# Healthcare Patient Analytics & ETL Star Schema Report

-- Sayan Das

## 1. Generating and Loading CSV Data

I started by generating synthetic data for six main entities:  
- **Patients** (patient demographics)  
- **Doctors** (doctor details)  
- **Admissions** (patient admissions)  
- **Vitals** (patient vitals during hospital stays)  
- **Treatments** (treatments and medications)  
- **Readmission\_Risk** (predicted readmission risk)

To ensure adequate variety, I created **at least 10 rows** per table. For example, in the **Patients** table, I randomly generated first names, last names, dates of birth, and phone numbers. In the **Doctors** table, I randomly generated specialties and contact numbers. Here is a **Python code snippet** showing how I generated some of the sample data:

import names

def random\_date(start\_year=2020, end\_year=2025):  
 """Generate a random date between start\_year and end\_year."""  
 start\_date = datetime(start\_year, 1, 1)  
 end\_date = datetime(end\_year, 12, 31)  
 delta = end\_date - start\_date  
 random\_days = random.randrange(delta.days)  
 return start\_date + timedelta(days=random\_days)  
  
def random\_phone():  
 """Generate a random 10-digit phone number as a string."""  
 return str(random.randint(10\*\*9, 10\*\*10 - 1))  
  
def random\_diagnosis():  
 return random.choice(["Pneumonia", "Hypertension", "Asthma Attack", "Diabetes Complications", "Heart Failure", "Sepsis", "Kidney Stones", "Migraine", "COVID-19", "Fracture"])  
  
def random\_specialization():  
 return random.choice(["Cardiologist", "Pulmonologist", "General Physician", "Neurologist", "Orthopedic", "Endocrinologist", "Gastroenterologist"])  
  
def random\_chronic\_condition():  
 return random.choice(["None", "Hypertension", "Asthma", "Diabetes", "Heart Disease", "None", "None"])  
  
# 1.1 Patients (>= 10 rows)  
num\_patients = 10  
patient\_ids = list(range(101, 101 + num\_patients))  
patients\_data = []  
for pid in patient\_ids:  
 patients\_data.append([  
 pid,  
 names.get\_first\_name(),  
 names.get\_last\_name(),  
 random\_date(1950, 2000).date(), # dob  
 random.choice(["Male", "Female"]),  
 random\_phone(),  
 f"{random.randint(100,999)} Main St",  
 random\_chronic\_condition()  
 ])  
  
patients\_df = pd.DataFrame(patients\_data, columns=[  
 "patient\_id", "first\_name", "last\_name", "dob", "gender", "contact\_no", "address", "chronic\_conditions"  
])  
  
# 1.2 Doctors (>= 10 rows)  
num\_doctors = 10  
doctor\_ids = list(range(301, 301 + num\_doctors))  
doctors\_data = []  
for did in doctor\_ids:  
 doctors\_data.append([  
 did,  
 names.get\_first\_name(),  
 names.get\_last\_name(),  
 random\_specialization(),  
 random\_phone()  
 ])  
  
doctors\_df = pd.DataFrame(doctors\_data, columns=[  
 "doctor\_id", "first\_name", "last\_name", "specialization", "contact\_no"  
])  
  
# 1.3 Admissions (>= 10 rows)  
# We will ensure each admission references an existing patient & doctor  
num\_admissions = 10  
admission\_ids = list(range(2001, 2001 + num\_admissions))  
admissions\_data = []  
for aid in admission\_ids:  
 patient\_id = random.choice(patient\_ids)  
 doctor\_id = random.choice(doctor\_ids)  
 admission\_date = random\_date(2024, 2025)  
 # Some admissions have not been discharged yet  
 discharge\_date = admission\_date + timedelta(days=random.randint(1, 10)) if random.random() > 0.3 else None  
 diagnosis = random\_diagnosis()  
 room\_no = random.choice(["A101","A102","B210","C305","B405","ICU1","ICU2","D110","D120","E201"])  
 admissions\_data.append([  
 aid, patient\_id, admission\_date.date(),  
 discharge\_date.date() if discharge\_date else None,  
 diagnosis, doctor\_id, room\_no  
 ])  
  
admissions\_df = pd.DataFrame(admissions\_data, columns=[  
 "admission\_id", "patient\_id", "admission\_date", "discharge\_date",  
 "diagnosis", "doctor\_id", "room\_no"  
])  
  
