# Project Phase 2 : Final Report

**Student Names and IDs:**

Idriss Benabdessadek 40248720,

Nigel Kyle Arintoc 40281248,

Sanjitt Kanagalingam  40313831,

Jeremy Fung 40252404

Quentin Bollore 40348932

**Course Section:** SOEN363 - Section S
**Professor Name:** Hamad Jafarpour
**Date:** November 26th, 2025

**Concordia University**
**Faculty of Engineering and Computer Science**

# Introduction

This project focuses on the design, implementation, and migration of a hospital database system through two major phases: a relational model (Phase I) and a NoSQL model (Phase II). The objective is to model and manage healthcare-related data representing patients, hospital admissions, ICU stays, clinical notes, and diagnoses. In Phase I, the database was built using the relational paradigm, with a complete ER diagram and SQL queries that demonstrated the ability to retrieve meaningful insights about patient care. This first phase established a structured and normalized environment suitable for integrity, consistency, and relational querying.

Phase II extends the same system into the NoSQL domain. This step evaluates how the same dataset can be represented using a schema-flexible model capable of handling large volumes of data in a more scalable and horizontally distributed environment. The goal is to migrate the data from the relational database to a NoSQL database, apply necessary design adjustments, and compare the two systems in terms of structure, performance, and query capabilities. Together, the two phases provide a complete view of data engineering approaches and demonstrate the impact of switching from a relational schema to a NoSQL document-oriented architecture.

# Project Description

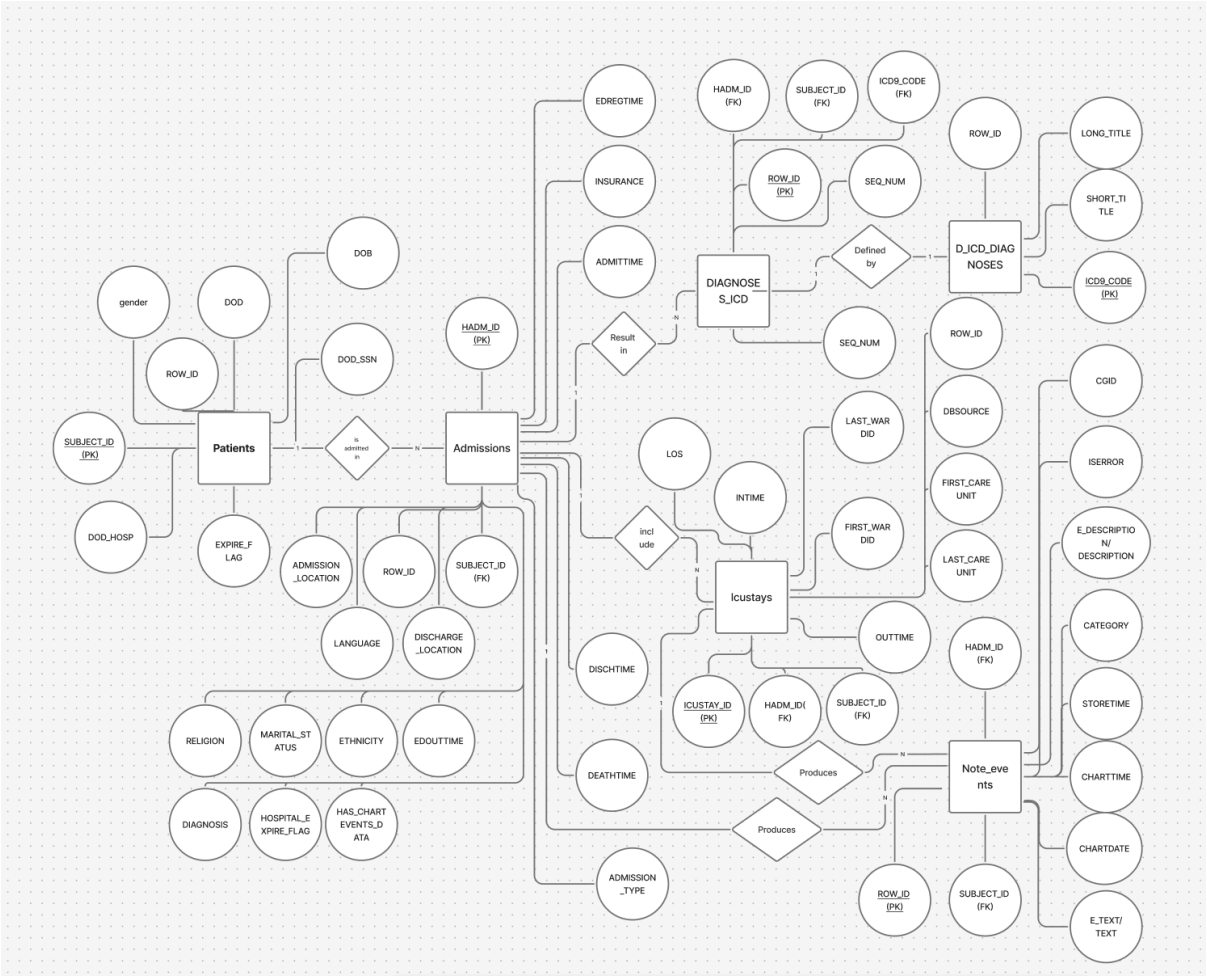
The project models the flow of medical information within a hospital environment. Each individual who interacts with the healthcare system is registered as a patient with a unique identifier that persists over their lifetime. A patient may have multiple hospital admissions, and each admission can include ICU stays, clinical notes, and diagnoses assigned according to the ICD-9 medical coding standard. These relationships form a complete representation of a patient's journey through the healthcare system.

In Phase I, this scenario was implemented as a relational database. An ER diagram was created to capture all entities, attributes, and relationships, including PATIENTS, ADMISSIONS, ICUSTAYS, NOTEEVENTS, DIAGNOSES_ICD, and D_ICD_DIAGNOSES. The ER model was then translated into an RDBMS schema using tables, primary keys, foreign keys, and constraints. Data was imported into the relational database, and a set of SQL queries was executed to analyze patient demographics, hospital admissions, ICU activity, clinical documentation, and diagnosis patterns.

