (`Doctor_ID`, `Specialization`, `Experience`, etc.).
- **`Appointment` Table:** Links patients & doctors with timestamps and fees.

6. Foreign Key Relationships

- **`Patient_ID` → Patient Table** (`ON DELETE CASCADE`).
- **`Doctor_ID` → Doctor Table** (`ON DELETE CASCADE`).

7. Start Transaction

- Executes `"BEGIN TRANSACTION"` to ensure atomicity and **avoid partial insertions**.

8. Insert Data into Tables

- Uses **`INSERT OR IGNORE`** to insert records while **preventing duplicates**.
- Inserts into:
  - **Patient Table**
  - **Doctor Table**
  - **Appointment Table**

9. Commit Transaction

- Calls `conn.commit()` to **finalize all inserts at once**, avoiding table locking.

10. Error Handling

- Catches `sqlite3.Error` and **rolls back** the transaction in case of failure.

11. Close Connection

- Ensures the **database connection is properly closed** after execution.

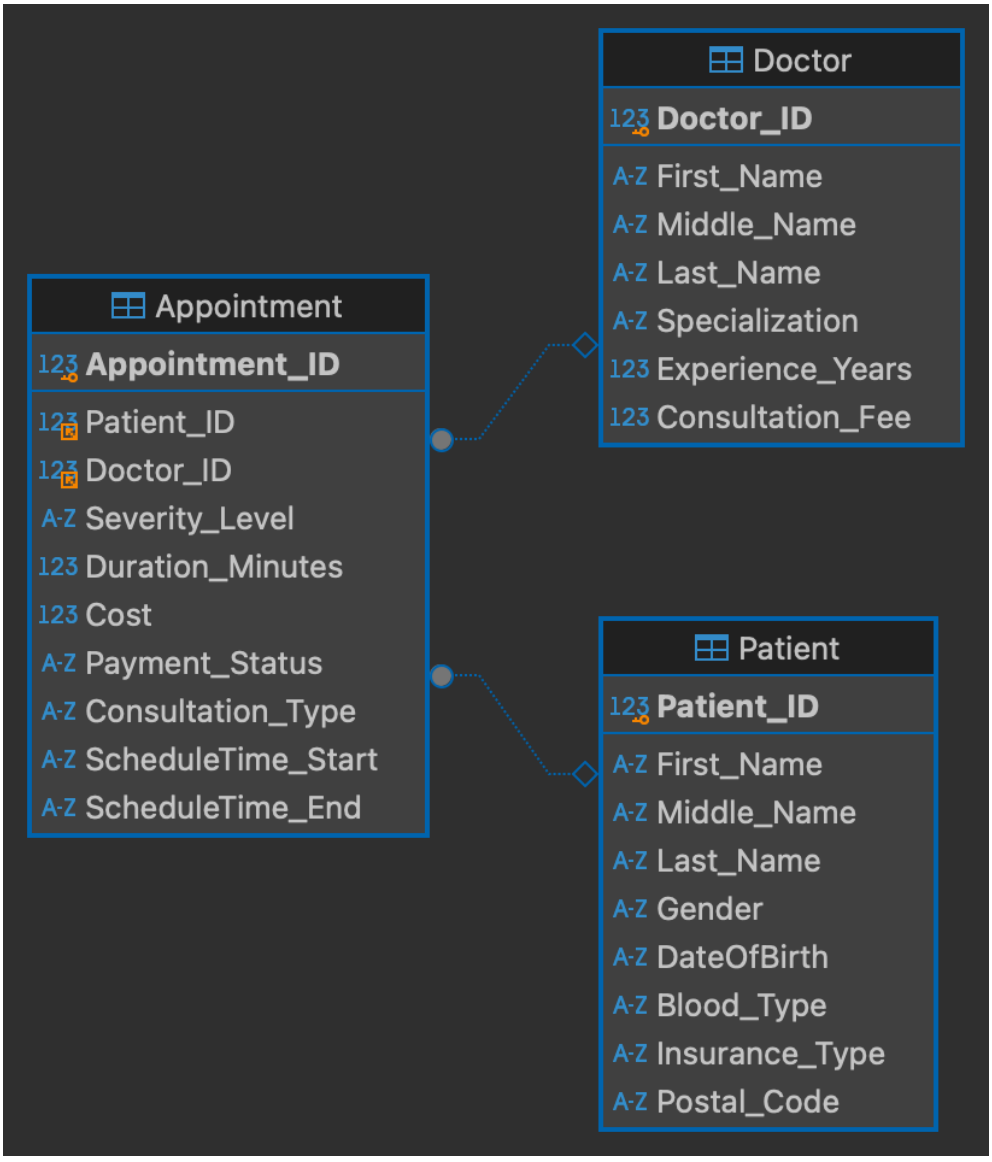
Database Relationships (Entity-Relationship Diagram)