# 1.4 Vitals (>= 10 rows)  
# Each vitals row references an existing admission  
num\_vitals = 10  
vital\_ids = list(range(5001, 5001 + num\_vitals))  
vitals\_data = []  
for vid in vital\_ids:  
 admission\_id = random.choice(admission\_ids)  
 # Just pick a random time near the admission\_date  
 base\_date = admissions\_df.loc[admissions\_df['admission\_id'] == admission\_id, 'admission\_date'].values[0]  
 # Convert base\_date to datetime  
 base\_datetime = pd.to\_datetime(base\_date)  
 recorded\_time = base\_datetime + timedelta(hours=random.randint(0, 100))  
 heart\_rate = random.randint(60, 120)  
 bp\_systolic = random.randint(100, 160)  
 bp\_diastolic = random.randint(70, 100)  
 blood\_pressure = f"{bp\_systolic}/{bp\_diastolic}"  
 oxygen\_level = random.randint(88, 100)  
 temperature = round(random.uniform(97.0, 103.0), 1)  
 vitals\_data.append([  
 vid, admission\_id, recorded\_time, heart\_rate, blood\_pressure, oxygen\_level, temperature  
 ])  
  
vitals\_df = pd.DataFrame(vitals\_data, columns=[  
 "vital\_id", "admission\_id", "recorded\_time", "heart\_rate",  
 "blood\_pressure", "oxygen\_level", "temperature"  
])  
  
# 1.5 Treatments (>= 10 rows)  
num\_treatments = 10  
treatment\_ids = list(range(7001, 7001 + num\_treatments))  
treatments\_data = []  
possible\_procedures = ["Nebulization", "Blood Pressure Monitoring", "ECG", "X-Ray",   
 "MRI Scan", "IV Fluid Therapy", "Physical Therapy", "Vaccination"]  
possible\_meds = ["Amoxicillin 500mg", "Prednisone 10mg", "Metoprolol 50mg", "Ibuprofen 400mg",  
 "Acetaminophen 500mg", "Atorvastatin 20mg", "Insulin 10units"]  
for tid in treatment\_ids:  
 admission\_id = random.choice(admission\_ids)  
 # approximate date of treatment around admission\_date  
 base\_date = admissions\_df.loc[admissions\_df['admission\_id'] == admission\_id, 'admission\_date'].values[0]  
 base\_datetime = pd.to\_datetime(base\_date)  
 treat\_date = base\_datetime + timedelta(days=random.randint(0, 5))  
 procedure = random.choice(possible\_procedures)  
 medication = random.choice(possible\_meds)  
 dosage = random.choice(["1x daily", "2x daily", "3x daily", "As needed"])  
 treatments\_data.append([  
 tid, admission\_id, treat\_date.date(), procedure, medication, dosage  
 ])  
  
treatments\_df = pd.DataFrame(treatments\_data, columns=[  
 "treatment\_id", "admission\_id", "treatment\_date", "procedure", "medication", "dosage"  
])  
  
