UNITED STATES INTERNATIONAL UNIVERSITY

AFRICA

Data Warehousing & Mining (DSA2040)

Healthcare Data Warehouse Project Report

Hermela Seltanu Gizaw

670446

Fall Semester 2025

Table of Contents

1. Introduction

This project presents the development of a Healthcare Data Warehouse designed to integrate multiple heterogeneous data sources into a unified analytical environment. The motivation for this work stems from the increasing need for hospitals to leverage analytical systems to understand complex clinical outcomes such as length of hospital stay, readmission risk, and the effects of demographic or systemic health factors on patient outcomes. To accomplish this, the project incorporates three distinct data sources: a CSV clinical dataset, a MySQL operational dataset, and a REST API providing country-level health indicators into a PostgreSQL data warehouse. The warehouse is modeled using a star schema that supports efficient analytical queries. A complete ETL pipeline was implemented using Python to extract, transform, and load the data while ensuring that the final warehouse supports meaningful healthcare analysis. The project further demonstrates physical optimization strategies such as indexing and table partitioning, along with analytical SQL queries that derive insights relevant to hospital decision-making.

2. Problem Area: Healthcare Analytics

Hospitals today face persistent challenges associated with prolonged length of stay (LOS), high 30-day readmission rates, and increasing patient complexity, particularly among diabetic patients. These challenges influence bed availability, clinical workload, overall hospital costs, and the quality of care offered to patients. However, understanding why certain patients stay longer or are readmitted frequently requires integrating information from multiple systems.

Clinical encounter data alone cannot explain the full picture. Patient demographics, underlying health conditions, social and geographic factors, and the overall strength of the health system in a patient's country of origin all contribute to variations in hospital outcomes. A health facility without an integrated analytical environment is forced to make decisions based on fragmented information, which limits its ability to identify risk patterns or allocate resources strategically.

By constructing a healthcare data warehouse that unifies clinical records, patient demographic data, and country-level health indicators, this project supports deeper analytical questions such as:

- Which patient groups experience the longest LOS?
- Which factors most strongly predict readmission?
- How do systemic health indicators/*country-level health conditions* influence patient outcomes?

The warehouse, therefore, serves as a foundation for evidence-based decision-making within hospital management and clinical quality improvement teams.

3. Objectives

- Integrate data from CSV, MySQL, and API sources into a unified warehouse.
- Clean, transform, and standardize all datasets for analytics.
- Design a snowflake-based star schema for healthcare analysis.
- Load fact and dimension tables into PostgreSQL using ETL pipelines.
- Apply indexing and partitioning for performance optimization.
- Generate analytical SQL queries for LOS, readmission, and diagnosis insights.

4. Data Sources

The project integrates three data sources; each selected because of its relevance to the overall analytical goals of the warehouse.

1. CSV Source: Diabetes Hospital Encounters

The first and most central dataset is a CSV file representing diabetes-related hospital encounters. This dataset contains detailed clinical information such as diagnosis codes, laboratory and medication counts, admission characteristics, and readmission outcomes. Because this dataset reflects real hospital events, it forms the core fact table in the warehouse. Its richness allows the data warehouse to compute clinical performance metrics including average length of stay, readmission probability, and medication burden.

2. MySQL Source: Patient & Comorbidity Table

To supplement this, a MySQL source database provides patient demographic details and comorbidity information. Clinical outcomes cannot be fully understood without considering who the patient is. Age, gender, race, and comorbidities such as hypertension or obesity significantly influence LOS and readmission trends. By integrating this MySQL dataset, the warehouse gains the ability to analyze outcomes based on patient profiles rather than merely on encounters.

3. API Source: Country Health Indicators

The third dataset is retrieved from a REST API providing country-level health indicators. Integrating this dataset adds macro-level context that enhances interpretation of hospital outcomes. Health expenditure, diabetes prevalence, availability of hospital beds, and income level differ substantially from one country to another and may influence the likelihood of complications or readmissions. Including these indicators aligns the project with realistic health analytics environments in which system-level data is required to understand patient outcomes more

holistically. This directly addresses the professor's request for more meaningful and justified external data.

