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"
])
```

```python
# 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:

```python
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:

```python
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:

```python
# 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.

```python
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:

```python
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)
```

2. **FactVitals**: Stores vitals for each admission (heart_rate, blood_pressure, etc.), keyed by admission_id.

3. **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:

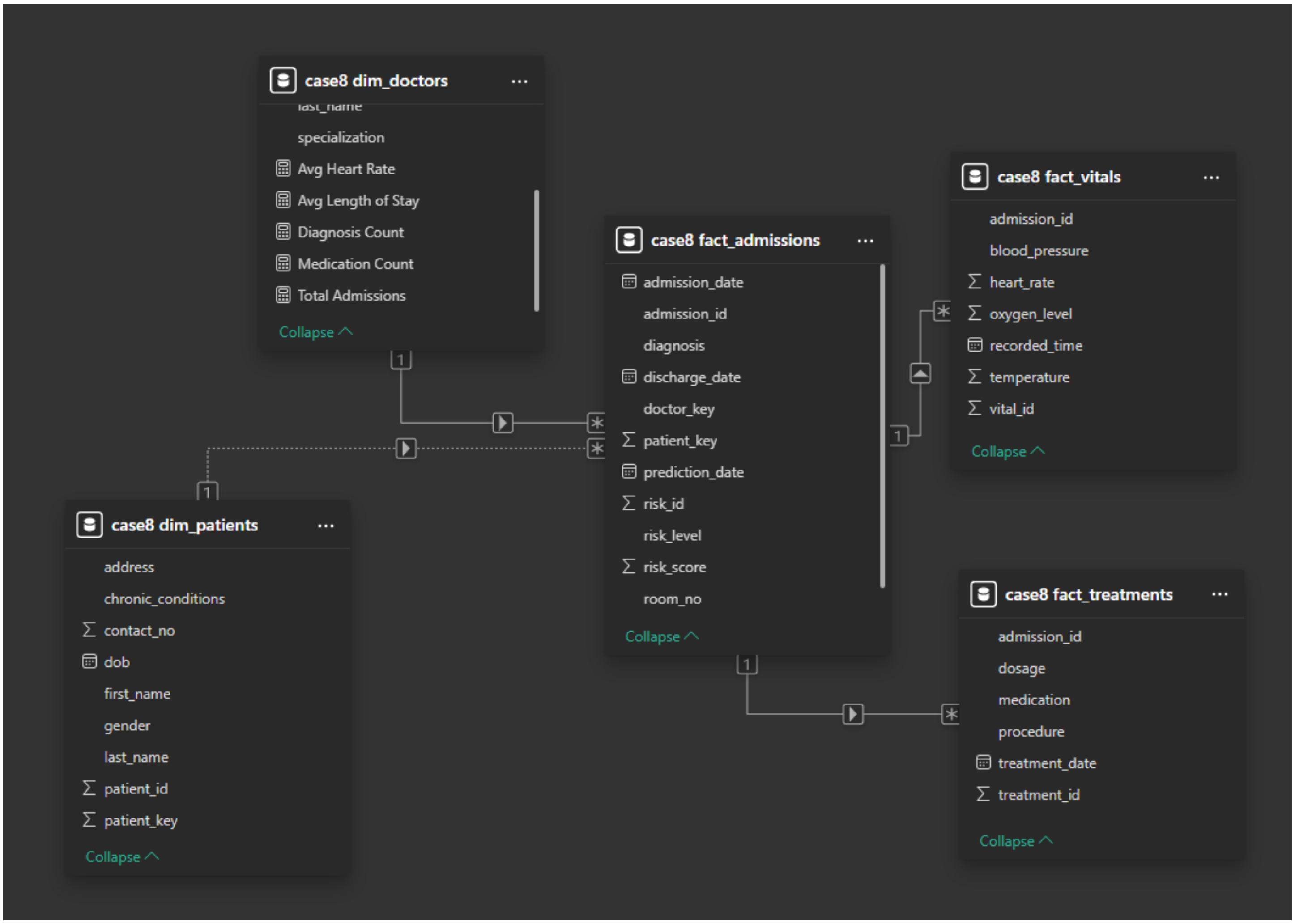
```
username = 'root'
password = '12345'
host = 'localhost'
port = '3306'
database = 'case8'
engine = create_engine(f"mysql+pymysql://{username}:{password}@{host}:{port}/{database}")

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;
```

	admission_day	total_admissions
▶	2024-01-14	1
	2024-01-15	1
	2024-05-30	1
	2024-06-28	2
	2024-11-06	1
	2025-03-10	1
	2025-04-09	1
	2025-06-16	1
	2025-10-09	1

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;
```

	diagnosis	avg_risk
▶	Asthma Attack	0.855
	Pneumonia	0.68
	Heart Failure	0.585
	Diabetes Complications	0.54
	Hypertension	0.505
	Kidney Stones	0.48
	COVID-19	0.27

Most Common Diagnoses

```
SELECT
    diagnosis,
    COUNT(*) AS diagnosis_count
FROM fact_admissions
GROUP BY diagnosis
ORDER BY diagnosis_count DESC
LIMIT 5;
```

	diagnosis	diagnosis_count
▶	Hypertension	2
	Asthma Attack	2
	Heart Failure	2
	Kidney Stones	1
	Diabetes Complications	1

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;
```

	diagnosis	avg_length_of_stay
▶	Pneumonia	8.0000
	Hypertension	5.5000
	Asthma Attack	4.5000
	COVID-19	2.0000
	Kidney Stones	1.0000
	Heart Failure	1.0000

Most Frequently Prescribed Medications

```
SELECT
    medication,
    COUNT(*) AS usage_count
FROM fact_treatments
GROUP BY medication
ORDER BY usage_count DESC
LIMIT 5;
```

	medication	usage_count
▶	Prednisone 10mg	2
	Metoprolol 50mg	2
	Amoxicillin 500mg	2
	Ibuprofen 400mg	2
	Atorvastatin 20mg	1

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;
```

	diagnosis	avg_heart_rate
▶	COVID-19	111.0000
	Heart Failure	111.0000
	Pneumonia	94.0000
	Diabetes Complications	90.0000
	Hypertension	84.5000
	Asthma Attack	74.5000
	Kidney Stones	72.0000

Conclusion

No conclusion because data is all self generated.