Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Bases de Dados NoSQL

Ano lectivo 2023/2024

Grupo 30

Abhimanyu Aryan, PG516325

Ana Rita Santos Poças, PG53645

Catarina Teves Martins da Costa, PG52676

Marta de Campos Loureiro e Barros Aguiar, PG52694

Nelson José Marques Martins Almeida, PG52697

4 de Junho de 2024

Index

Figures Index

1 Introduction

This practical assignment is part of the Bases de Dados NoSQL Curricular Unit, belonging to the Master in Software Engineering from the Universidade do Minho. The assignment had the motivation to allow us to create skills, namely in the use of different database paradigms and their application in systems design and implementation.

The aim of this practical is to carry out a work of analysis, planning, implementation of a relation and two non-relational DBMS, MongoDB and Neo4j. In order to accomplish this task, we were supplied an Oracle database that represents a fictitious hospital and includes various database objects.

The theme of the practical assignment is a Hospital Management System relational database.

1.1 Contextualization

The healthcare sector faces unique challenges in efficiently managing large volumes of data related to patients, medical records, treatments, staff, and various hospital operations. The choice of proper database models can definitely impact the performance, scalability, and overall user experience within a hospital management system.

Database systems have developed beyond traditional relational models due to technological advancements and increased need for efficient data storage and processing. NoSQL databases have become available, providing options to meet various data requirements and formats. Hospital management involves the management of a variety of data, such as patient records, medical histories, appointments, treatments, and staff information.

The database provided is built in Oracle and serves as an imaginary hospital management system, containing multiple entities. These entities are connected and offer details on patients, their medical backgrounds, treatments, as well as staff, departments, hospital rooms, medications, and medical examinations. A mix of relational and non-relational database models was utilized to efficiently handle the data. The use of Oracle to implement the relational model guarantees organized data storage and intricate query capabilities. Furthermore, two alternative models were selected: a document-based model with MongoDB and a graph-based model with Neo4j. These models address various aspects of managing hospital data, improving performance, scalability, and efficiency of data retrieval.

1.2 Structure of the report

This report starts with an introduction, where we explain the context and the main purpose of this assignment.

After, we have a section ”Data Provided”, where we describe the data provided by the professors.

The section ”Data Migration”is divided into two parts: MongoDB and Neo4j. Each explains how the migration was done in both databases.

We then have the section ”Queries”where we show examples of the queries implemented in our Non Relational databases.

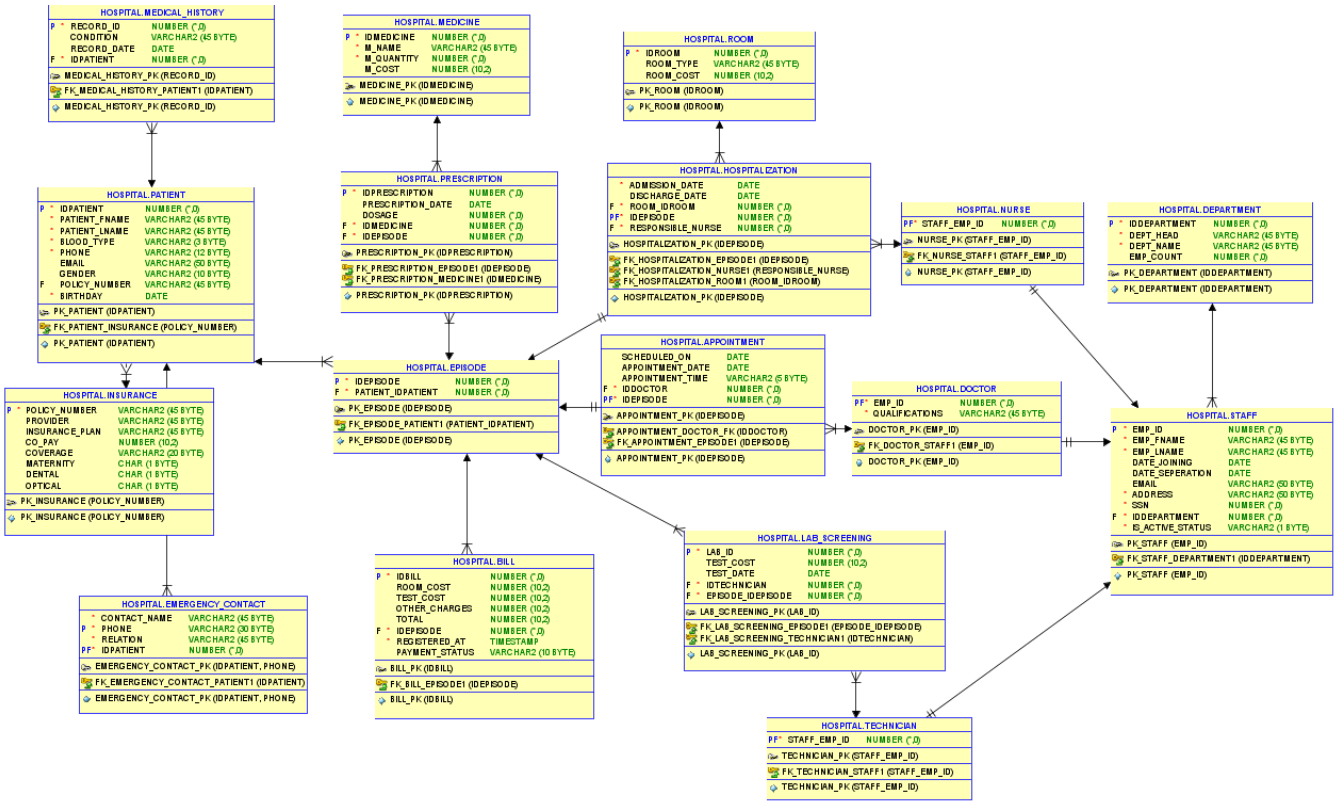
Finally, we have the section "Conclusion", where we talk about the work overall.

2 Data Provided

The Oracle data base provided by the professors represents the management of a hospital system and includes various database objects. The database provided has the following tables:

- **appointment**: stores information about medical appointments.
- **bill**: stores billing information related to medical episodes.
- **department**: stores information about different departments within the hospital.
- **doctor**: stores information about doctors within the hospital.
- **emergency_contact**: stores emergency contact information for patients.
- **episode**: stores information about medical episodes associated with patients.
- **hospitalization**: stores information about patient hospitalizations.
- **insurance**: stores information about patient's insurance plans.
- **lab_screening**: stores information about lab screenings conducted for medical episodes.
- **MEDICAL_HISTORY**: stores medical history records for patients.
- **medicine**: stores information about different medicines.
- **nurse**: stores information about the nurses in the hospital.
- **patient**: stores personal and contact information for patients, with each patient uniquely identified by their patient ID.
- **prescription**: stores details about prescriptions issued during medical episodes, including the prescription date, dosage, associated medicine, and episode ID.
- **room**: stores information about rooms in the hospital, including room type and cost, with each room uniquely identified by its room ID.
- **staff**: stores personal, contact, and employment details for staff members, including department affiliation and active status, with each staff member uniquely identified by their employee ID.
- **technician**: stores information about technicians.