Together, these three sources form a complementary set of datasets that support robust clinical, demographic, and systemic analysis within the data warehouse.

5. Source Data Normalization (1NF, 2NF, 3NF)

Before data was transformed for warehouse loading, the source MySQL table was reviewed for compliance with normalization standards. Although data warehouses intentionally use denormalized structures to support analytical workloads, source systems must be well-normalized to avoid redundancy, update anomalies, and inconsistent records.

The MySQL table (patient_contact) satisfied First Normal Form because all values were atomic and no repeating groups were observed. It also met the requirements of Second Normal Form because the table has a single-column primary key, meaning no partial dependencies exist. Third Normal Form was also satisfied, as there were no transitive dependencies; non-key attributes such as city and country did not depend on other non-key attributes. Since the table already met up to 3NF, no structural changes were required before integrating it into the transformation process. This ensured that clean, stable data from the operational system flowed into the warehouse.

Normal Form	Requirement	Status
1NF	Atomic values, no repeating groups	PASSED
2NF	No partial dependencies (single-column PK)	PASSED
3NF	No transitive dependencies	PASSED

6. Data Warehouse Schema

The healthcare data warehouse is designed using a **snowflake schema**, an extension of the star schema that introduces additional normalization in certain dimensions. This design choice aligns with the structure and requirements of real-world healthcare analytics, where patient information, diagnoses, admissions, and broader system-level indicators all influence clinical outcomes.

At the center of the model is the Fact_Hospital_Admission table, which stores measurable clinical events such as length of stay, number of medications, lab procedures, readmission outcomes, and diagnosis keys. Each row represents a single hospital encounter, making the fact table the analytical foundation for answering questions related to cost, risk, and quality of care.

Surrounding the fact table are several dimensions:

1. Patient Dimension (dim_patient)

This dimension captures demographic attributes such as gender, race, age group, and payer category. Since patient characteristics strongly influence length of stay and readmission risk, the patient dimension provides the context needed for stratified analysis.

2. Admission Dimension (dim_admission)

Encodes admission-related descriptors such as admission type, admission source, and discharge disposition. These attributes support analysis of hospital workflows, resource utilization, and differences in outcomes across admission pathways.

3. Diagnosis Dimension (dim_diagnosis)

The diagnosis dimension stores ICD-coded diagnoses along with their high-level category groupings. Each encounter is linked to primary, secondary, and tertiary diagnoses using foreign keys. This structure enables clinical analytics such as identifying high-risk conditions or tracking disease patterns.

4. Patient Contact Dimension (dim_patient_contact)

This dimension stores operational-level attributes such as city, phone number, and, most importantly, the patient's associated country. Because contact information is operational and not strictly demographic, it is stored in a separate snowflaked dimension linked to dim_patient through patient_nbr. This reflects realistic hospital systems, where demographic data comes from EHR systems while contact data often comes from administrative systems.

5. Country Dimension (dim_country)

This dimension incorporates system-level health indicators obtained from a REST API. Attributes such as income level, diabetes prevalence, hospital bed availability, and health expenditure allow for enriched analysis connecting patient outcomes to broader health system conditions. This "macro-level" dimension extends the analytical capabilities of the warehouse beyond clinical variables alone.