# 1.6 Readmission\_Risk (>= 10 rows)  
# We'll keep a 1-to-1 relationship with admissions for demonstration  
risk\_ids = list(range(9001, 9001 + num\_admissions))  
risk\_data = []  
for i, aid in enumerate(admission\_ids):  
 pred\_date = admissions\_df.loc[admissions\_df['admission\_id'] == aid, 'admission\_date'].values[0]  
 pred\_date = pd.to\_datetime(pred\_date) + timedelta(days=random.randint(0,2))  
 risk\_score = round(random.uniform(0.2, 0.9), 2)  
 if risk\_score < 0.4:  
 risk\_level = "Low"  
 elif risk\_score < 0.7:  
 risk\_level = "Medium"  
 else:  
 risk\_level = "High"  
 risk\_data.append([  
 risk\_ids[i], aid, pred\_date.date(), risk\_score, risk\_level  
 ])  
  
risk\_df = pd.DataFrame(risk\_data, columns=[  
 "risk\_id", "admission\_id", "prediction\_date", "risk\_score", "risk\_level"  
])

Each DataFrame was then written out to a CSV file:

patients\_df.to\_csv("patients.csv", index=False)  
doctors\_df.to\_csv("doctors.csv", index=False)  
admissions\_df.to\_csv("admissions.csv", index=False)  
vitals\_df.to\_csv("vitals.csv", index=False)  
treatments\_df.to\_csv("treatments.csv", index=False)  
risk\_df.to\_csv("readmission\_risk.csv", index=False)  
  
print("Sample CSV files created with >= 10 rows each.")

## 2. Reading the CSV Files into Pandas

After generating the CSV files, I **extracted** the data into Pandas DataFrames. I used the following code to read each CSV:

patients = pd.read\_csv("patients.csv", parse\_dates=["dob"])  
doctors = pd.read\_csv("doctors.csv")  
admissions = pd.read\_csv("admissions.csv", parse\_dates=["admission\_date", "discharge\_date"])  
vitals = pd.read\_csv("vitals.csv", parse\_dates=["recorded\_time"])  
treatments = pd.read\_csv("treatments.csv", parse\_dates=["treatment\_date"])  
readmission\_risk = pd.read\_csv("readmission\_risk.csv", parse\_dates=["prediction\_date"])

I parsed date columns such as dob, admission\_date, discharge\_date, and recorded\_time to ensure they were stored in proper date/datetime formats.

## 3. Data Cleaning and Quality Checks

### 3.1 Primary Key Uniqueness

I verified that the **primary keys** (e.g., patient\_id, doctor\_id, admission\_id) were unique within their respective DataFrames. If duplicates existed, I dropped them:

def check\_and\_drop\_duplicates(df, pk\_col, table\_name):  
 dup\_count = df.duplicated(subset=[pk\_col]).sum()  
 if dup\_count > 0:  
 print(f"{table\_name}: Dropping {dup\_count} duplicate rows based on primary key {pk\_col}.")  
 df = df.drop\_duplicates(subset=[pk\_col])  
 return df  
  
patients = check\_and\_drop\_duplicates(patients, "patient\_id", "Patients")  
doctors = check\_and\_drop\_duplicates(doctors, "doctor\_id", "Doctors")  
admissions = check\_and\_drop\_duplicates(admissions, "admission\_id", "Admissions")  
vitals = check\_and\_drop\_duplicates(vitals, "vital\_id", "Vitals")  
treatments = check\_and\_drop\_duplicates(treatments, "treatment\_id", "Treatments")  
readmission\_risk = check\_and\_drop\_duplicates(readmission\_risk, "risk\_id", "Readmission\_Risk")

### 3.2 Not‐Null Checks

Certain columns, such as first\_name, last\_name, admission\_date, diagnosis, etc., are **NOT NULL** in the conceptual design. I dropped rows that had missing values in these columns:

def drop\_missing\_required(df, required\_cols, table\_name):  
 missing\_mask = df[required\_cols].isnull().any(axis=1)  
 missing\_count = missing\_mask.sum()  
 if missing\_count > 0:  
 print(f"{table\_name}: Dropping {missing\_count} rows with missing data in required columns {required\_cols}.")  
 df = df[~missing\_mask]  
 return df  
  