Phase II focuses on converting this relational design into a NoSQL implementation. A document-oriented model was chosen to group related information, such as admissions, notes, ICU stays, and diagnoses directly under each patient record. This conversion required modifying the original relational design by embedding strongly related entities, removing foreign key dependencies, and restructuring weak entities to fit a document-based structure. Data migration scripts were developed to extract data from the relational database and load it into the NoSQL database while preserving correctness and consistency. The final NoSQL system enables comparisons with the relational system regarding performance, data access patterns, and structural design.

# ER Diagram

The Entity–Relationship (ER) diagram models the core components of the hospital information system and captures how patient-related medical data is structured and connected. The diagram is centered around five main entities: **PATIENTS**, **ADMISSIONS**, **ICUSTAYS**, **NOTEEVENTS**, and **DIAGNOSES_ICD**, along with a supporting dictionary entity **D_ICD_DIAGNOSES**. Each entity contains a set of attributes and participates in relationships reflecting the real behavior of the healthcare environment. The following sections describe the entities and their relationships in detail.



# 1. Entities and Their Attributes

## 1.1 PATIENTS

This entity represents individuals receiving medical care.
Key attributes include:

- **SUBJECT_ID (PK)**: Unique identifier assigned to each patient.
- **GENDER**, **DOB**, **DOD**, **DOD_HOSP**, **DOD_SSN**: Demographic and mortality information.
- **EXPIRE_FLAG**: Indicates whether the patient is deceased.
- **ROW_ID**: Internal row identifier.

The patient is the root entity to which most other data is linked throughout the system.

## 1.2 ADMISSIONS

Represents each hospital visit. A patient can have several admissions over time.
Key attributes:

- **HADM_ID (PK)**: Unique identifier for a hospital admission.
- **SUBJECT_ID (FK)** → PATIENTS.
- **ADMITTIME**, **DISCHTIME**, **DEATHTIME**.
- **ADMISSION_TYPE**, **ADMISSION_LOCATION**, **DISCHARGE_LOCATION**.
- **INSURANCE**, **RELIGION**, **MARITAL_STATUS**, **ETHNICITY**.
- **EDREGTIME**, **EDOUTTIME**, **DIAGNOSIS**.
- **HOSPITAL_EXPIRE_FLAG**, **HAS_CHARTEVENTS_DATA**.

This entity captures contextual and demographic information for each hospital stay.

## 1.3 ICUSTAYS

Represents ICU periods occurring during or related to an admission.
Key attributes:

- **ICUSTAY_ID (PK)**.
- **SUBJECT_ID (FK)** → PATIENTS.
- **HADM_ID (FK)** → ADMISSIONS.
- **INTIME**, **OUTTIME**, **LOS** (length of stay).
- **FIRST_CAREUNIT**, **LAST_CAREUNIT**, **FIRST_WARDID**, **LAST_WARDID**.

Every ICU stay is tied to both a patient and a specific admission.

## 1.4 NOTEEVENTS

Represents clinical notes produced by doctors, nurses, or technicians.
Key attributes:

- **ROW_ID (PK)**.
- **SUBJECT_ID (FK)** → PATIENTS.
- **HADM_ID (FK)** → ADMISSIONS.
- **CHARTDATE**, **CHARTTIME**, **STORETIME**.
- **CATEGORY**.
- **E_DESCRIPTION**.
- **CGID** (author identifier).
- **ISERROR**, **E_TEXT**.

Notes provide narrative documentation that complements structured hospital data.

## 1.5 DIAGNOSES_ICD

Represents diagnoses assigned at discharge.
Key attributes:

- **ROW_ID (PK)**.
- **SUBJECT_ID (FK)** → PATIENTS.
- **HADM_ID (FK)** → ADMISSIONS.
- **ICD9_CODE (FK)** → D_ICD_DIAGNOSES.
- **SEQ_NUM**: Order of diagnosis.

This entity links patients and admissions with standardized ICD codes.

## 1.6 D_ICD_DIAGNOSES

Dictionary table containing medical definitions.
Key attributes:

- **ICD9_CODE (PK)**.
- **SHORT_TITLE**, **LONG_TITLE**.
- **ROW_ID**.

This entity ensures consistency across diagnosis codes.

# 2. Relationships and Cardinalities

## 2.1 PATIENTS → ADMISSIONS

**1-to-many**
A single patient may have *zero, one, or multiple* hospital admissions.
Cardinality: **PATIENTS (1) — (N) ADMISSIONS**
This is enforced via `SUBJECT_ID` in ADMISSIONS referencing PATIENTS.

## 2.2 ADMISSIONS → ICUSTAYS

**1-to-many**
A hospital admission may involve *zero or more* ICU stays.
Cardinality: **ADMISSIONS (1) — (N) ICUSTAYS**
Each ICU stay must belong to exactly one admission.

## 2.3 PATIENTS → ICUSTAYS

**1-to-many**
A patient can have multiple ICU stays over multiple admissions.
Cardinality: **PATIENTS (1) — (N) ICUSTAYS**

This dual dependency (PATIENT + ADMISSION) ensures ICU stays are properly linked to the correct episode of care.

## 2.4 ADMISSIONS → NOTEEVENTS

**1-to-many**
An admission generates multiple clinical notes.
Cardinality: **ADMISSIONS (1) — (N) NOTEEVENTS**

Each note is tied to a specific admission via `HADM_ID`.

## 2.5 PATIENTS → NOTEEVENTS

**1-to-many**
A patient may have multiple notes across different visits.
Cardinality: **PATIENTS (1) — (N) NOTEEVENTS**

## 2.6 ADMISSIONS → DIAGNOSES_ICD

**1-to-many**
A single hospital admission can have several diagnoses.
Cardinality: **ADMISSIONS (1) — (N) DIAGNOSES_ICD**

## 2.7 DIAGNOSES_ICD → D_ICD_DIAGNOSES

**1-to-1**
Each diagnosis refers to a code defined in the ICD dictionary.
Cardinality: **DIAGNOSES_ICD (1) — (1) D_ICD_DIAGNOSES**

This ensures that diagnoses follow standardized medical terminology.