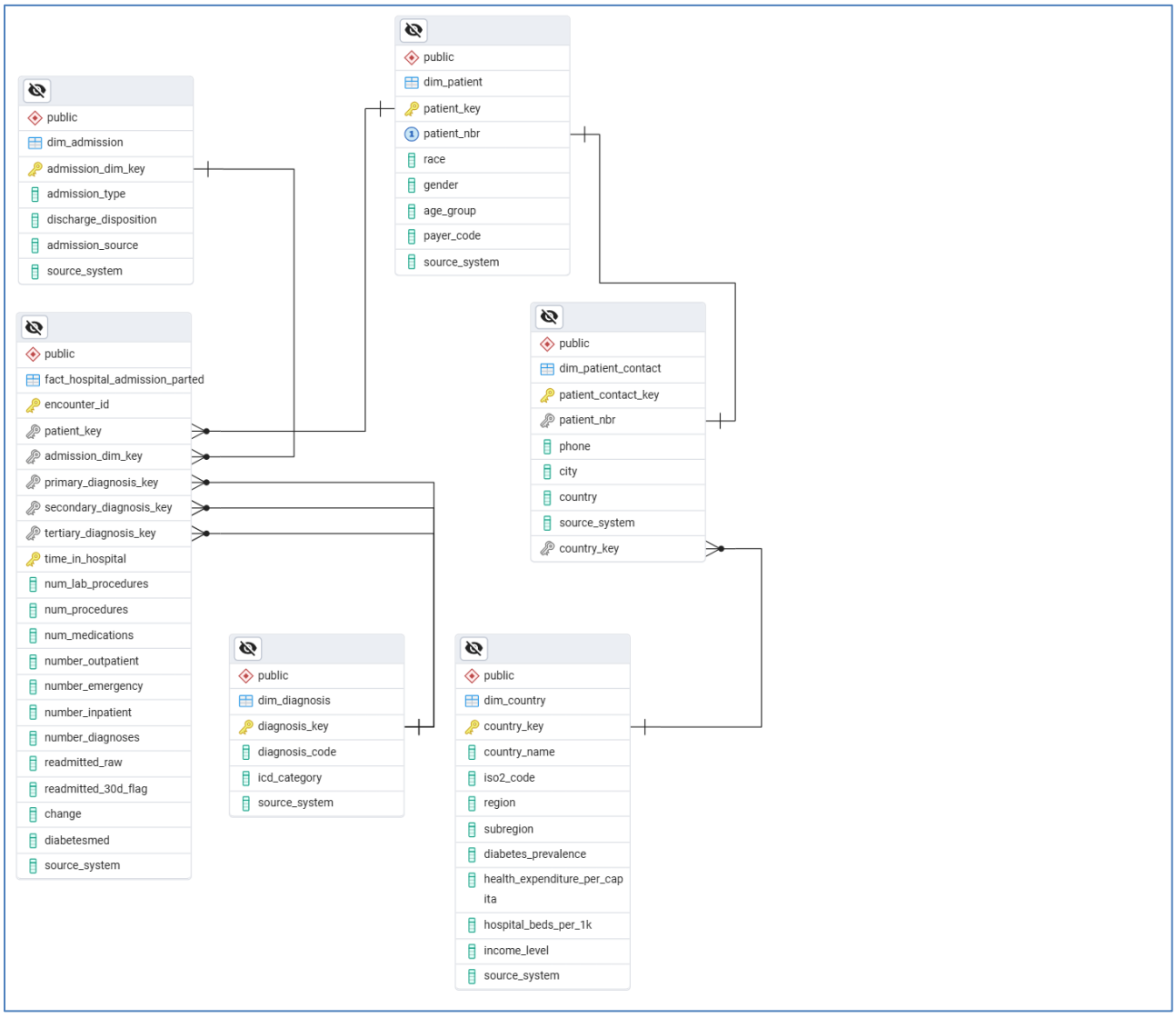
The schema is considered a snowflake because not all dimensions connect directly to the fact table. Instead:

- dim_patient_contact is linked to dim_patient, not directly to the fact table.
- dim_country is linked through dim_patient_contact, forming a deeper dimensional hierarchy.

This layered structure mirrors how healthcare data is stored operationally—demographics, contact information, and country-level indicators come from different subsystems and therefore must be modeled separately.