One-to-Many Relationships:

- **Each Patient** can have **multiple Appointments** over time.
- **Each Doctor** can be assigned to **multiple Appointments** with different patients.

Many-to-Many Relationship:

- Since an **Appointment** links both a **Patient** and a **Doctor**, this creates a **many-to-many** relationship between **Patients and Doctors** (mediated through the Appointment table).



This structure ensures **efficient appointment scheduling, tracking patient history, and maintaining doctor availability** within the hospital database.

Hospital Database Data Anomalies Report

1. Duplicate Appointments Based on Patient ID and Schedule Time

- **Total Duplicate Appointments**: **11**
- **Percentage of Duplicate Records**: **1.10%**
- **Issue**:
  Patients have multiple appointments scheduled at the same time, which may indicate **data entry errors or double booking issues**.

Duplicate Generation Script:

```
#   create duplicate records
def introduce_duplicates(df, num_duplicates=10,num_appointments=1000):

    duplicate_records = df.sample(num_duplicates).copy()
    duplicate_records["Appointment_ID"] = range(300000 + num_appointments, 300000 + num_appointments + num_duplicates)

    return
```

SQL Query to Detect Duplicates:

SELECT Patient_ID, ScheduleTime_Start, COUNT(*) AS Duplicate_Count
FROM Appointment
GROUP BY Patient_ID, ScheduleTime_Start
HAVING COUNT(*) > 1;

Results:

Patient_ID	ScheduleTime_Start	Duplicate_Count
100052	2024-03-14 11:20:00	2
100143	2023-12-01 16:20:00	2
100188	2023-07-08 14:20:00	2
100223	2024-11-21 12:40:00	2
100245	2024-09-06 16:40:00	2
100252	nan	2
100303	2024-03-16 12:00:00	2
100351	2024-09-22 10:00:00	2
100359	2023-12-30 12:00:00	2
100412	2023-07-03 15:00:00	2
100484	2024-06-07 15:20:00	2

Fix:

- Implement **unique constraints** to prevent duplicate appointment bookings.
- Introduce **pre-booking validation** to detect scheduling conflicts before confirming appointments.

2. Empty Entries in Schedule Data

- **Total Empty Schedule Records**: **10**
- **Percentage of Empty Schedule Entries**: **1.00%**
- **Issue**:
  Missing values in **ScheduleTime_Start** and **ScheduleTime_End** fields can lead to **operational issues** in managing appointments, such as scheduling conflicts and system errors.

Null Values Generation:

```
# introduce null values
def introduce_nulls(df, num_nulls=10, null_columns=None):
    null_indices_date = np.random.choice(df.index, 10, replace=False)
    df.loc[null_indices_date, "ScheduleTime_Start"] = np.nan
    df.loc[null_indices_date, "ScheduleTime_End"] = np.nan

    null_indices_severity = np.random.choice(df.index, 10, replace=False)
    df.loc[null_indices_severity, "Severity_Level"] = np.nan

    return df
```

SQL Query to Detect Empty Schedule Entries:

SELECT COUNT(*) AS Empty_Schedule_Records
FROM Appointment
WHERE ScheduleTime_Start ==”nan" OR ScheduleTime_End ==“nan";

Empty_Schedule_Records
10

Fix:

- Ensure all appointment records have **valid start and end times** before storing them in the database.
- Set up **data validation rules** to prevent incomplete records from being entered into the system.
- Run periodic **data quality checks** to identify and correct missing schedule entries.