And the physical model of the data base looks like this:


HOSPITAL.MEDICAL_HISTORY
P * RECORD_ID NUMBER (*,0)
CONDITION VARCHAR2 (45 BYTE)
RECORD_DATE DATE
F * IDPATIENT NUMBER (*,0)
MEDICAL_HISTORY_PK (RECORD_ID)
FK_MEDICAL_HISTORY_PATIENT1 (IDPATIENT)
MEDICAL_HISTORY_PK (RECORD_ID)
HOSPITAL.MEDICINE
P * IDMEDICINE NUMBER (*,0)
* M_NAME VARCHAR2 (45 BYTE)
* M_QUANTITY NUMBER (*,0)
M_COST NUMBER (10,2)
MEDICINE_PK (IDMEDICINE)
MEDICINE_PK (IDMEDICINE)
HOSPITAL.ROOM
P * IDROOM NUMBER (*,0)
ROOM_TYPE VARCHAR2 (45 BYTE)
ROOM_COST NUMBER (10,2)
PK_ROOM (IDROOM)
PK_ROOM (IDROOM)
HOSPITAL.PATIENT
P * IDPATIENT NUMBER (*,0)
* PATIENT_FNAME VARCHAR2 (45 BYTE)
* PATIENT_LNAME VARCHAR2 (45 BYTE)
* BLOOD_TYPE VARCHAR2 (3 BYTE)
* PHONE VARCHAR2 (12 BYTE)
EMAIL VARCHAR2 (50 BYTE)
GENDER VARCHAR2 (10 BYTE)
F POLICY_NUMBER VARCHAR2 (45 BYTE)
* BIRTHDAY DATE
PK_PATIENT (IDPATIENT)
FK_PATIENT_INSURANCE (POLICY_NUMBER)
PK_PATIENT (IDPATIENT)
HOSPITAL.PRESCRIPTION
P * IDPRESCRIPTION NUMBER (*,0)
PRESCRIPTION_DATE DATE
DOSAGE NUMBER (*,0)
F * IDMEDICINE NUMBER (*,0)
F * IDEPISODE NUMBER (*,0)
PRESCRIPTION_PK (IDPRESCRIPTION)
FK_PRESCRIPTION_EPISODE1 (IDEPISODE)
FK_PRESCRIPTION_MEDICINE1 (IDMEDICINE)
PRESCRIPTION_PK (IDPRESCRIPTION)
HOSPITAL.HOSPITALIZATION
* ADMISSION_DATE DATE
DISCHARGE_DATE DATE
F * ROOM_IDROOM NUMBER (*,0)
PF* IDEPISODE NUMBER (*,0)
F * RESPONSIBLE_NURSE NUMBER (*,0)
HOSPITALIZATION_PK (IDEPISODE)
FK_HOSPITALIZATION_EPISODE1 (IDEPISODE)
FK_HOSPITALIZATION_NURSE1 (RESPONSIBLE_NURSE)
FK_HOSPITALIZATION_ROOM1 (ROOM_IDROOM)
HOSPITALIZATION_PK (IDEPISODE)
HOSPITAL.NURSE
PF* STAFF_EMP_ID NUMBER (*,0)
NURSE_PK (STAFF_EMP_ID)
FK_NURSE_STAFF1 (STAFF_EMP_ID)
NURSE_PK (STAFF_EMP_ID)
HOSPITAL.DEPARTMENT
P * IDDEPARTMENT NUMBER (*,0)
* DEPT_HEAD VARCHAR2 (45 BYTE)
* DEPT_NAME VARCHAR2 (45 BYTE)
EMP_COUNT NUMBER (*,0)
PK_DEPARTMENT (IDDEPARTMENT)
PK_DEPARTMENT (IDDEPARTMENT)
HOSPITAL.EPISODE
P * IDEPISODE NUMBER (*,0)
F * PATIENT_IDPATIENT NUMBER (*,0)
PK_EPISODE (IDEPISODE)
FK_EPISODE_PATIENT1 (PATIENT_IDPATIENT)
PK_EPISODE (IDEPISODE)
HOSPITAL.APPOINTMENT
SCHEDULED_ON DATE
APPOINTMENT_DATE DATE
APPOINTMENT_TIME VARCHAR2 (5 BYTE)
F * IDDOCTOR NUMBER (*,0)
PF* IDEPISODE NUMBER (*,0)
APPOINTMENT_PK (IDEPISODE)
APPOINTMENT_DOCTOR_FK (IDDOCTOR)
FK_APPOINTMENT_EPISODE1 (IDEPISODE)
APPOINTMENT_PK (IDEPISODE)
HOSPITAL.DOCTOR
PF* EMP_ID NUMBER (*,0)
* QUALIFICATIONS VARCHAR2 (45 BYTE)
DOCTOR_PK (EMP_ID)
FK_DOCTOR_STAFF1 (EMP_ID)
DOCTOR_PK (EMP_ID)
HOSPITAL.STAFF
P * EMP_ID NUMBER (*,0)
* EMP_FNAME VARCHAR2 (45 BYTE)
* EMP_LNAME VARCHAR2 (45 BYTE)
DATE_JOINING DATE
DATE_SEPERATION DATE
EMAIL VARCHAR2 (50 BYTE)
* ADDRESS VARCHAR2 (50 BYTE)
* SSN NUMBER (*,0)
F * IDDEPARTMENT NUMBER (*,0)
* IS_ACTIVE_STATUS VARCHAR2 (1 BYTE)
PK_STAFF (EMP_ID)
FK_STAFF_DEPARTMENT1 (IDDEPARTMENT)
PK_STAFF (EMP_ID)
HOSPITAL.INSURANCE
P * POLICY_NUMBER VARCHAR2 (45 BYTE)
PROVIDER VARCHAR2 (45 BYTE)
INSURANCE_PLAN VARCHAR2 (45 BYTE)
CO_PAY NUMBER (10,2)
COVERAGE VARCHAR2 (20 BYTE)
MATERNITY CHAR (1 BYTE)
DENTAL CHAR (1 BYTE)
OPTICAL CHAR (1 BYTE)
PK_INSURANCE (POLICY_NUMBER)
PK_INSURANCE (POLICY_NUMBER)
HOSPITAL.BILL
P * IDBILL NUMBER (*,0)
ROOM_COST NUMBER (10,2)
TEST_COST NUMBER (10,2)
OTHER_CHARGES NUMBER (10,2)
TOTAL NUMBER (10,2)
F * IDEPISODE NUMBER (*,0)
* REGISTERED_AT TIMESTAMP
PAYMENT_STATUS VARCHAR2 (10 BYTE)
BILL_PK (IDBILL)
FK_BILL_EPISODE1 (IDEPISODE)
BILL_PK (IDBILL)
HOSPITAL.LAB_SCREENING
P * LAB_ID NUMBER (*,0)
TEST_COST NUMBER (10,2)
* TEST_DATE DATE
F * IDTECHNICIAN NUMBER (*,0)
F * EPISODE_IDEPISODE NUMBER (*,0)
LAB_SCREENING_PK (LAB_ID)
FK_LAB_SCREENING_EPISODE1 (EPISODE_IDEPISODE)
FK_LAB_SCREENING_TECHNICIAN1 (IDTECHNICIAN)
LAB_SCREENING_PK (LAB_ID)
HOSPITAL.EMERGENCY_CONTACT
* CONTACT_NAME VARCHAR2 (45 BYTE)
P * PHONE VARCHAR2 (30 BYTE)
* RELATION VARCHAR2 (45 BYTE)
PF* IDPATIENT NUMBER (*,0)
EMERGENCY_CONTACT_PK (IDPATIENT, PHONE)
FK_EMERGENCY_CONTACT_PATIENT1 (IDPATIENT)
EMERGENCY_CONTACT_PK (IDPATIENT, PHONE)
HOSPITAL.TECHNICIAN
PF* STAFF_EMP_ID NUMBER (*,0)
TECHNICIAN_PK (STAFF_EMP_ID)
FK_TECHNICIAN_STAFF1 (STAFF_EMP_ID)
TECHNICIAN_PK (STAFF_EMP_ID)

3 Data Migration

In this chapter, we will be explaining, in detail, our approach to migrate the provided data into non relational systems - **MongoDB** and **Neo4j**.