patients = drop\_missing\_required(patients, ["patient\_id", "first\_name", "last\_name", "dob", "gender", "contact\_no"], "Patients")  
doctors = drop\_missing\_required(doctors, ["doctor\_id", "first\_name", "last\_name", "specialization", "contact\_no"], "Doctors")  
admissions = drop\_missing\_required(admissions, ["admission\_id", "patient\_id", "admission\_date", "diagnosis", "doctor\_id"], "Admissions")  
vitals = drop\_missing\_required(vitals, ["vital\_id", "admission\_id", "recorded\_time", "heart\_rate", "blood\_pressure", "oxygen\_level", "temperature"], "Vitals")  
treatments = drop\_missing\_required(treatments, ["treatment\_id", "admission\_id", "treatment\_date", "medication"], "Treatments")  
readmission\_risk = drop\_missing\_required(readmission\_risk, ["risk\_id", "admission\_id", "prediction\_date", "risk\_score", "risk\_level"], "Readmission\_Risk")

### 3.3 Foreign Key Validation

I also enforced **foreign key constraints** to ensure that, for instance, every admission referenced a valid patient\_id and doctor\_id. Rows with invalid references were dropped:

# Admissions -> Patients  
invalid\_pat\_fk = ~admissions['patient\_id'].isin(patients['patient\_id'])  
if invalid\_pat\_fk.sum() > 0:  
 print(f"Admissions: Dropping {invalid\_pat\_fk.sum()} rows with invalid patient\_id.")  
 admissions = admissions[~invalid\_pat\_fk]  
  
# Admissions -> Doctors  
invalid\_doc\_fk = ~admissions['doctor\_id'].isin(doctors['doctor\_id'])  
if invalid\_doc\_fk.sum() > 0:  
 print(f"Admissions: Dropping {invalid\_doc\_fk.sum()} rows with invalid doctor\_id.")  
 admissions = admissions[~invalid\_doc\_fk]  
  
# Vitals -> Admissions  
invalid\_adm\_fk\_v = ~vitals['admission\_id'].isin(admissions['admission\_id'])  
if invalid\_adm\_fk\_v.sum() > 0:  
 print(f"Vitals: Dropping {invalid\_adm\_fk\_v.sum()} rows with invalid admission\_id.")  
 vitals = vitals[~invalid\_adm\_fk\_v]  
  
# Treatments -> Admissions  
invalid\_adm\_fk\_t = ~treatments['admission\_id'].isin(admissions['admission\_id'])  
if invalid\_adm\_fk\_t.sum() > 0:  
 print(f"Treatments: Dropping {invalid\_adm\_fk\_t.sum()} rows with invalid admission\_id.")  
 treatments = treatments[~invalid\_adm\_fk\_t]  
  
# Readmission\_Risk -> Admissions  
invalid\_adm\_fk\_r = ~readmission\_risk['admission\_id'].isin(admissions['admission\_id'])  
if invalid\_adm\_fk\_r.sum() > 0:  
 print(f"Readmission\_Risk: Dropping {invalid\_adm\_fk\_r.sum()} rows with invalid admission\_id.")  
 readmission\_risk = readmission\_risk[~invalid\_adm\_fk\_r]  
  
print("Data cleaning & validation complete.")

## 4. Star Schema Construction

### 4.1 Dimension Tables with Surrogate Keys

I created two dimension tables: **dim\_patients** and **dim\_doctors**. Rather than using the natural IDs (patient\_id, doctor\_id) as the primary keys, I introduced **surrogate keys** (patient\_key, doctor\_key). This approach is beneficial for slowly changing dimensions and ensures consistent referencing over time.

dim\_patients = patients.copy()  
dim\_patients["patient\_key"] = range(1, len(dim\_patients) + 1)  
dim\_patients = dim\_patients[[  
 "patient\_key", "patient\_id", "first\_name", "last\_name", "dob", "gender", "contact\_no", "address", "chronic\_conditions"  
]]