# 3. Summary of the ER Model

The ER diagram models a realistic hospital environment by emphasizing:

- **Strong hierarchical relationships**, with PATIENT → ADMISSION → ICU/NOTES/DIAGNOSES.
- **Clear cardinalities** that represent real-world medical processes.
- **Normalized design**, where repeated medical terminology is centralized in the ICD dictionary.
- **Comprehensive tracking**, from patient demographics to admission context, ICU care, documentation, and diagnostic outcomes.

The model provides a complete and structured view of patient trajectories, supporting both analytical SQL queries and the subsequent conversion to a more flexible NoSQL architecture.

# RDBMS Database: ER-to-Relational Conversion

The relational database was implemented using SQL based on the structure defined in the ER diagram. The goal of this phase was to translate the conceptual model into a normalized relational schema that enforces data integrity, supports efficient querying, and preserves all relationships between entities. The database is composed of six tables corresponding directly to the major entities in the ER model: **PATIENTS**, **ADMISSIONS**, **NOTEEVENTS**, **DIAGNOSES_ICD**, **D_ICD_DIAGNOSES**, and **ICUSTAYS**. Each table includes primary keys, foreign keys, data types, and constraints that reflect the cardinalities and logical structure found in the ER diagram. The SQL script to create the database can be found under Appendix A.2.

# 1. Mapping Entities to Tables

## 1.1 PATIENTS Table

The **PATIENTS** entity is the central reference point of the system, so it is represented by its own table with a primary key:

- **SUBJECT_ID (PK)** uniquely identifies each patient.
- Demographic attributes (**GENDER**, **DOB**, **DOD**, **DOD_HOSP**, **DOD_SSN**) and status fields (**EXPIRE_FLAG**) are stored as columns.
- Since this table is referenced by multiple other tables, its primary key forms the basis for several foreign-key constraints.

This table corresponds directly to the PATIENTS entity in the ER diagram with no structural change.

## 1.2 ADMISSIONS Table

The **ADMISSIONS** table represents hospital visits. It maps directly from the ER diagram and includes:

- **HADM_ID (PK)** as the unique admission identifier.
- **SUBJECT_ID (FK)** referencing PATIENTS(SUBJECT_ID), implementing the **1-to-many** relationship between patients and admissions.
- Temporal attributes (**ADMITTIME**, **DISCHTIME**, **DEATHTIME**) and descriptive attributes (**ADMISSION_TYPE**, **INSURANCE**, **LANG**, etc.) are stored as columns.
- All attributes from the ER diagram appear directly in the table.

The foreign key enforces the rule that every admission must belong to an existing patient.

## 1.3 NOTEEVENTS Table

Clinical notes are implemented in the **NOTEEVENTS** table with:

- **ROW_ID (PK)** uniquely identifying each note.
- **SUBJECT_ID (FK)** → PATIENTS.
- **HADM_ID (FK)** → ADMISSIONS.

This setup enforces the ER diagram's relationships:

- One patient can have multiple notes.
- One admission can produce multiple notes.

The presence of both foreign keys preserves the dual linkage shown in the ER model. Narrative attributes (**E_TEXT**, **CATEGORY**, **E_DESCRIPTION**) are stored as text fields.

## 1.4 DIAGNOSES_ICD Table

This table represents the assignment of diagnoses to a patient during an admission. It includes:

- **ROW_ID (PK)**.
- **SUBJECT_ID (FK)** → PATIENTS.
- **HADM_ID (FK)** → ADMISSIONS.
- **ICD9_CODE (FK)** → D_ICD_DIAGNOSES.

This design reflects the ER diagram's **one-to-one** relationship between diagnoses and the ICD dictionary, as well as the **1-to-many** relationships between admissions and their diagnoses.

The presence of three foreign keys guarantees referential integrity across multiple entities.

## 1.5 D_ICD_DIAGNOSES Table

This dictionary table stores definitions of ICD codes. It maps directly to its ER diagram entity with:

- **ICD9_CODE (PK)**.
- **SHORT_TITLE**, **LONG_TITLE**.

This table is never updated by user operations and serves as a static reference, consistent with the design of the ER model.

## 1.6 ICUSTAYS Table

ICU stays correspond directly to the **ICUSTAYS** entity. The table includes:

- **ICUSTAY_ID (PK)**.
- **SUBJECT_ID (FK)** and **HADM_ID (FK)** to enforce:
    - a patient can have multiple ICU stays,
    - an admission can have multiple ICU stays.

Attributes for ICU tracking (e.g., **DBSOURCE**, **FIRST_CAREUNIT**, **LAST_CAREUNIT**, **INTIME**, **OUTTIME**, **LOS**) map exactly from the ER diagram.

This table completes the representation of the two-level dependency of ICU stays: each belongs to both a patient and an admission, as shown in the ER diagram.

# 2. Implementation of Relationships and Cardinalities

## 2.1 Primary Keys

Each table contains a well-defined primary key:

- PATIENTS(SUBJECT_ID)
- ADMISSIONS(HADM_ID)
- NOTEEVENTS(ROW_ID)
- DIAGNOSES_ICD(ROW_ID)
- D_ICD_DIAGNOSES(ICD9_CODE)
- ICUSTAYS(ICUSTAY_ID)

These keys ensure each row is uniquely identifiable, matching the identifiers defined in the ER diagram.