3.1 MongoDB

Considering MongoDB's document-oriented paradigm, we decided to migrate data from a **patient-centric** perspective.

For each patient in the relational database, a document is created in the MongoDB collection. It stores their:

- **ID**
- **Name**
- **Blood type**
- **Email**
- **Gender**
- **Birthday**
- Complete **medical history**
- All **emergency contacts**
- Every **episode** (including screenings, appointments, and hospitalization data)

In section (6) of the report, it is present a sample of a *Patient* document.

3.1.1 Connection

We started our migration procedure by establishing the connection to the **Oracle Database** and **MongoDB Atlas**. We decided to use MongoDB Atlas instead of a local instance of MongoDB, so we could be able to create the triggers.

The following code is a Python code that establishes the mentioned connection based on a option system, using environment variables for credentials and handling exceptions to ensure a proper connection to each database.

```
def connect(self, option):
       if (option == "Oracle" or option==0):
           try:
               load_dotenv()

               user_db = os.getenv('ORACLE_USER')
               password_db = os.getenv('ORACLE_PASSWORD')
```

```
            connection = oracle.connect(user=user_db,
                                        mode=oracle.AUTH_MODE_SYSDBA,
                                        password=password_db,
                                        host="localhost",
                                        port=1521,
                                        service_name="xe"
                                    )


            print("Successfull connection to Oracle Database")
            return connection
        except Exception as e:
            print('Error connecting to the database: ', e)
            return None

    if (option == "MongoDB" or option==1):
        try:
            mongoUserName = os.getenv('MONGO_USER')
            mongoPassword = os.getenv('MONGO_PASSWORD')
            uri = f"mongodb+srv://{mongoUserName}:{mongoPassword}@bdnosql.isx6fkl.mongodb.
                net/?retryWrites=true&w=majority&appName=bdnosql"

            client = MongoClient(uri, server_api=ServerApi('1'))
            client.admin.command('ping')

            print("Successfull connection to MongoDB Database")
            return client
        except Exception as e:
            print('Error connecting to the database')
            print("Error: ", e)
            return None
```

Listing 3.1: Ensure connection to the databases

In case the connection to the databases is successful, we proceed to start the migration process. First, we assert the existence of the tables, triggers, procedures and views. In case they exist we drop them before starting the migration.

In order to retrieve all the procedures, views and triggers from the SQL file our group elaborated regular expressions to capture them for further use in the migration.

Each command in the SQL file is ran individually resorting to the *oracledb* Python library.

3.1.2 Migration

If the SQL script is successfully imported into OracleDB, we start querying the database aiming to create the *Patients* in the MongoDB.

To create the *Patients* documents in MongoDB, we begin by retrieving all records from the *Patients* we begin by retrieving all records from the *Patients table* in the Oracle database. For each patient record, we then perform additional queries against all relevant tables to gather comprehensive information.

The code is responsible for the migration process is in the attachments area (6) of the report.

3.1.3 Create View

The original *hospital.sql* file has a view that aims to provide a comprehensive summary of patient appointments by consolidating relevant information from multiple tables into a single view. It includes details about the appointment, the assigned doctor, the associated department, and the patient's personal information.

This view streamlines the process of querying and retrieving comprehensive appointment data, thereby simplifying the management and analysis of patient appointments, doctor assignments, and departmental workload distribution.

For our MongoDB approach we came up with the following query to implement the view.

```
self.mongoDB.create_collection(
    viewName,
    viewOn = "Patient",
    pipeline=[
    {
        '$unwind': {
            'path': '$episodes'
        }
    }, {
        '$unwind': {
            'path': '$episodes.appointments'
        }
    }, {
        '$project': {
            'appointment_scheduled_date': '$episodes.appointments.scheduled_on',
            'appointment_date': '$episodes.appointments.appointment_date',
            'appointment_time': '$episodes.appointments.appointment_time',
            'doctor_id': '$episodes.appointments.doctor.employee.employee_id',
            'doctor_qualifications': '$episodes.appointments.doctor.qualifications',
            'department_name': '$episodes.appointments.doctor.employee.department.dept_name'
                ,
            'patient_first_name': '$fname',
            'patient_last_name': '$lname',
            'patient_blood_type': '$blood_type',
            'patient_phone': '$phone',
            'patient_email': '$email',
            'patient_gender': '$gender'
        }
    }
    ])
```

Listing 3.2: Mongo DB view equivalent to the SQL view

Let's compare the output from the SQL view with our migration efforts to verify accurate implementation.

```
"appointment_scheduled_date": {
  "$date": "2013-11-20T00:00:00.000Z"
},
"appointment_date": {
  "$date": "2013-12-21T00:00:00.000Z"
},
"appointment_time": "13:13",
"doctor_id": 99,
"doctor_qualifications": "PhD",
"department_name": "Emergency_1",
"patient_first_name": "John",
"patient_last_name": "Doe",
"patient_blood_type": "A+",
"patient_phone": "123-456-7890",
"patient_email": "john.doe@example.com",
"patient_gender": "Male"
```

Listing 3.3: Mongo DB PatientAppointmentView equivalent

```
(datetime.datetime(2013, 11, 20, 0, 0), datetime.datetime(2013, 12, 21, 0, 0), '13:13', 99, '
    PhD', 'Emergency_1', 'John', 'Doe', 'A+', '123-456-7890', 'john.doe@example.com', 'Male')
```

Listing 3.4: SQL PatientAppointmentView

As we can see, the MongoDB view accurately replicates the original intended view within its own framework.

3.1.4 Create trigger

The **Oracle Database** featured a trigger designed to generate bills for patient episodes upon updating the discharge date from null to a specific date.