Similarly for doctors:

dim\_doctors = doctors.copy()  
dim\_doctors["doctor\_key"] = range(1, len(dim\_doctors) + 1)  
dim\_doctors = dim\_doctors[[  
 "doctor\_key", "doctor\_id", "first\_name", "last\_name", "specialization", "contact\_no"  
]]

### 4.2 Fact Tables

1. **FactAdmissions**: I merged the **Admissions** table with the **Readmission\_Risk** table. This single table contains columns such as admission\_date, discharge\_date, diagnosis, plus risk\_score and risk\_level. I then replaced patient\_id and doctor\_id with the new surrogate keys from dim\_patients and dim\_doctors.

* fact\_admissions = admissions.merge(readmission\_risk, on="admission\_id", how="left")  
  fact\_admissions = fact\_admissions.merge(dim\_patients[["patient\_id", "patient\_key"]], on="patient\_id", how="left")  
  fact\_admissions = fact\_admissions.merge(dim\_doctors[["doctor\_id", "doctor\_key"]], on="doctor\_id", how="left")  
  fact\_admissions.drop(["patient\_id", "doctor\_id"], axis=1, inplace=True)

1. **FactVitals**: Stores vitals for each admission (heart\_rate, blood\_pressure, etc.), keyed by admission\_id.
2. **FactTreatments**: Stores treatment and medication data for each admission, also keyed by admission\_id.

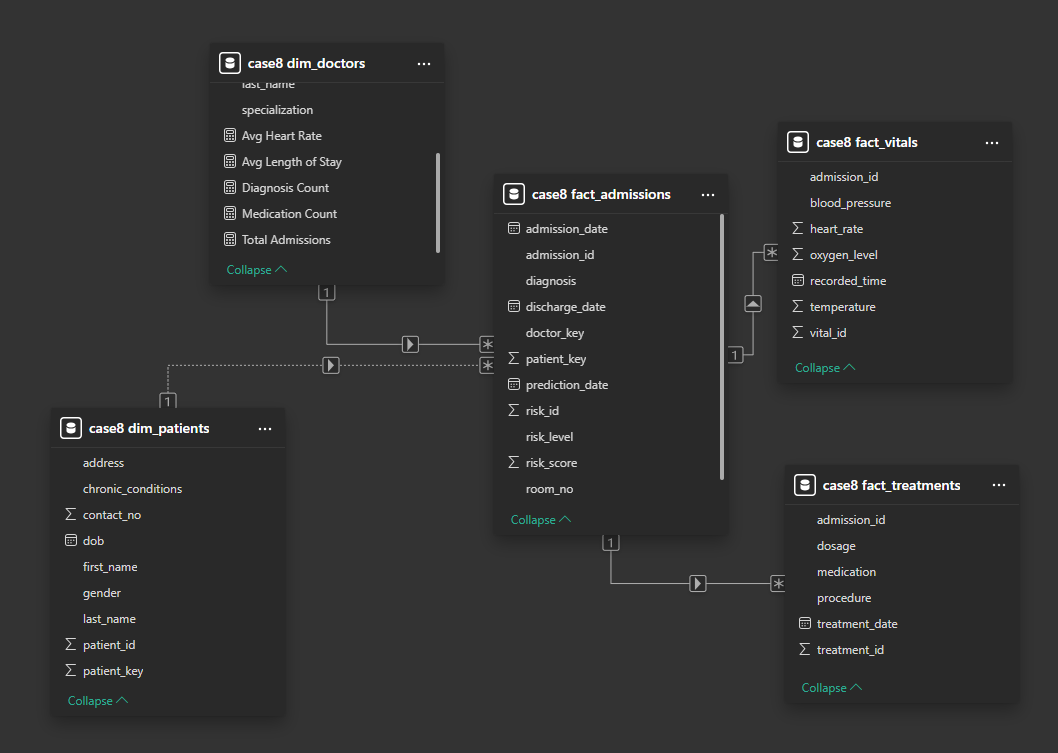
## 5. Loading the Data into MySQL

Finally, I **loaded** the dimension and fact tables into MySQL using **SQLAlchemy**. Below is an example of how I wrote each table to the database:

username = 'root'  
password = '12345'  
host = 'localhost'  
port = '3306'  
database = 'case8'  
engine = create\_engine(f"mysql+pymysql://{username}:{password}@{host}:{port}/{database}")  
  
dim\_patients.to\_sql('dim\_patients', engine, if\_exists='replace', index=False)  
dim\_doctors.to\_sql('dim\_doctors', engine, if\_exists='replace', index=False)  
fact\_admissions.to\_sql('fact\_admissions', engine, if\_exists='replace', index=False)  
fact\_vitals.to\_sql('fact\_vitals', engine, if\_exists='replace', index=False)  
fact\_treatments.to\_sql('fact\_treatments', engine, if\_exists='replace', index=False)  
  
print("Data successfully loaded to MySQL with improved star schema.")

This finalizes the **ETL pipeline**. At this point, I can connect **Power BI** (or any other BI tool) to the case8 MySQL database and begin constructing dashboards and analytics.

## Schema



## 6. PowerBI Report

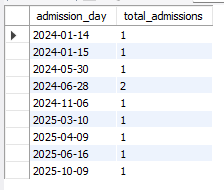
All Data is self generated so creating report is unnessesary

## 7. Example SQL Queries for Analytics

Once the data is in MySQL, I can run queries such as:

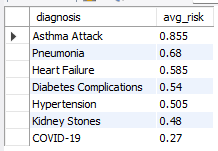
### Admissions by Day

SELECT   
 DATE(admission\_date) AS admission\_day,  
 COUNT(\*) AS total\_admissions  
FROM fact\_admissions  
GROUP BY DATE(admission\_date)  
ORDER BY admission\_day;



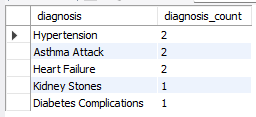
### Average Risk Score by Diagnosis

SELECT   
 diagnosis,  
 AVG(risk\_score) AS avg\_risk  
FROM fact\_admissions  
WHERE risk\_score IS NOT NULL  
GROUP BY diagnosis  
ORDER BY avg\_risk DESC;



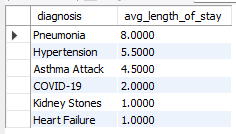
### Most Common Diagnoses

SELECT  
 diagnosis,  
 COUNT(\*) AS diagnosis\_count  
FROM fact\_admissions  
GROUP BY diagnosis  
ORDER BY diagnosis\_count DESC  
LIMIT 5;



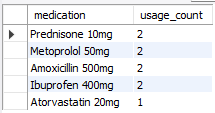
### Average Length of Stay by Diagnosis

SELECT  
 diagnosis,  
 AVG(DATEDIFF(discharge\_date, admission\_date)) AS avg\_length\_of\_stay  
FROM fact\_admissions  
WHERE discharge\_date IS NOT NULL  
GROUP BY diagnosis  
ORDER BY avg\_length\_of\_stay DESC;



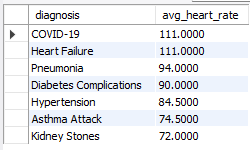
### Most Frequently Prescribed Medications

SELECT  
 medication,  
 COUNT(\*) AS usage\_count  
FROM fact\_treatments  
GROUP BY medication  
ORDER BY usage\_count DESC  
LIMIT 5;



### Average Heart Rate by Diagnosis

SELECT  
 fa.diagnosis,  
 AVG(fv.heart\_rate) AS avg\_heart\_rate  
FROM fact\_admissions AS fa  
JOIN fact\_vitals AS fv  
 ON fa.admission\_id = fv.admission\_id  
GROUP BY fa.diagnosis  
ORDER BY avg\_heart\_rate DESC;



# Conclusion

No conclusion because data is all self generated.