## 2.2 Foreign Keys and Cardinality Enforcement

The ER diagram's cardinalities were implemented through foreign keys:

| Relationship | Cardinality | Implementation |
|---|---|---|
| PATIENTS → ADMISSIONS | 1-to-many | ADMISSIONS.SUBJECT_ID FK |
| PATIENTS → NOTEEVENTS | 1-to-many | NOTEEVENTS.SUBJECT_ID FK |
| PATIENTS → DIAGNOSES_ICD | 1-to-many | DIAGNOSES_ICD.SUBJECT_ID FK |
| PATIENTS → ICUSTAYS | 1-to-many | ICUSTAYS.SUBJECT_ID FK |
| ADMISSIONS → NOTEEVENTS | 1-to-many | NOTEEVENTS.HADM_ID FK |

| | | |
|---|---|---|
| ADMISSIONS → DIAGNOSES_ICD | 1-to-many | DIAGNOSES_ICD.HADM_ID FK |
| ADMISSIONS → ICUSTAYS | 1-to-many | ICUSTAYS.HADM_ID FK |
| DIAGNOSES_ICD → D_ICD_DIAGNOSES | many-to-1 | DIAGNOSES_ICD.ICD9_CODE FK |

This ensures that:

- No admission exists without a patient.
- No diagnosis exists without both a patient and an admission.
- No note exists without the corresponding patient and admission.
- Each diagnosis references a valid ICD9 entry.

# 3. Data Types and Constraints

The RDBMS implementation uses types that match the nature of the data:

- **INT** for identifiers and counters.
- **VARCHAR** for categorical attributes.
- **DATETIME** for temporal medical information.
- **TEXT / MEDIUMTEXT** for narrative clinical notes.
- **TINYINT** for binary flags.

These choices ensure consistency, readability, and storage efficiency.

Constraints such as `NOT NULL`, default timestamps, and structured foreign keys reproduce the mandatory/optional nature of attributes from the ER model.

# 4. Normalization and Design Justification

The database is fully normalized:

- No redundant storage of ICD descriptions (they reside in a single dictionary table).
- Admissions, stays, diagnoses, and notes are separated into their own tables.
- Referential integrity prevents invalid patient, admission, or diagnosis references.

This normalization is directly derived from the ER diagram and ensures high data quality in the relational implementation.

## 5. Summary

The relational schema reproduces the conceptual ER model, capturing all entities, attributes, and relationships with appropriate keys and constraints. The structure supports efficient SQL querying and ensures referential integrity while maintaining logical consistency with the ER diagram. This schema also forms the basis for Phase II, where the same data is transformed into a more flexible NoSQL representation.

# Document Database Justification

## 1. Justification

For this system, a Document Database was used because it is scalable like a Key-Value database, but also understands the internal structure of documents. Instead of storing data as simple values, a document database stores data in structured formats such as JSON and allows fields within a document to be queried and indexed. This is well-suited for a hospital application because medical datasets contain unstructured and semi-structured data (such as doctor notes, reports, and discharge summaries), which can be stored together within a single patient document. Each document is also associated with a unique key, similar to a Key-Value database, making data retrieval efficient.

## 2. Advantages and disadvantages

The advantages of using a document database for this system is the flexibility of the schema since hospital systems will continue to evolve. New tests, codes and reports will be added and records will often change, and the ability to add new data fields without redesigning an entire database is a big advantage. It is also easy to understand as it is patient-centric, where each patient is stored as one document that contains all of its information. This also makes it have very fast access to patient records since no JOIN operations are needed. However, document databases have weak support, if any, for relationships between data. Because they are not relational, and to avoid slow JOIN operations, data is duplicated across documents which waste storage space. There are also no schema rules that are enforced, since it is able to deal with unstructured/semi-structured data, which can lead to dirty data and can allow for invalid/inconsistent field formats.

# 3. Conversion Steps

### Step 1: Identify Tables and Collections

Each RDBMS table can be treated as a potential collection in the document database. We first identified all the tables in the SQL schema and analyzed their relationships. SQL tables:

- PATIENTS - main patient information
- ADMISSIONS - hospital visits linked to each patient
- ICUSTAYS - ICU records for each admission
- DIAGNOSES_ICD - diagnoses assigned during an admission
- NOTEEVENTS - clinical notes linked to each admission

At this stage, each table can theoretically become its own collection. However, we also determined which entities should be self-contained inside a document rather than separated. In our dataset, a patient acts as the natural parent entity, and all other tables represent data that is tightly linked to that patient's admission history. This makes the patient document the ideal container for storing these:

- Admissions
- ICU stays
- Diagnoses
- Notes

This avoids the need for multiple collections and allows us to embed a complete patient history in a single self-contained document.

Example Mapping:

- Table PATIENTS -> Collection: patients
- Tables: ADMISSIONS, ICUSTAYS, DIAGNOSES_ICD, NOTEEVENTS -> embedded in patient document
- Table: D_ICD_DIAGNOSES -> used for lookups during processing, so not needed as a collection

### Step 2: Choose Document Key

We selected a unique identifier for the document in the MongoDB collection. We chose the patient's subject_id to be the document key of the patients collection. It uniquely identifies each patient, it naturally groups all admissions under the same patient, and it simplifies queries.

Example:

- Key = _id: 2002 for a patient document

### Step 3: Create Document Structure

We merged all relevant SQL fields into one self-contained patient document, organizing the information in a hierarchical JSON format.

A patient document includes:

- Patient's information (gender, DOB, etc.)
- An array of Admissions
- Each Admission contains
    - Diagnoses
    - ICU stays
    - Notes

This ensures the entire patient can be retrieved with one query.