In order to translate that to MongoDB we created the following trigger resorting to a JavaScript function:

```
exports = async function(changeEvent) {
    const docId = 0;
    const serviceName = "bdnosql";
    const database = "BDNOSQL";
    const collection = context.services.get(serviceName).db(database).collection("Patient");

    try {
      if (changeEvent.operationType === "update") {
        const document = changeEvent.fullDocument;
        const documentBefore = changeEvent.fullDocumentBeforeChange;
        let flag = false;

        for (let i = 0; i < documentBefore.episodes.length; i++) {
          for (let j = 0; j < documentBefore.episodes[i].hospitalizations.length; j++) {
            const hospitalizationBefore = documentBefore.episodes[i].hospitalizations[j];
            const hospitalization = document.episodes[i].hospitalizations[j];
            if (hospitalizationBefore.dischargeDate == null && hospitalization.dischargeDate !=
                null) {
              flag = true;

              let roomCost = hospitalization.room.room_cost;
              let testCost = 0;
              let otherCharges = 0;

              for (let k = 0; k < documentBefore.episodes[i].screenings.length; k++) {
                testCost += documentBefore.episodes[i].screenings[k].screening_cost;
              }

              for (let k = 0; k < documentBefore.episodes[i].prescriptions.length; k++) {
                const prescription = documentBefore.episodes[i].prescriptions[k];
                otherCharges += prescription.medicine.medicine_cost * prescription.dosage;
              }

              let totalCost = roomCost + testCost + otherCharges;

              const billPayload = {
                "bill_id": 1,
                "room_cost": roomCost,
                "test_cost": testCost,
                "other_charges": otherCharges,
                "total_cost": totalCost,
                "register_date": new Date(),
                "payment_status": "PENDING"
              };
```

```
            document.episodes[i].bills.push(billPayload);
            console.log("Document _id: "+docId);
            console.log("episodes."+2+".bills");
            await collection.updateOne(
              { "patient_id" : 2 },
              { "$push" : {
                "episodes.2.bills" : billPayload
                }
              }
            )
            .then(result => {
              console.log("Successfully updated document");
            })
            .catch(error => {
              console.error("Error updating document:", error);
            });
            console.log("Updated bill with values:");
            console.log("Room Cost: "+ roomCost);
            console.log("Testing Cost: "+testCost);
            console.log("Aditional Charges: "+ otherCharges);
            console.log("Total cost of the episode: "+totalCost);
            console.log("Successfully inserted bill for episode: " + document.episodes[i].
                episode_id);
          }
        }
      }

      if (!flag) {
        await collection.replaceOne({ "_id": docId }, document);
        console.log("Document replaced successfully");
      }
    } else {
      console.log("Unsupported operation type: " + changeEvent.operationType);
    }
  } catch(err) {
    console.error("Error performing MongoDB write: ", err.message);
  }
};
```

This function seeks to replicate the functionality of the SQL trigger by checking if the any episode's date has been updated by comparing the documents episodes. If any episode's discharge date has been updated, then we proceed to calculate all the costs involved in the episode like the room cost, testing costs, prescription charges and then we update the document's bills with the new one with the respective data before inserting the update in the database.

Our setup in order to compare documents and use the trigger is as follows:

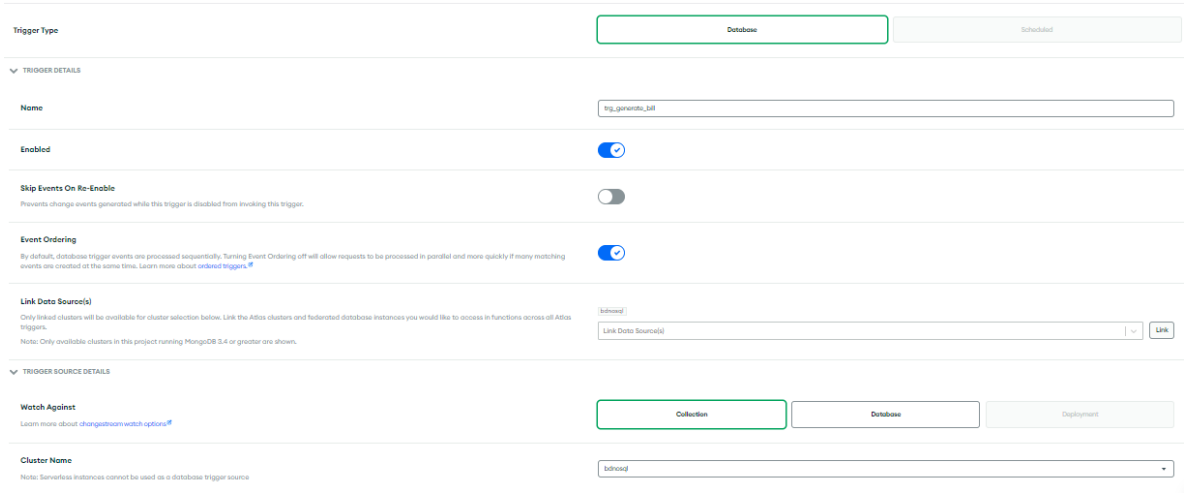


Figura 3.1: MongoDB trigger setup

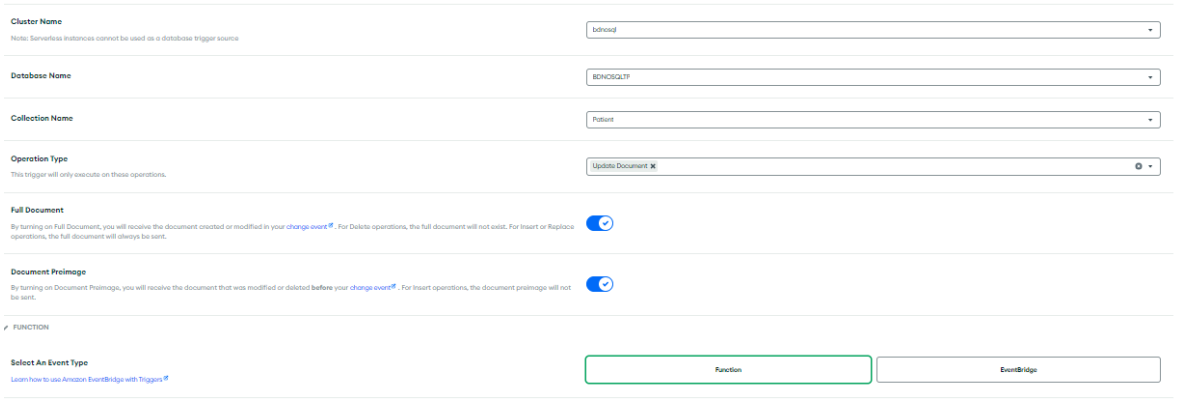


Figura 3.2: Continuation of the MongoDB trigger setup

We wrote debugging commands/logs to the output in order to verify that the information is being updated correctly.

```
"Successfully updated document",
"Updated bill with values:",
"Room Cost: 300",
"Testing Cost: 0",
"Aditional Charges: 3780",
"Total cost of the episode: 4080",
"Successfully inserted bill for episode: 181"
```

Figura 3.3: Logs from the trigger execution

The before and after results can be seen in the attachments area (6) of the report. And as we can see the list of bills has been updated with a pending bill that refers to the total of the episode.

3.2 Neo4j

Since non relation databases do not have tables, but graphs, we have decided to create nodes, and then connect them according to the relationship that exists between them.

In order to accomplish this task, we decided to first follow a ”manual”approach, in order to understand what kind of relationships should exist between the nodes (we first exported the tables to csv, created nodes from there and created the relationships manually), and once we understood that part then we went on to write a script that could automate the process of the migration of the data to neo4j.