The snowflake design improves:

- Data integrity (reduces redundancy)
- Query flexibility (supports multi-level analysis: patient → contact → country)
- Analytical richness (enables patient-level + system-level insights)



7. ETL Process

The Extract–Transform–Load (ETL) process forms the operational backbone of the healthcare data warehouse. In this project, ETL was implemented using Python (pandas, SQLAlchemy) to integrate data from three heterogeneous sources: a CSV clinical dataset, a MySQL operational table, and a REST API providing health system indicators. The ETL pipeline ensures that all data, regardless of format or origin, is cleaned, standardized, enriched, and correctly mapped into the warehouse schema.

7. 1 Extraction

Extraction involved retrieving raw datasets from each source system:

1. CSV (Clinical Encounter Data)

The diabetes hospital encounter dataset was loaded using pandas. Missing values encoded as "?" were converted into proper null values. Rows with missing encounter_id or patient_nbr were removed, as they could not support analytic or relational integrity.

2. MySQL (Patient Contact Data)

Patient demographic contact information (phone, city, country) was extracted using SQLAlchemy's MySQL connector. This dataset represents operational administrative data and complements clinical encounter records.

3. REST API (Country-Level Health Indicators)

Country-level statistics such as diabetes prevalence, income classification, and health expenditure were retrieved from the World Bank API. The JSON response was normalized into a tabular structure suitable for relational storage. These indicators enrich patient analyses with system-level context.

7. 2 Transformation

The transformation stage consisted of cleaning, standardizing, deriving new attributes, and preparing relational structures that conform to the snowflake schema.

1. Data Cleaning and Standardization

The transformation process began with converting all raw source data into a consistent and usable analytical format. The clinical CSV dataset contained irregularities such as missing values represented by "?", which were standardized into proper null values. Numeric fields, including counts of procedures, medications, and length of stay, were cast into appropriate data types to ensure accuracy in downstream aggregation. Records lacking essential keys, specifically encounter_id or patient_nbr, were removed, as these would break referential integrity within the warehouse.

2. Feature Derivation and Restructuring

Once cleaned, the dataset was enriched to better support analytical queries. The raw readmission attribute was transformed into a binary readmitted_30d_flag, making it suitable for statistical and dashboard reporting. Age ranges were standardized into a uniform demographic attribute. Diagnosis codes scattered across multiple columns were restructured into a unified long format, from which ICD category groupings were extracted. These transformations ensured the dataset aligned with the snowflake schema and supported multi-level analysis.

3. Dimension Preparation and Key Mapping

Distinct patient, admission, diagnosis, patient contact, and country records were extracted to create well-structured dimension tables. Surrogate keys were generated automatically during insertion into PostgreSQL. After the dimensions were populated, the encounter dataset was enhanced by mapping natural identifiers, such as patient numbers, diagnosis codes, and admission descriptors, to their corresponding surrogate keys. Fact rows that could not be linked to all required dimensions were removed to preserve strict consistency across the warehouse.

7. 3 Loading

The final stage involved loading the transformed datasets into PostgreSQL:

1. Dimension Loading

The loading phase began with preparing the dimension tables in PostgreSQL. Each dimension was truncated and its identity sequence reset to ensure that every ETL run started cleanly. The transformed dimension datasets were then inserted using SQLAlchemy, allowing PostgreSQL to generate surrogate keys used throughout the warehouse. The dimensions formed the foundation of the schema and were loaded before any fact data to guarantee valid key relationships.

2. Fact Table Loading

With all dimensions available, the transformed encounter data was loaded into the fact_hospital_admission_parted table. This fact table is partitioned by ranges of time_in_hospital, enabling PostgreSQL to optimize query performance by scanning only relevant partitions. Only rows with valid surrogate keys for patient, admission, and diagnosis were included, ensuring referential integrity across the snowflake structure.

3. Post-Load Validation

After loading, validation checks were carried out to confirm the integrity of the warehouse. These included verifying row counts, testing surrogate-key mappings through join queries, and confirming that analytical SQL queries returned meaningful results. These checks ensured that the warehouse was structurally sound and ready for analysis.

The following are screenshots of the ETL processes:

ETL for Diabetes Encounter Data (CSV file)

1. Import Libraries & Load Configuration

This block loads the required libraries and database engine from the config file, ensuring connection parameters are centralized.