Example - Patient Document Structure

```
{
  "_id": 123,
  "gender": "M",
  "admissions": [
    {
      "hadm_id": 456,
      "admit_time": "2011-04-10",
      "diagnoses": [...],
      "icustays": [...],
      "notes": [...]
    }
  ]
}
```

### Step 4: Map Relationships into Documents

In this step, relational tables connected through foreign keys are now embedded as nested objects inside the main patient document. This replaces the need for the SQL joins and creates a fully self-contained document structure. To do such, we have to identify all the relationships from our SQL database.

| Relationship | Cardinality | Foreign Key | Meaning |
|---|---|---|---|
| PATIENTS → ADMISSIONS | 1-to-many | ADMISSIONS.SUBJECT_ID FK | A patient can have multiple admissions. |
| PATIENTS → NOTEEVENTS | 1-to-many | NOTEEVENTS.SUBJECT_ID FK | All notes belong to the patient. |
| PATIENTS → DIAGNOSES_ICD | 1-to-many | DIAGNOSES_ICD.SUBJECT_ID FK | All diagnoses belong to the patient. |
| PATIENTS → ICUSTAYS | 1-to-many | ICUSTAYS.SUBJECT_ID FK | All ICU stays belong to the patient. |
| ADMISSIONS → NOTEEVENTS | 1-to-many | NOTEEVENTS.HADM_ID FK | Notes are tied to a specific admission. |

| | | | |
|---|---|---|---|
| ADMISSIONS → DIAGNOSES_ICD | 1-to-many | DIAGNOSES_ICD.HADM_ID FK | Diagnoses are tied to a specific admission. |
| ADMISSIONS → ICUSTAYS | 1-to-many | ICUSTAYS.HADM_ID FK | ICU stays occur during a specific admission. |
| DIAGNOSES_ICD → D_ICD_DIAGNOSES | many-to-1 | DIAGNOSES_ICD.ICD9_CODE FK | Diagnoses lookup for code descriptions. |

Using these relationships, we embed all dependent entities directly inside the patient document:

- Admissions becomes an array under each patient
- Inside each admission, we embed:
  - Diagnoses (expanded with the lookup table from D_ICD_DIAGNOSES)
  - ICU stays
  - Notes

Therefore, instead of navigating multiple collections and needing references, we embed them into a single hierarchical structure:

```
Patient
  └── admissions[]
      ├── diagnoses[]        (with ICD descriptions)
      ├── icustays[]
      └── notes[]
```

All related entities belong exclusively to a single patient and are not reused across documents. This allows everything to be embedded without using references, keeping documents simple, readable, and efficient for our case.

### Step 5: Insert Documents into Collection

After constructing the final hierarchical patient documents, we inserted each document into the patients collection in MongoDB. Each document is stored using the selected key (subject_id) and contains all embedded admissions, ICU stays, diagnoses and notes.

Example:

```
Collection: patients
Key: 123
Value: { ...nested patient document shown in steps 3 and 4... }
```

After inserting the documents into the patients collection, we verified the migration using a combination of direct queries in MongoDB. We queried by _id (subject_id) and also filtered on embedded fields such as diagnosis codes, ICU care units, and notes to confirm that the nested structure was correct.

# 4. Implementation

The migration was implemented in Python using a virtual environment to keep dependencies isolated. We used PyMySQL to read data from SQL database (HSP_DB) and PyMongo to write transformed documents into MongoDB. The core implementation follows a standard ETL (Extract-Transform-Load) process.

### 4.1 Extract

The script, attached in the Appendix, used a PyMySQL cursor to execute SQL queries and retrieve each data from each table in HSP_DB. The cursor allows the script to issue SELECT statements and return the results as Python dictionaries, making it easier to join tables and assemble the final document structure. This migration begins by querying all major SQL tables:

- PATIENTS
- ADMISSIONS
- ICUSTAYS
- DIAGNOSES_ICD
- NOTEEVENTS
- D_ICD_DIAGNOSES.

Each table was fetched separately and stored in memory. Foreign keys such as SUBJECT_ID and HADM_ID were used as the abscess for linking entities during the transformation phase.

### 4.2 Transform

In the transformation stage, we reconstructed the relational structure into a hierarchical document model suitable for MongoDB. Rather than performing SQL joins directly in MySQL, the script manually recreated them by matching rows using foreign keys. For each patient (SUBJECT_ID), the script collected all corresponding admissions. For each admission (HADM_ID), it then gathered all related ICU stays, diagnosis entries, and clinical notes. This replicates the effect of multiple SQL joins but restructures the data into nested Python dictionaries and lists. A key part of the transformation involved combining DIAGNOSES_ICD with the lookup table D_ICD_DIAGNOSES to enrich each diagnosis entry. Below is a small excerpt from the script (full code in the Appendix) showing how this join was performed:

```
def get_diagnoses(subject_id, hadm_id):
    cursor.execute("""
        SELECT d.SEQ_NUM, d.ICD9_CODE, dd.SHORT_TITLE, dd.LONG_TITLE
        FROM DIAGNOSES_ICD d
        JOIN D_ICD_DIAGNOSES dd ON d.ICD9_CODE = dd.ICD9_CODE
        WHERE d.SUBJECT_ID=%s AND d.HADM_ID=%s
```

```
    """, (subject_id, hadm_id))
    return cursor.fetchall()
```

The join is done by matching the shared key ICD9_CODE, and the cursor returns the combined result for each diagnosis. This gives us, in one query:

- the patient's diagnosis code
- the sequence number
- the short title of the diagnosis
- the long/official description.

After retrieving this joined data, this diagnosis object was then embedded inside the admission. This step effectively replaces the many-to-one SQL relationship with a fully self-contained embedded object in MongoDB.

After gathering all related data, the script assembled a single hierarchical patient documetn containing:

- Patient's information (gender, DOB, etc.)
- An array of Admissions
- Each Admission contains
    - Diagnoses
    - ICU stays
    - Notes

This transformation denormalized the SQL structure and prepares it for insertion into MongoDB.

**4.3 Load**

After reconstructing each patient's document, with admissions, ICU stays, diagnosis details, and notes embedded, the script inserted the completed document into the MongoDB patients collection using PyMongo. The script cleared the collection before each run to prevent duplicates. Once all documents were inserted, we verified the structure by running test queries and inspecting sample documents in MongoDB Compass. This confirmed that all relationships were preserved and correctly embedded, and that the final document model matched the intended structure described in the Conversion steps. The complete migration script is included in the Appendix.