3.2.1 Nodes Created

We have created the following nodes:

- DEPARTMENT
- STAFF
- DOCTOR
- TECHNICIAN
- NURSE
- HOSPITALIZATION
- ROOM
- EPISODE
- BILL
- PRESCRIPTION
- MEDICINE
- APPOINTEMENT
- LAB_SCREENING
- PATIENT
- EMERGENCY_CONTACT
- INSURANCE
- MEDICAL_HISTORY

Since Neo4j doesn't have the concept of primary key and we needed it on our system, we were able to implement them using constraints. So for example, in the case of the node ”DEPARTMENT”we used this to make the id of the department its primary key:

```
"DEPARTMENT": {
    "query": """
        CREATE (:Department {
            IDDEPARTMENT: $col0,
            DEPT_HEAD: $col1,
            DEPT_NAME: $col2,
            EMP_COUNT: toInteger($col3)
        })
    """,
    "constraints": "CREATE CONSTRAINT FOR (d:Department) REQUIRE d.IDDEPARTMENT IS UNIQUE;"
```

A similar process was applied to the rest of the nodes in our program.

3.2.2 Relationships

We have decided to establish the following relationships:

- **WORKS_IN**: the relationship indicates that a staff member from the node "STAFF"works in a specific department from the node "DEPARTMENT". It means that a staff works in a department.
- **IS_DOCTOR_FOR**: the relationship connects a doctor from the node "DOCTOR"to their corresponding staff profile "STAFF"node. It means that a doctor is a staff member.
- **IS_TECHNICIAN_FOR**: the relationship connects a technician from the node "TECHNICIAN"to their corresponding staff profile "STAFF"node. It means that a technician is a staff member.
- **IS_NURSE_FOR**: the relationship connects the node "NURSE"to their corresponding staff profile "STAFF"node. It means that a nurse is a staff member.
- **RESPONSIBLE_FOR**: the relationship indicates that a hospitalization from the node "HOSPITALIZATION"is overseen by a specific nurse from a node "NURSE". It means that a nurse is responsible for a hospitalization.
- **ASSIGNED_TO**: the relationship assigns a hospitalization from a node "HOSPITALIZATION"to a specific room from a node "ROOM". It means that a hospitalization is assigned to a room.
- **INVOLVES**: this relationship indicates that a hospitalization from the node "HOSPITALIZATION"involves a specific episode from an "EPISODE"node. It means that a hospitalization involves an episode.
- **BIILED_FOR**: this relationship shows that a bill from a node "BILL"is related to a specific episode from an "EPISODE"node. It means that a bill is billed for an episode.
- **PRESCRIBED_FOR**: this relationship connects a prescription from the node "PRESCRIPTION"to a node "EPISODE"it was prescribed for. It means that a prescription is prescribed for an episode.
- **PRESCRIBED_MEDICINE**: this relationship shows that a prescription node "PRESCRIPTION"includes a specific medicine from a node "MEDICINE". It means that a prescription involves a specific medicine.
- **HAS_DOCTOR**: this relationship connects an appointment node "APPOINTMENT"to the doctor node "DOCTOR"assigned to it. It means that an appointment has a doctor.
- **BELONGS_TO_EPISODE**: this relationship shows that an appointment node "APPOINTMENT"is part of a specific episode node "EPISODE". It means that an appointment belongs to an episode.
- **BELONGS_TO**: this relationship indicates that a lab screening node "LAB_SCREENING"is associated with a specific episode node "EPISODE". It means that a lab screening belongs to an episode.
- **PERFORMED_BY**: this relationship connects a lab screening node "LAB_SCREENING"to the technician node "TECHNICIAN"who performed it. It means that a lab screening is performed by a technician.
- **HAS_EPISODE**: this relationship indicates that a patient node "PATIENT"has a specific episode node "EPISODE". It means that a patient has an episode.
- **CONTACT_FOR**: this relationship connects a patient node "PATIENT"to their emergency contact node "EMERGENCY_CONTACT". It means that a patient has an emergency contact.
- **HAS_INSURANCE**: this relationship indicates that a patient node "PATIENT"has a specific insurance policy node "INSURANCE". It means that a patient has insurance.

- **HAS_MEDICAL_HISTORY**: this relationship connects a patient node ”PATIENT”to their medical history records node ”MEDICAL_HISTORY”. It means that a patient has medical history.

3.3 Triggers

Since triggers are not part of neo4j by default we had to enable them by using the *APOC* (Awesome Procedures on Cypher) library. We have defined the following triggers:

- **Trigger for Logging New Patient Creation**

```
CALL apoc.trigger.add('logPatientCreation',
     "UNWIND $createdNodes AS n
      WITH n
      WHERE n:Patient
      CREATE (log:Log {message: 'New patient created', patientId: n.IDPATIENT,
       timestamp: timestamp()})",
     {phase: 'after'}
)
```

- **Trigger for Ensuring Medical History Relationship**

```
CALL apoc.trigger.add('createMedicalHistoryRelationship',
     "UNWIND $createdNodes AS mh
      WITH mh
      WHERE mh:Medical_History
      MATCH (p:Patient {IDPATIENT: mh.IDPATIENT})
      CREATE (p)-[:HAS_MEDICAL_HISTORY]->(mh)",
     {phase: 'after'}
)
```

4 Queries

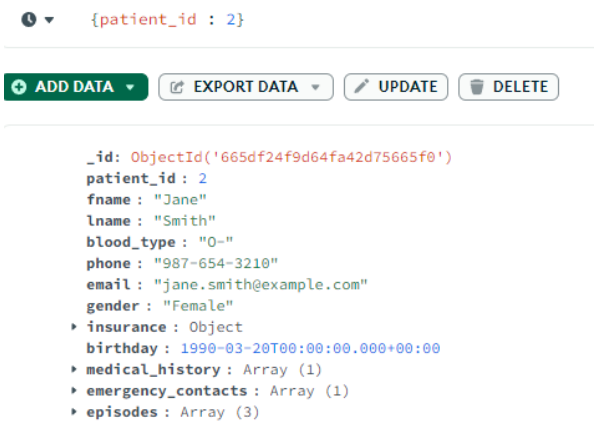
In this section we will be presenting the queries we have developed to demonstrate the functionality of the system we have implemented. We will show the queries both in **MongoDB** and in **neo4j**.

- **Get patient by its id**
  In neo4j:

```
MATCH(p:Patient {IDPATIENT: 1})
RETURN p
```

  In Mongo:

```
"name" : "Get Patient by ID",
"command" : "find({ patient_id: 1 })"
```



- **Get all episodes for a specific patient**
  In neo4j:

```
MATCH(p:Patient {IDPATIENT: 1})-[:HAS_EPISODE]->(e:Episode)
RETURN e
```

  In Mongo:

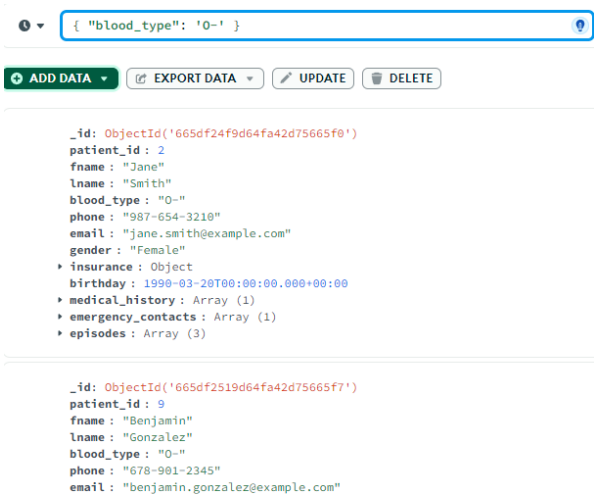
```
"name" : "Get all Episodes for a specific Patient",
"command" : "aggregate([ { $match: { patient_id: 1 } }, { $unwind: \"$episodes\"
    }])"
```