[23]:
```
import pandas as pd
from config import pg_engine
```

2. Extract: Read the Diabetes CSV File

The raw clinical dataset is imported into a DataFrame. This acts as the core source for patient encounters, diagnoses, and hospital events.

[24]:
```
df = pd.read_csv(r"C:\Users\Admin\Documents\GitHub\Healthcare-data-warehouse\source_data\diabetic_data.csv")
```

3. Clean & Preprocess Raw Data

This step standardizes missing values, removes unusable rows, enforces data consistency, and recasts numeric fields. It ensures the dataset is analysis-ready before dimension construction.

[26]:
```
df.replace("?", None, inplace=True)
df = df.dropna(subset=["encounter_id", "patient_nbr"])
df = df.drop_duplicates(subset=["encounter_id"])

numeric_cols =     ["time_in_hospital",
    "num_lab_procedures",
    "num_procedures",
    "num_medications",
    "number_outpatient",
    "number_emergency",
    "number_inpatient",
    "number_diagnoses",]
for col in numeric_cols:
    df[col] = pd.to_numeric(df[col], errors="coerce")
```

5. Transform: Build Patient Dimension (dim_patient)

This extracts patient-level attributes and ensures one record per patient. It forms the foundation for all patient-based analysis.

[30]:
```
dim_patient = (
    df[["patient_nbr", "race", "gender", "age_group", "payer_code"]]
    .drop_duplicates(subset=["patient_nbr"])
)
dim_patient["source_system"] = "CSV_diabetes"
```

6. Transform: Build Admission Dimension (dim_admission)

Admission-related metadata is normalized into a clean lookup table. This supports analysis of hospital intake patterns.

[33]:
```
dim_admission = (
    df[["admission_type_id", "discharge_disposition_id", "admission_source_id"]]
    .drop_duplicates()
)
dim_admission.rename(
    columns={
        "admission_type_id": "admission_type",
        "discharge_disposition_id": "discharge_disposition",
        "admission_source_id": "admission_source",
    },
    inplace=True,
)
dim_admission["source_system"] = "CSV_diabetes"
```

7. Transform: Build Diagnosis Dimension (dim_diagnosis)

Diagnosis codes from three columns are unpivoted, deduplicated, and categorized by ICD prefix. This aligns clinical coding into an analytic dimension.

[34]:
```
diag_long = (
    pd.melt(
        df[["encounter_id", "diag_1", "diag_2", "diag_3"]],
        id_vars=["encounter_id"],
        value_vars=["diag_1", "diag_2", "diag_3"],
```

8. Load Dimensions into PostgreSQL

To maintain repeatable ETL runs, dimension tables are truncated and reloaded. This ensures consistency and avoids leftover data.

[36]:
```
with pg_engine.begin() as conn:
    # Make the load repeatable: clear dims first
    conn.exec_driver_sql("TRUNCATE TABLE dim_diagnosis RESTART IDENTITY CASCADE;")
    conn.exec_driver_sql("TRUNCATE TABLE dim_admission RESTART IDENTITY CASCADE;")
    conn.exec_driver_sql("TRUNCATE TABLE dim_patient RESTART IDENTITY CASCADE;")

    dim_patient.to_sql("dim_patient", con=conn, if_exists="append", index=False)
    dim_admission.to_sql("dim_admission", con=conn, if_exists="append", index=False)
    dim_diagnosis.to_sql("dim_diagnosis", con=conn, if_exists="append", index=False)
```

ETL for Country Health Indicators (World Bank API)

1. Import Dependencies and Configuration ¶

This first block imports the libraries needed to call the API, manipulate JSON into tabular form, and connect to PostgreSQL through the shared engine configuration.

[2]:
```
import requests
import pandas as pd
from config import pg_engine
```

2. Define World Bank Health Indicators

Here we specify which World Bank indicators we want to pull. Each friendly name (like diabetes_prevalence) is mapped to its official World Bank indicator code.