# Discussion and Analysis:

The principal challenge encountered when working with a **relational database management system (RDBMS)** is that some of the data could not be completely and properly imported in the hospital database. It was due to the fact that the data provided in some tables was too large and the format of the data was not appropriate. For instance, the **NOTEEVENTS** table contains all the notes for a patient. Therefore, it contains a lot of data which made mass imports fail many times. Also, relational databases are not designed and optimized to store unstructured data like human-written notes. Therefore, the data from **NOTEEVENTS** table required some pre-processing to treat issues with reserved delimiters for **SQL** like commas. Lastly, the unstructured nature of some data made some of the querying more difficult. Relationships across multiple tables are also costly in terms of query performance.

While **NoSQL** was better for storing unstructured data, a few challenges have to be mentioned. First, in order to migrate the data from an **RDBMS** to a **NoSQL** one, certain factors had to be considered. Since a **NoSQL DBMS** is much more flexible, it is possible to choose for tables of the dataset to be a collection of their own or embedded inside another collection. For instance, the choice that had to be made was: should the table have its own collection or should it be contained under another collection. In a **RDBMS**, there is no dilemma, everything is a table. Furthermore, the nesting depth had to be optimized between embedding some tables under other collections to respect relationships & interconnected data and query performance. The deeper a table is nested, the slower and more complex the query gets.

The lesson to learn is that both **NoSQL** and relational database management systems have their advantages and their inconveniences. A relational database management system is better for data integrity and security as it is more strict. It has strong relationships between the tables which is better for interconnected data. It requires a strong structure and if the data does not fit the mold it is rejected. Therefore it is less flexible, and some problems can be encountered using unstructured data. A **NoSQL** database management system has the capacity to hold a large amount of unstructured data. It is much more scalable, but there is less emphasis on relationships, data integrity and security. Therefore, one should choose their system depending on their needs or adopt a hybrid system where some data is treated by a relational database management system and others by a **NoSQL** database management system.

# Conclusion

This project demonstrated a part of data engineering from database model conceptualisation with the ER diagram, to actually implementing the database using two different types of database management systems. It also demonstrated how a relational database management system and a **NoSQL** system are inherently different. They are used for different purposes, each having its strengths and weaknesses. It was also possible to notice some differences in performance such as storage efficiency and querying performances.

The first phase of the project addressed relational databases. It needed the design of an ER Diagram respecting the relationships between the hospital dataset's tables. Then, it was possible to translate the ER diagram into an **SQL** script to create the most precise and accurate representation of the dataset's tables and their relationships ensuring data integrity. Finally, the data had to be imported in order to test querying performance.

The second phase of the project addressed **NoSQL** databases. A script to migrate from a relational database to a **NoSQL** had to be written. Factors like which tables are collections of their own and which are embedded as well as at which level of depth should the nesting go were important challenges faced.

# Appendix A: Implementation Scripts

### A.1 - Python Data Migration Script

```
import pymysql
from pymongo import MongoClient

# MySQL connection
print("Connecting to MySQL...")
conn = pymysql.connect(
    host='localhost',
    user='root',
    password='',
    database='HSP_DB',
    charset='utf8mb4'
)
cursor = conn.cursor(pymysql.cursors.DictCursor)
print("Connected to MySQL ✅")

# MongoDB connection
print("Connecting to MongoDB...")
mongo = MongoClient("mongodb://localhost:27017/")
db = mongo["HSP_NoSQL"]
collection = db["patients"]
print("Connected to MongoDB ✅")

collection.drop()  # clear old data

# Helper queries
def get_admissions(subject_id):
    cursor.execute("SELECT * FROM ADMISSIONS WHERE SUBJECT_ID=%s",
(subject_id))
    return cursor.fetchall()

def get_icustays(subject_id, hadm_id):
    cursor.execute("SELECT * FROM ICUSTAYS WHERE SUBJECT_ID=%s AND
HADM_ID=%s",
                   (subject_id, hadm_id))
    return cursor.fetchall()

def get_diagnoses(subject_id, hadm_id):
    cursor.execute("""
```

```
        SELECT d.SEQ_NUM, d.ICD9_CODE, dd.SHORT_TITLE, dd.LONG_TITLE
        FROM DIAGNOSES_ICD d
        JOIN D_ICD_DIAGNOSES dd ON d.ICD9_CODE = dd.ICD9_CODE
        WHERE d.SUBJECT_ID=%s AND d.HADM_ID=%s
    """, (subject_id, hadm_id))
    return cursor.fetchall()

def get_notes(subject_id, hadm_id):
    cursor.execute("""
        SELECT ROW_ID, CHARTDATE, CHARTTIME, STORETIME,
               CATEGORY, E_DESCRIPTION, CGID, ISERROR, E_TEXT
        FROM NOTEEVENTS
        WHERE SUBJECT_ID=%s AND HADM_ID=%s
        LIMIT 100
    """, (subject_id, hadm_id))
    return cursor.fetchall()

# Load patients
print("Loading patients…")
cursor.execute("SELECT * FROM PATIENTS")
patients = cursor.fetchall()
print(f"Found {len(patients)} patients ✔")

# --- Migration ---
for p in patients:
    sid = p["SUBJECT_ID"]
    print(f"\nProcessing patient SUBJECT_ID={sid}…")

    patient_doc = {
        "_id": sid,
        "gender": p["GENDER"],
        "dob": str(p["DOB"]),
        "dod": str(p["DOD"]) if p["DOD"] else None,
        "expire_flag": p["EXPIRE_FLAG"],
        "admissions": []
    }

    admissions = get_admissions(sid)
    print(f"  {len(admissions)} admissions found")

    for a in admissions:
        hadm = a["HADM_ID"]
        print(f"    Admission HADM_ID={hadm}")
```

```
        adm_doc = {
            "hadm_id": hadm,
            "admittime": str(a["ADMITTIME"]),
            "dischtime": str(a["DISCHTIME"]),
            "admission_type": a["ADMISSION_TYPE"],
            "admission_location": a["ADMISSION_LOCATION"],
            "discharge_location": a["DISCHARGE_LOCAITON"],
            "insurance": a["INSURANCE"],
            "diagnosis": a["DIAGNOSIS"],
            "hospital_expire_flag": a["HOSPITAL_EXPIRE_FLAG"],
            "has_chartevents_data": a["HAS_CHARTEVENTS_DATA"],
            "icu_stays": [],
            "diagnoses": [],
            "notes": []
        }

        adm_doc["icu_stays"] = get_icustays(sid, hadm)
        adm_doc["diagnoses"] = get_diagnoses(sid, hadm)
        adm_doc["notes"] = get_notes(sid, hadm)

        patient_doc["admissions"].append(adm_doc)

    collection.insert_one(patient_doc)
    print("    Inserted into MongoDB ✅")

print("\n-------------------------------------")
print("        MIGRATION COMPLETED      ")
print("-------------------------------------")
```