- **Get all patients with specific blood type**
  In neo4j:

```
MATCH(p:Patient {BLOOD_TYPE: 'O-'})
RETURN p
```

  In Mongo:

```
"name" : "Get all patients with specific blood type",
"command" : "find({ blood_type: 'O-' })"
```



- **Find the average age of all Patients**
  In neo4j:

```
MATCH (p:Patient)
RETURN avg(date().year -p.BIRTHDAY.year) AS averageAge
```

  In Mongo:

```
"name" : "Find the average age of all Patients",
"command" : "aggregate([{$group: {_id: null,averageAge: {$avg:
   {$divide: [{ $subtract: [new Date(), \"$birthday\"] },
   1000 * 60 * 60 * 24 * 365]}}}}])"
```

- **Get patient phone number and update it**
  In neo4j:

```
MATCH (p:Patient {IDPATIENT:1})
return p.PHONE

MATCH (p:Patient {IDPATIENT:1})
SET p.PHONE = '555-6789'
RETURN p
```

  In Mongo:

```
"name" : "Get patient phone number and update it",
"command" : "updateOne({ patient_id: 1 },{ $set: { phone: '555-6789' } })"
```

- **See patients medical history and new condition to medical history**
  In neo4j:

```
MATCH (p:Patient {IDPATIENT: 1})
CREATE (mh:Medical_History {
    RECORD_ID: 47,
    CONDITION: 'Back Pain',
    RECORD_DATE: date('2024-05-20'),
    IDPATIENT: 1
})
CREATE (p)-[:HAS_MEDICAL_HISTORY]->(mh)
RETURN p, mh
```

  In Mongo:

```
"name" : "See patients medical history and add new condition to medical history",
"command" : "updateOne: {{ patient_id: 1 }, $push: {
    medical_history: { medical_history_id: 1098, condition: \"Schizophrenia\",
        record_date: new Date() } }, }"
```

- **Delete a patient and all their episodes**
  In neo4j:

```
MATCH (p:Patient {IDPATIENT: 2}) - [:HAS_EPISODE] -> (e:Episode)
DETCH DELETE p, e
```

  In Mongo:

```
"name" : "Delete a Patient and all their Episodes",
"command" : "deleteOne({ patient_id: 2 })"
```

- **Remove a specific episode for a patient**
  In neo4j:

```
MATCH (p:Patient {IDPATIENT: 1}) - [:HAS_EPISODE] -> (e: Episode {IDEPISODE:
    180})
DETACH DELETE e
```

  In Mongo:

```
"name" : "Remove a specific episode for a patient",
"command" : "updateOne({ patient_id: 1 },{ $pull: { episodes: { episode_id: 180 }
     } })"
```

- **Find patients who have been prescribed a specific medicine**
  In neo4j:

```
MATCH (p:Patient)-[:HAS_EPISODE]->(e:Episode)<-[:PRESCRIBED_FOR]-(pr:Prescription
    )-[:PRESCRIBED_MEDICINE]->(m:Medicine {M_NAME: 'Paracetamol'})
RETURN p
```

  In Mongo:

```
"name" : "Find Patients who have been prescribed a specific medicine",
"command" : "aggregate([{ $unwind: \"$episodes\" },{ $unwind:
   \"$episodes.prescriptions\" },{ $match: {
   \"episodes.prescriptions.medicine.medicine_name\": \"Paracetamol\" }}])"
```

- **Get the total cost of all bills for a specific patient**
  In neo4j:

```
MATCH (p:Patient {IDPATIENT: 3})-[:HAS_EPISODE]->(e:Episode)<-[:BILLED_FOR]-(b:
    Bill)
RETURN sum(b.TOTAL) AS totalCost
```

  In Mongo:

```
"name" : "Get the total cost of all bills for a specific Patient",
"command" : "aggregate([{ $match: { patient_id: 3 } },{ $unwind: \"$episodes\" },
   { $unwind: \"$episodes.bills\" },{$group: {_id: null,totalCost:
      { $sum: \"$episodes.bills.total\" }}}])"
```

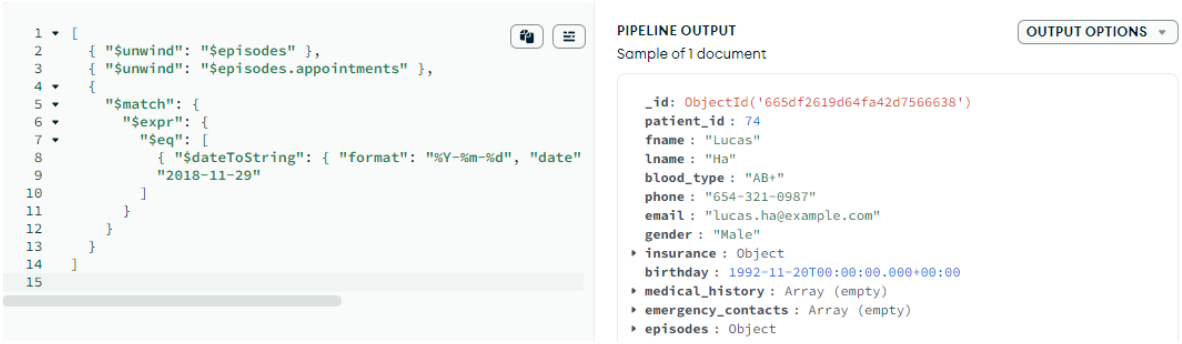
- **Find all patients who have an appointment on a specific date**
  In neo4j:

```
MATCH (:Episode)-[r]-(:Appointment)
RETURN type(r)

MATCH (p:Patient)-[:HAS_EPISODE]->(e:Episode)<-[:BELONGS_TO_EPISODE]-(a:
    Appointment)
WHERE a.APPOINTMENT_DATE = datetime("2018-11-29T00:00:00Z")
RETURN p
```

  In Mongo:

```
"name" : "Find all patients who have an appointment on a specific date",
"command" : "{ $unwind: $episodes },
{ $unwind: $episodes.appointments },
{ $match: { $expr: { $eq: [ { $dateToString: { format: "%Y-%m-%d", date:
    $episodes.appointments.date }}, "2018-11-29" ]}}}"
```

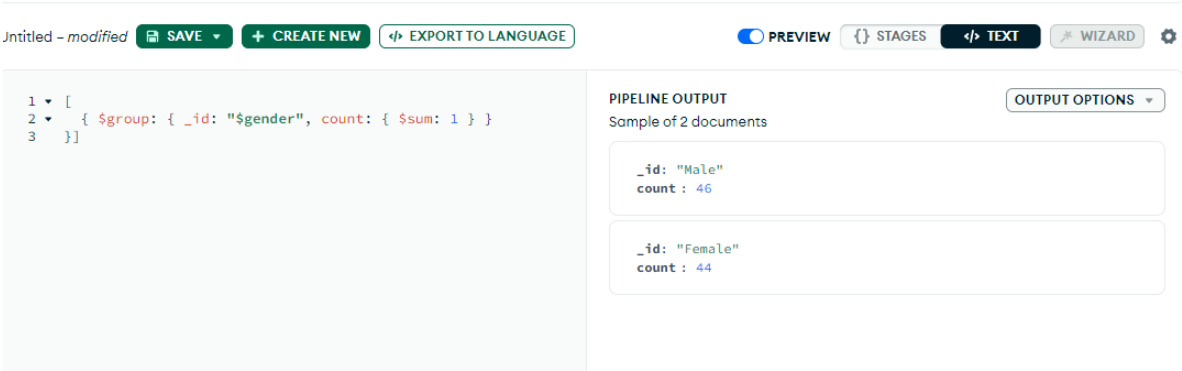


- **Count the number of patients by gender**
  In neo4j:

```
MATCH (p:Patient)
RETURN p.GENDER, count(*) AS count
```