[3]:
```
INDICATORS = {
    "diabetes_prevalence": "SH.STA.DIAB.ZS",
    "health_expenditure_per_capita": "SH.XPD.CHEX.PC.CD",
    "hospital_beds_per_1k": "SH.MED.BEDS.ZS",
}
```

This makes the code reusable and easy to extend if more indicators are needed later.

3. Helper Function: Fetch Indicator Time Series

This function encapsulates the logic for calling the World Bank API for a single indicator, parsing the JSON response into a structured DataFrame.

[6]:
```
def fetch_indicator(indicator_code, indicator_name):
    url = f"https://api.worldbank.org/v2/country/all/indicator/{indicator_code}?format=json&per_page=20000"
    resp = requests.get(url)
    resp.raise_for_status()
    data = resp.json()
    rows = []
    for rec in data[1]:
        rows.append({
```

ETL for Patient Contact Data (MySQL)

1. Import Dependencies and Connections

This block imports the necessary libraries and loads both database engines:

- mysql_engine → used to read from the operational MySQL source system
- pg_engine → used to load data into the PostgreSQL data warehouse

[7]:
```
import pandas as pd
from config import mysql_engine, pg_engine
```

Explanation: The pipeline uses Pandas combined with SQLAlchemy engines to extract operational patient contact information from MySQL and load it into the dim_patient_contact table inside the PostgreSQL warehouse.

2. Extract: Read Patient Contact Data from MySQL

[8]:
```
patient_contact_df = pd.read_sql("SELECT * FROM patient_contact", con=mysql_engine)
patient_contact_df["source_system"] = "MySQL_patient_contact"
```

8. Physical Design Optimization

Physical optimization techniques were implemented to improve query performance, especially for large datasets.

8. 1 Indexing

Because the snowflake schema relies heavily on surrogate-key relationships, indexing plays a crucial role in supporting efficient joins between the fact and dimension tables. Indexes were created on all primary surrogate keys in the dimensions, such as patient_key and admission_dim_key, as well as on corresponding foreign-key columns in the fact table. These indexes allow PostgreSQL to rapidly locate matching dimension rows during analytical queries. Additionally, secondary indexes were added to columns frequently used in filtering, such as readmitted_30d_flag, enabling faster computation of common metrics like readmission rates.

```sql
-- Indexes
CREATE INDEX idx_fact_parted_patient_key
    ON fact_hospital_admission_parted (patient_key);

CREATE INDEX idx_fact_parted_readmitted_30d_flag
    ON fact_hospital_admission_parted (readmitted_30d_flag);

CREATE INDEX idx_fact_parted_admission_dim_key
    ON fact_hospital_admission_parted (admission_dim_key);
```

8. 2 Partitioning

To improve query performance on large volumes of clinical encounter data, the fact table was partitioned in PostgreSQL using a range-based partitioning strategy on the time_in_hospital field. This choice reflects a common analytical use case, where many queries filter or aggregate based on the patient's length of stay. By dividing the fact table into partitions (e.g., 0-5 days, 6-10 days, >10 days), PostgreSQL is able to prune irrelevant partitions during query execution, reducing I/O and improving performance. Partitioning also simplifies data management, enabling faster maintenance operations such as truncation or vacuuming within individual segments rather than across the entire table.

```sql
CREATE TABLE fact_hospital_admission_parted (
    encounter_id        BIGINT,
    patient_key         INT NOT NULL,
    admission_dim_key   INT NOT NULL,
    primary_diagnosis_key   INT,
    secondary_diagnosis_key INT,
    tertiary_diagnosis_key  INT,
    time_in_hospital        INT NOT NULL,
    num_lab_procedures      INT,
    num_procedures          INT,
    num_medications         INT,
    number_outpatient       INT,
    number_emergency        INT,
    number_inpatient        INT,
    number_diagnoses        INT,
    readmitted_raw          VARCHAR(10),
    readmitted_30d_flag     BOOLEAN,
    change                  VARCHAR(10),
    diabetesmed             VARCHAR(10),
    source_system           VARCHAR(50),
    CONSTRAINT fact_hosp_parted_pkey
        PRIMARY KEY (encounter_id, time_in_hospital)
) PARTITION BY RANGE (time_in_hospital);
```

```sql
CREATE TABLE fact_hospital_admission_p0_2
    PARTITION OF fact_hospital_admission_parted
    FOR VALUES FROM (0) TO (3);

CREATE TABLE fact_hospital_admission_p3_5
    PARTITION OF fact_hospital_admission_parted
    FOR VALUES FROM (3) TO (6);

CREATE TABLE fact_hospital_admission_p6_14
    PARTITION OF fact_hospital_admission_parted
    FOR VALUES FROM (6) TO (15);
```