### A.2 - SQL Database Creation Script

```
DROP  DATABASE hsp_db;

CREATE DATABASE HSP_DB;
USE HSP_DB;

CREATE TABLE PATIENTS(
    ROW_ID INT NOT NULL,
    SUBJECT_ID INT PRIMARY KEY,
    GENDER VARCHAR(5) NOT NULL,
    DOB DATETIME  NOT NULL,
    DOD DATETIME  NULL,
    DOD_HOSP DATETIME  NULL,
    DOD_SSN DATETIME  NULL,
    EXPIRE_FLAG VARCHAR(5) NULL
);

CREATE TABLE ADMISSIONS(
    ROW_ID INT NOT NULL,
    SUBJECT_ID INT NOT NULL, -- Foreign Key
    HADM_ID INT PRIMARY KEY,
    ADMITTIME DATETIME  NOT NULL,
    DISCHTIME DATETIME  NOT NULL DEFAULT CURRENT_TIMESTAMP ,
    DEATHIME DATETIME  NULL,
    ADMISSION_TYPE VARCHAR(50) NOT NULL,
    ADMISSION_LOCATION VARCHAR(50) NOT NULL,
    DISCHARGE_LOCAITON VARCHAR(50) NOT NULL,
    INSURANCE VARCHAR(255) NULL,
    LANG VARCHAR(10) NOT NULL,
    RELIGION VARCHAR(50) NULL,
    MARITAL_STATUS VARCHAR(50) NULL,
    ETHNICITY VARCHAR(200) NULL,
    EDREGTIME DATETIME  NULL,
    EDOUTTIME DATETIME  NULL,
    DIAGNOSIS VARCHAR(300) NOT NULL,
    HOSPITAL_EXPIRE_FLAG TINYINT NOT NULL DEFAULT 0,
    HAS_CHARTEVENTS_DATA TINYINT NOT NULL,
    CONSTRAINT fk_admissions_patients
    FOREIGN KEY (SUBJECT_ID) REFERENCES PATIENTS(SUBJECT_ID)
);

CREATE TABLE NOTEEVENTS(
```

```
    ROW_ID INT PRIMARY KEY,
    SUBJECT_ID INT NOT NULL, -- Foreign key
    HADM_ID INT NOT NULL, -- Foreign key
    CHARDATE DATETIME  NOT NULL,
    CHARTTIME DATETIME  NULL,
    STORETIME DATETIME  NULL,
    CATEGORY VARCHAR(50) NOT NULL,
    E_DESCRIPTION VARCHAR(300) NOT NULL, -- E_ for EVENTS
    CGID INT NOT NULL,
    ISERROR CHAR(1) NULL,
    E_TEXT MEDIUMTEXT NOT NULL,
    CONSTRAINT fk_noteevent_patients
    FOREIGN KEY (SUBJECT_ID) REFERENCES PATIENTS(SUBJECT_ID),
    CONSTRAINT fk_noteevents_admissions
    FOREIGN KEY (HADM_ID) REFERENCES ADMISSIONS(HADM_ID)
);

CREATE TABLE D_ICD_DIAGNOSES(
    ROW_ID INT NOT NULL,
    ICD9_CODE VARCHAR(10) PRIMARY KEY,
    SHORT_TITLE VARCHAR(10) NOT NULL,
    LONG_TITLE VARCHAR(300) NOT NULL
);

CREATE TABLE DIAGNOSES_ICD(
    ROW_ID INT PRIMARY KEY,
    SUBJECT_ID INT NOT NULL, -- Foreign key
    HADM_ID INT NOT NULL, -- Foreign Key
    SEQ_NUM INT NULL,
    ICD9_CODE VARCHAR(10) NOT NULL,-- Foreign Key
    CONSTRAINT fk_diagnosesicd_patients
    FOREIGN KEY (SUBJECT_ID) REFERENCES PATIENTS(SUBJECT_ID),
    CONSTRAINT fk_diagnosesicd_admissions
    FOREIGN KEY (HADM_ID) REFERENCES ADMISSIONS(HADM_ID),
    CONSTRAINT fk_diagnosesicd_dicddiagnoses
    FOREIGN KEY (ICD9_CODE) REFERENCES D_ICD_DIAGNOSES(ICD9_CODE)
);

CREATE TABLE ICUSTAYS(
    ROW_ID INT NOT NULL,
    SUBJECT_ID INT NOT NULL,
    HADM_ID INT NOT NULL,
    ICUSTAY_ID INT PRIMARY KEY,
```

```
    DBSOURCE VARCHAR(20) NOT NULL,
    FIRST_CAREUNIT VARCHAR(20) NOT NULL,
    LAST_CAREUNIT VARCHAR(20) NOT NULL,
    FIRST_WARDID SMALLINT NOT NULL,
    LAST_WARDID SMALLINT NOT NULL,
    INTIME DATETIME NOT NULL,
    OUTTIME DATETIME  NULL,
    LOS DOUBLE PRECISION NULL

);
```