  In Mongo:

```
"name" : "Count the number of patients by gender",
"command" : "{ $group: { _id: $gender, count: { $sum: 1 }}}"
```



- **Find the most common medical condition among patients**
  In neo4j:

```
MATCH (mh:Medical_History)
RETURN mh.CONDITION, count(*) AS frequency
ORDER BY frequency DESC
LIMIT 1
```

  In Mongo:

```
"name" : "Find the most common medical condition among Patients",
"command" : "aggregate([{ $unwind: \"$medical_history\" },{$group: {_id: \"
    $medical_history.condition\",frequency: { $sum: 1 }}},{ $sort: { frequency:
    -1 } },{ $limit: 1 }]))"
```

```
[
  { "$unwind": "$medical_history" },
  { "$group": { "_id": "$medical_history.condition", "freque
  { "$sort": { "frequency": -1, "_id":1 } },
  { "$limit": 1 }
]
```

PIPELINE OUTPUT

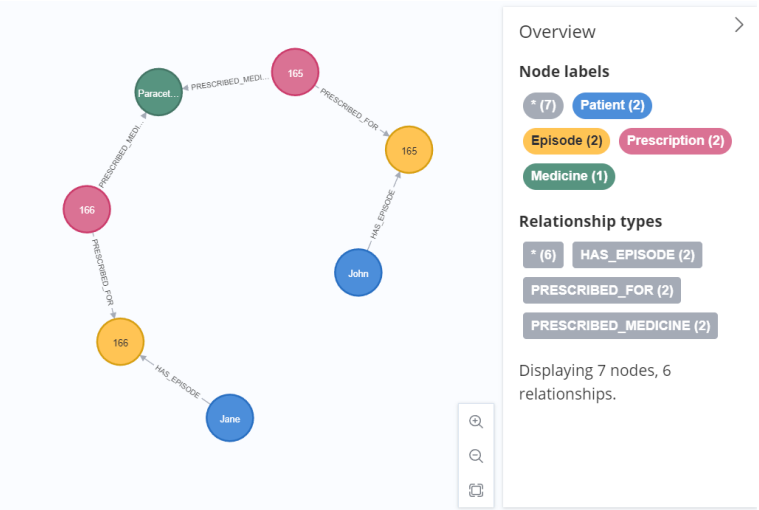
OUTPUT OPTIONS

Sample of 1 document

```
_id: "Anxiety"
frequency : 2
```

- **Find the shortest path between two patients through their common medical history conditions**
  In neo4j:

```
MATCH (p1:Patient {IDPATIENT: 1}), (p2:Patient {IDPATIENT: 2}),
   path = shortestPath((p1)-[*]-(p2))
RETURN path
```



5 Conclusion

Finally, this assignment helped us expand our understanding gained during our classes and achieve further insight related to NoSQL databases. We were able to evaluate how NoSQL databases offer increased flexibility and scalability.

Through hands-on experience with MongoDB and Neo4j, we explored the strengths and appropriate use cases of document-oriented and graph database models. This practical application allowed us to contrast these models with traditional relational databases, understanding their advantages in handling unstructured data and complex relationships.

We believe that we have met the goals of this practical assignment by successfully implementing and integrating multiple database paradigms to manage diverse data types effectively. This has not only reinforced our theoretical knowledge but also provided valuable practical skills that are highly relevant in today's data-driven world.

Overall, the assignment has prepared us to make informed decisions about database design and implementation in various real-world scenarios, contributing to our readiness for professional challenges in the field of database management.

6 Attachments

6.1 Sample Patient document


```
[{
  "patient_id": 0,
  "fname": "Jane",
  "lname": "Smith",
  "blood_type": "O-",
  "phone": "987-654-3210",
  "email": "jane.smith@example.com",
  "gender": "Female",
  "insurance": {
    "policy_number": "POL002",
    "provider": "XYZ Insurance",
    "insurance_type": "Premium Plan",
    "co_pay": 30,
    "coverage": "Partial Coverage",
    "maternity": "N",
    "dental": "Y",
    "vision": "Y"
  },
  "birthday": {
    "$date": "1990-03-20T00:00:00.000Z"
  },
  "medical_history": [
    {
      "medical_history_id": 2,
      "patient_id": "Allergy",
      "condition": {
        "$date": "2023-03-05T00:00:00.000Z"
      },
      "record_date": 2
    }
  ],
  "emergency_contacts": [
    {
      "name": "Jane Smith",
      "phone": "222-333-4444",
      "relation": "Mother"
    }
  ],
  "episodes": [
    {
      "episode_id": 2,
      "patient_id": 2,
      "prescriptions": [
        {
```

```
    "prescription_id": 384,
    "prescription_date": {
      "$date": "2019-04-18T00:00:00.000Z"
    },
    "medicine": {
      "medicine_id": 1,
      "medicine_name": "Paracetamol",
      "medicine_quantity": 50,
      "medicine_cost": 10
    },
    "dosage": 81
  }
],
"bills": [
  {
    "bill_id": 0,
    "room_cost": 0,
    "test_cost": 0,
    "add_charges": 0,
    "total_cost": 0,
    "register_date": {
      "$date": ""
    },
    "payment_status": "PENDING"
  },
],
"screenings": [
  {
    "screening_id": 88,
    "screening_date": {
      "$date": "2019-04-18T00:00:00.000Z"
    },
    "screening_technician": {
      "employee_id": 49,
      "fname": "Ashley",
      "lname": "Stein",
      "dateJoining": {
        "$date": "2014-10-02T00:00:00.000Z"
      },
      "dateSeparation": null,
      "email": "krystal59@example.org",
      "address": "\"8049 Adrian Throughway",
      "department": {
        "department_id": 9,
        "dept_head": "Travis Smith",
        "dept_name": "Diagnostic_2",
        "dept_employeeCount": 3
      },
      "ssn": 414092098,
      "is_active": "Y"
    },
    "screening_cost": 198.34
  }
],
"appointments": [],
"hospitalizations": [
  {
    "admissionDate": {
      "$date": "2019-04-17T00:00:00.000Z"
    },
```

```
        "dischargeDate": null,
        "room": {
          "room_id": 1,
          "room_type": "Single",
          "room_cost": 100
        },
        "responsibleNurse": {
          "employee_id": 4,
          "fname": "Joe",
          "lname": "Ferguson",
          "dateJoining": {
            "$date": "2018-08-10T00:00:00.000Z"
          },
          "dateSeparation": null,
          "email": "vsullivan@example.org",
          "address": "DPO AE 27029\"",
          "department": {
            "department_id": 17,
            "dept_head": "Ethan Brown",
            "dept_name": "Gynecology",
            "dept_employeeCount": 5
          },
          "ssn": 531094042,
          "is_active": "Y"
        }
      }
    ]
  }
 ]
}]
```