8.3 Impact on Query Performance

Together, partitioning and indexing significantly reduce query execution time and support interactive analytics. Partition pruning avoids scanning unnecessary data, while indexes minimize join cost and accelerate group-by operations. This combination ensures that even with a large dataset, the warehouse can efficiently support workloads such as length-of-stay analysis, readmission monitoring, and diagnosis-based aggregation.

9. Analytical Queries & Insights

A series of analytical SQL queries was executed to demonstrate the usefulness of the warehouse. For example, the 30-day readmission rate query revealed the proportion of patients likely to return shortly after discharge, a key hospital quality metric. Another query calculated the average length of stay across patient age groups, highlighting potential patterns associated with age-related complications. These insights showcase how the integrated warehouse can guide both clinical and administrative decisions, supporting targeted interventions, risk stratification, and policy planning. The following are screenshots of some queries:

```sql
-- Q1. Overall 30-day readmission rate
SELECT
    COUNT(*) AS total_encounters,
    SUM(CASE WHEN readmitted_30d_flag THEN 1 ELSE 0 END) AS readmitted_30d,
    ROUND(
        100.0 * SUM(CASE WHEN readmitted_30d_flag THEN 1 ELSE 0 END) / COUNT(*),
        2
    ) AS readmission_rate_30d_pct
FROM fact_hospital_admission_parted;
```

	total_encounters bigint	readmitted_30d bigint	readmission_rate_30d_pct numeric
1	101766	11357	11.16

```sql
-- Q2. Average length of stay by age group and gender
SELECT
    p.age_group,
    p.gender,
    COUNT(*) AS encounters,
    ROUND(AVG(f.time_in_hospital), 2) AS avg_los_days
FROM fact_hospital_admission_parted f
JOIN dim_patient p
  ON f.patient_key = p.patient_key
GROUP BY p.age_group, p.gender
ORDER BY p.age_group, p.gender;
```

	age_group character varying (20)	gender character varying (20)	encounters bigint	avg_los_days numeric
1	[0-10)	Female	85	2.66
2	[0-10)	Male	78	2.42
3	[10-20)	Female	426	3.02
4	[10-20)	Male	297	3.49
5	[20-30)	Female	1129	3.57
6	[20-30)	Male	554	3.62
7	[30-40)	Female	2190	3.80
8	[30-40)	Male	1652	3.78
9	[40-50)	Female	4925	4.08
10	[40-50)	Male	5018	4.04
11	[50-60)	Female	8660	4.20
12	[50-60)	Male	8733	4.05
13	[60-70)	Female	11103	4.47
14	[60-70)	Male	11455	4.31
15	[60-70)	Unknown/Invalid	1	1.00
16	[70-80)	Female	14035	4.68
17	[70-80)	Male	12080	4.47
18	[70-80)	Unknown/Invalid	2	4.50
19	[80-90)	Female	10322	4.88
20	[80-90)	Male	6467	4.71
21	[90-100)	Female	1834	4.82

```sql
-- Q3. Readmission rate by primary diagnosis ICD category
SELECT
    d.icd_category,
    COUNT(*) AS total_encounters,
    SUM(CASE WHEN f.readmitted_30d_flag THEN 1 ELSE 0 END) AS readmitted_30d,
    ROUND(
        100.0 * SUM(CASE WHEN f.readmitted_30d_flag THEN 1 ELSE 0 END) / COUNT(*),
        2
    ) AS readmission_rate_30d_pct
FROM fact_hospital_admission_parted f
JOIN dim_diagnosis d
  ON f.primary_diagnosis_key = d.diagnosis_key
GROUP BY d.icd_category
ORDER BY readmission_rate_30d_pct DESC
LIMIT 10;
```