### A.3 - Phase I Queries

```
--query 1 (list all patients)

SELECT SUBJECT_ID, DOB, GENDER
FROM PATIENTS;

--query 2 (admissions of a given patient)
SELECT * FROM ADMISSIONS
WHERE SUBJECT_ID = SELECT * FROM ADMISSIONS
WHERE SUBJECT_ID = 3288;  -- sample id
;

--query 3 (number of admissions for a given patient)
SELECT SUBJECT_ID, COUNT(HADM_ID) AS NUM_ADMISSIONS
FROM ADMISSIONS
GROUP BY SUBJECT_ID;

--query 4 (all patients that were discharged to home)
SELECT DISTINCT SUBJECT_ID
FROM ADMISSIONS
WHERE DISCHARGE_LOCAITON = 'HOME';
--query 5 (patients with private insurance)
SELECT DISTINCT p.SUBJECT_ID, p.DOB, p.GENDER, a.INSURANCE
FROM PATIENTS p
JOIN ADMISSIONS a
    ON p.SUBJECT_ID = a.SUBJECT_ID
WHERE a.INSURANCE = 'Private';

--query 6 (patients transferred from one location to another location)
```

```sql
SELECT DISTINCT p.SUBJECT_ID, i.FIRST_CAREUNIT, i.LAST_CAREUNIT,
i.FIRST_WARDID, i.LAST_CAREUNIT
FROM PATIENTS p
JOIN ICUSTAYS i
    ON p.SUBJECT_ID = i.SUBJECT_ID
WHERE i.FIRST_CAREUNIT <> i.LAST_CAREUNIT
   OR i.FIRST_WARDID <> i.LAST_WARDID;

--query 7 (patients who have > 1 ICU stays in a single hodpital
admission)
SELECT SUBJECT_ID, HADM_ID, COUNT(ICUSTAY_ID) AS TOTAL_ICU_STAYS
FROM ICUSTAYS
GROUP BY SUBJECT_ID, HADM_ID
HAVING COUNT(ICUSTAY_ID) > 1;

--query 8 (patients in the ICU admitted to MICU until they left)
SELECT DISTINCT p.SUBJECT_ID, i.ICUSTAY_ID, i.FIRST_CAREUNIT,
i.LAST_CAREUNIT
FROM PATIENTS p
JOIN ICUSTAYS i
    ON p.SUBJECT_ID = i.SUBJECT_ID
WHERE i.FIRST_CAREUNIT = 'MICU'
  AND i.LAST_CAREUNIT = 'MICU';

-- query 9 (all notes for a specific admission ID with its author and
error if any)
SELECT HADM_ID, SUBJECT_ID, E_DESCRIPTION, E_TEXT, CATEGORY, CGID,
ISERROR
FROM NOTEEVENTS
WHERE HADM_ID = '5002' --sample hospital admission id
-- query 10
SELECT *
FROM NOTEEVENTS
WHERE CATEGORY = 'Discharge summary'
ORDER BY CHARTTIME
LIMIT 10;

-- query 11
SELECT HADM_ID, COUNT(*) AS num_notes
FROM NOTEEVENTS
GROUP BY HADM_ID;

-- query 12 can replace the subject_id with anyone (a given patient)
```

```sql
SELECT d.ICD9_CODE, d.SHORT_TITLE, d.LONG_TITLE
FROM DIAGNOSES_ICD di
JOIN D_ICD_DIAGNOSES d ON di.ICD9_CODE = d.ICD9_CODE
WHERE di.SUBJECT_ID = 101;

-- query 13
SELECT d.ICD9_CODE, d.LONG_TITLE, COUNT(*) AS frequency
FROM DIAGNOSES_ICD di
JOIN D_ICD_DIAGNOSES d ON di.ICD9_CODE = d.ICD9_CODE
GROUP BY d.ICD9_CODE, d.LONG_TITLE
ORDER BY frequency DESC
LIMIT 5;

-- query 14
SELECT DISTINCT a.*
FROM ADMISSIONS a
JOIN ICUSTAYS i ON a.HADM_ID = i.HADM_ID
JOIN DIAGNOSES_ICD di ON a.HADM_ID = di.HADM_ID
WHERE di.ICD9_CODE = '401.9';

-- query 15
SELECT SUBJECT_ID FROM PATIENTS
WHERE SUBJECT_ID NOT IN (SELECT DISTINCT SUBJECT_ID FROM ICUSTAYS);

-- query 16
SLECT DISTINCT p.SUBJECT_ID, p.GENDER, p.DOB FROM PATIENTS
p INNER JOIN NOTEEVENTS n ON p.SUBJECT_ID WHERE n.CATEGORY =
'Radiology';

-- query 17
SELECT DISTINCT p.SUBJECT_ID, p.GENDER, p.DOB FROM PATIENTS p
INNER JOIN NOTEEVENTS n ON p.SUBJECT_ID = n.SUBJECT_ID WHERE n.CATEGORY
= 'Radiology' AND
(n.E_DESCRIPTION LIKE '%CHEST%' OR n.E_TEXT LIKE '%chest$');

-- query 18
SLECT DISTINCT p.SUBJECT_ID, p.GENDER, p.DOB FROM PATIENTS
p INNER JOIN NOTEEVENTS n ON p.SUBJECT_ID = n. n.SUBJECT_ID WHERE
n.CATEGORY = 'Discharge summary';

-- query 19
SELECT DISTINCT p.SUBJECT_ID, p.GENDER, p.DOB, n.CATEGORY
FROM PATIENTS
```

```
p INNER JOIN NOTEEVENTS n ON p.SUBJECT_ID = n.SUBJECT_ID
WHERE n.CATEGORY IN ('Radiology', 'ECG') ORDER BY p.SUBJECT_ID,
n.CATEGORY;

-- query 20
SELECT
    (SELECT COUNT(HADM_ID) FROM ADMISSIONS WHERE SUBJECT_ID = 26) AS
Total_Admissions,
    (SELECT COUNT(ICUSTAY_ID) FROM ICUSTAYS WHERE SUBJECT_ID = 26) AS
Total_Icu_Stays,
    (SELECT COUNT(ICD9_CODE) FROM DIAGNOSES_ICD WHERE SUBJECT_ID = 26)
AS Total_Diagnoses;
```