Hospital Database Analysis

1. Total Appointments Per Doctor

- Shows the number of appointments handled by each doctor.
- Helps in identifying the busiest doctors and workload distribution.

SQL Query:

```
SELECT d.Doctor_ID, d.First_Name, d.Last_Name, d.Specialization,
COUNT(a.Appointment_ID) AS Total_Appointments
FROM Doctor d
JOIN Appointment a ON d.Doctor_ID = a.Doctor_ID
GROUP BY d.Doctor_ID, d.First_Name, d.Last_Name, d.Specialization
ORDER BY Total_Appointments DESC;
```

Results:

Doctor_ID	First_Name	Last_Name	Specialization	Total_Appointments
200025	Robert	Williams	Oncologist	19
200039	Emily	Williams	Surgeon	18
200050	Olivia	Brown	Orthopedic	18
200097	Robert	Taylor	General Physician	18
200004	James	Brown	Oncologist	17
200058	Sarah	Wilson	Neurologist	17
200061	Robert	Williams	ENT Specialist	17

2. Average Cost Per Severity Level

- Displays the **average appointment cost** categorized by severity (**Mild, Moderate, Severe**).
- Helps in assessing whether appointment costs vary significantly based on severity.

SQL Query:

```
SELECT Severity_Level, AVG(Cost) AS Average_Cost
FROM Appointment
GROUP BY Severity_Level;
```

Results:

Severity_Level	Average_Cost
Mild	106.30831643002
Moderate	104.354838709677
Severe	103.401015228426

3. Appointment Distribution by Consultation Type

- Compares the number of **In-Person** vs **Telemedicine** appointments.
- Provides insights into **patient preferences and hospital service demand**.

SQL Query:

```
SELECT Consultation_Type, COUNT(*) AS Appointment_Count
FROM Appointment
GROUP BY Consultation_Type;
```

Results:

Consultation_Type	Appointment_Count
In-Person	488
Telemedicine	512

4. Patients Count by Insurance Type

- Identifies the distribution of patients based on **Private, Public, or No Insurance**.
- Useful for understanding **financial coverage trends among hospital patients**.

SQL Query:

```
SELECT Insurance_Type, COUNT(*) AS Patient_Count
FROM Patient
GROUP BY Insurance_Type;
```

Results:

Insurance_Type	Patient_Count
None	155
Private	173
Public	172

Justification of Table Choices

- **Three Core Tables**: The **Patient, Doctor, and Appointment** tables are well-justified as they form the foundation of a hospital management system.
- **Entity-Relationship Design**: The report outlines how each table is interconnected using **Primary Keys (PKs) and Foreign Keys (FKs)**, ensuring **data integrity** and logical relationships.
- **Realistic Data Simulation**:
  - **Patient Data**: Covers demographics, insurance, and medical details.
  - **Doctor Data**: Includes specialization, experience, and fees.
  - **Appointment Data**: Captures scheduling, consultation types, and financial transactions.

Ethics and Data Privacy Discussion

- **Inclusivity in Patient Data**: Gender is recorded as **Male, Female, or Other**, ensuring fair representation.
- **Data Privacy Considerations**:
  - **Randomized Data Generation**: Uses synthetic data to avoid handling real patient records, aligning with data privacy best practices.
  - **Foreign Key Constraints**: Enforces referential integrity, preventing unauthorized deletions that could lead to inconsistent records.
  - **Handling Duplicate and Null Data**: Identifies duplicate bookings and empty schedule fields, suggesting validation rules to prevent errors and maintain data accuracy.

Overall Strengths

- **Efficient Database Design:** Ensures structured storage, fast querying, and easy management.
- **Privacy-Conscious Approach:** Uses synthetic data, integrity constraints, and validation mechanisms.
- **Data Anomaly Detection & Fixes:** Highlights potential data quality issues and their resolutions.
-