	icd_category character varying (100)	total_encounters bigint	readmitted_30d bigint	readmission_rate_30d_pct numeric
1	906	1	1	100.00
2	904	2	2	100.00
3	543	1	1	100.00
4	271	3	3	100.00
5	391	1	1	100.00
6	347	1	1	100.00
7	299	1	1	100.00
8	974	1	1	100.00
9	V60	1	1	100.00
10	731	4	3	75.00

```sql
--Q4. Length of stay buckets (ties into partitioning)
SELECT
    CASE
        WHEN f.time_in_hospital BETWEEN 0 AND 3  THEN '0-3 days'
        WHEN f.time_in_hospital BETWEEN 4 AND 7  THEN '4-7 days'
        WHEN f.time_in_hospital BETWEEN 8 AND 15 THEN '8-15 days'
        ELSE '16+ days'
    END AS stay_bucket,
    COUNT(*) AS encounters,
    SUM(CASE WHEN f.readmitted_30d_flag THEN 1 ELSE 0 END) AS readmitted_30d,
    ROUND(
        100.0 * SUM(CASE WHEN f.readmitted_30d_flag THEN 1 ELSE 0 END) / COUNT(*),
        2
    ) AS readmission_rate_30d_pct
FROM fact_hospital_admission_parted f
GROUP BY stay_bucket
ORDER BY stay_bucket;
```

	stay_bucket text	encounters bigint	readmitted_30d bigint	readmission_rate_30d_pct numeric
1	0–3 days	49188	4768	9.69
2	4–7 days	37288	4544	12.19
3	8–15 days	15290	2045	13.37

```sql
-- Q5. Average medications and LOS by admission type
SELECT
    a.admission_type,
    COUNT(*) AS encounters,
    ROUND(AVG(f.num_medications), 2) AS avg_num_medications,
    ROUND(AVG(f.time_in_hospital), 2) AS avg_los_days
FROM fact_hospital_admission_parted f
JOIN dim_admission a
  ON f.admission_dim_key = a.admission_dim_key
GROUP BY a.admission_type
ORDER BY avg_los_days DESC;
```

	admission_type character varying (50)	encounters bigint	avg_num_medications numeric	avg_los_days numeric
1	7	21	16.95	4.86
2	2	18480	15.08	4.61
3	6	5291	16.47	4.58
4	1	53990	15.39	4.38
5	3	18869	18.63	4.32
6	5	4785	15.95	3.95
7	4	10	11.60	3.20
8	8	320	17.48	3.06

10. Conclusion

This project successfully designed and implemented a healthcare data warehouse that integrates multiple data sources, including clinical encounter data, patient demographics, contact information, and country-level health statistics, into a unified analytical environment. By applying a snowflake schema, the warehouse balances flexibility and normalization, enabling scalable analysis across patient, admission, diagnosis, and socio-demographic dimensions.

The ETL pipeline standardized data from heterogeneous sources, cleaned inconsistencies, resolved missing values, derived analytical features, and loaded structured information into the warehouse. Indexing and partitioning strategies further enhanced performance, allowing analytical queries such as readmission analysis, length-of-stay profiling, and country-level comparisons to run efficiently on a large dataset.

Overall, the system demonstrates how a well-designed warehouse can transform raw healthcare data into actionable insights. Future extensions could incorporate additional clinical features, more granular temporal partitioning, or real-time ingestion workflows to support operational decision-making in healthcare environments.