6.2 Migration Code

```
    #Lets get all the patients
    cursor = self.OracleConnection.cursor()
    cursor.execute("SELECT * FROM PATIENT")
    patients = cursor.fetchall()
    for patient in tqdm(patients, desc="Migrating patients", unit="patient"):

        ...

        #Lets get the medical history of the patient
        cursor.execute(f"SELECT * FROM MEDICAL_HISTORY WHERE IDPATIENT = {patientID}")
        medical_histories = cursor.fetchall()
        medical_histories_list = []
        for medical_history in medical_histories:
                ...
               medical_histories_list.append(MedicalHistory(

        #Lets get the insurance of the patient
        cursor.execute(f"SELECT * FROM INSURANCE WHERE policy_number='{patientPolicyNumber}'")
        insurance = cursor.fetchone()

        patientInsurance = Insurance(insurancePolNum, insuranceProvider, insuranceType,
            insuranceCoPay, insuranceCoverage, insuranceMaternity, insuranceDental,
            insuranceVision)
```

```
#Lets get the emergency contacts of the patient
cursor.execute(f"SELECT * FROM EMERGENCY_CONTACT WHERE IDPATIENT = {patientID}")
emergency_contacts = cursor.fetchall()
emergency_contacts_list = []
for emergency_contact in emergency_contacts:
    ...
    emergency_contacts_list.append(EmergencyContact(emergencyContactName,
        emergencyContactPhone, emergencyContactRelation))


#Lets get the patients episodes
cursor.execute(f"SELECT * FROM EPISODE WHERE PATIENT_IDPATIENT = {patientID}")
episodes = cursor.fetchall()
episodesList = []

for episode in episodes:
    episodeId = episode[0]
    #Lets get the prescriptions of the episode
    cursor.execute(f"SELECT * FROM PRESCRIPTION WHERE IDEPISODE = {episodeId}")
    prescriptions = cursor.fetchall()
    prescriptions_list = []
    for prescription in prescriptions:

        ...
        prescriptions_list.append(Prescription(prescriptionId, prescriptionDate,
            prescriptionMedicine, prescriptionDosage))

    #Lets get the bills of the episode
    cursor.execute(f"SELECT * FROM BILL WHERE IDEPISODE = {episodeId}")
    bills = cursor.fetchall()
    bills_list = []
    for bill in bills:
        ...
        bills_list.append(Bill(billId, billRoomCost, billTestCost, billAddCharges,
            billTotalCost, billRegisterDate, billPaymentStatus))

    #Lets get the episodes screenings
    cursor.execute(f"SELECT * FROM LAB_SCREENING WHERE EPISODE_IDEPISODE = {episodeId}"
        )
    screenings = cursor.fetchall()
    screenings_list = []
    for screening in screenings:

        ...

        #Lets get the technician data
        cursor.execute(f"SELECT STAFF_EMP_ID FROM TECHNICIAN WHERE STAFF_EMP_ID = {
            screeningTechnicianId}")
        technician = cursor.fetchone()
        screeningTechnicianId = technician[0]

        #Lets get the technician data by querying the staff table
        cursor.execute(f"SELECT * FROM STAFF WHERE EMP_ID = {screeningTechnicianId}")
        technician = cursor.fetchone()
        technicianDeptId = technician[8]

        #Lets get the department data
        cursor.execute(f"SELECT * FROM DEPARTMENT WHERE IDDEPARTMENT = {technicianDeptId
```

```
        }")
    technicianDept = cursor.fetchone()
    technicianDept = Department(technicianDept[0], technicianDept[1], technicianDept
        [2], technicianDept[3])

    technician = Employee(technician[0], technician[1], technician[2], technician
        [3], technician[4], technician[5], technician[6], technician[7],
        technicianDept, technician[9])

    screen = LabScreening(screeningId, screeningCost, screeningDate, technician)

    screenings_list.append(screen)

#Lets get the episodes appointment
cursor.execute(f"SELECT * FROM APPOINTMENT WHERE IDEPISODE = {episodeId}")
appointments = cursor.fetchall()
appointments_list = []
for appointment in appointments:

    ...
    #Lets get the doctor data
    cursor.execute(f"SELECT * FROM DOCTOR WHERE emp_id = {appointmentDoctorId}")
    doctor = cursor.fetchone()
    qualifications = doctor[1]

    #Lets get the doctor data by querying the staff table
    cursor.execute(f"SELECT * FROM STAFF WHERE EMP_ID = {appointmentDoctorId}")
    doctor = cursor.fetchone()
    doctorDeptId = doctor[8]

    #Lets get the department data
    cursor.execute(f"SELECT * FROM DEPARTMENT WHERE IDDEPARTMENT = {doctorDeptId}")
    doctorDept = cursor.fetchone()
    doctorDept = Department(doctorDept[0], doctorDept[1], doctorDept[2], doctorDept
        [3])


    doctor = Doctor(Employee(doctor[0], doctor[1], doctor[2], doctor[3], doctor[4],
        doctor[5], doctor[6], doctor[7], doctorDept, doctor[9]), qualifications)
    #scheduled_on, appointment_date, appointment_time, doctor
    appointments_list.append(Appointment(scheduledOn, appointmentDate,
        appointmentTime, doctor))

#Lets get all the hospitalizations of the episode
cursor.execute(f"SELECT * FROM HOSPITALIZATION WHERE IDEPISODE = {episodeId}")
hospitalizations = cursor.fetchall()
hospitalizations_list = []
for hospitalization in hospitalizations:
    ...


    #Lets get the nurse data by querying the staff table
    cursor.execute(f"SELECT * FROM STAFF WHERE EMP_ID = {
        hospitalizationResponsibleNurseId}")
    nurse = cursor.fetchone()

    #Lets get the department data
    cursor.execute(f"SELECT * FROM DEPARTMENT WHERE IDDEPARTMENT = {nurse[8]}")
    nurseDept = cursor.fetchone()
    nurseDept = Department(nurseDept[0], nurseDept[1], nurseDept[2], nurseDept[3])
```

```
            nurse = Employee(nurse[0], nurse[1], nurse[2], nurse[3], nurse[4], nurse[5],
                nurse[6], nurse[7], nurseDept, nurse[9])

            #Lets get the room data
            cursor.execute(f"SELECT * FROM ROOM WHERE IDROOM = {hospitalizationRoomId}")
            room = cursor.fetchone()
            room = Room(room[0], room[1], room[2])

            hospitalizations_list.append(Hospitalization(hospitalizationAdmissionDate,
                hospitalizationDischargeDate, room, nurse))


        episodesList.append(Episode(episodeId, patientID, prescriptions_list, bills_list,
            screenings_list, appointments_list, hospitalizations_list))

    #Lets convert the Patient to a json object and insert it into the MongoDB
    patient = Patient(patientID, patientFname, patientLname, patientBloodType, patientPhone
        , patientEmail, patientGender, patientBirthday, medical_histories_list,
        patientInsurance, emergency_contacts_list, episodesList)
    self.mongoDB["Patient"].insert_one(patient.to_json())
```

Listing 6.1: Portion of the migration process applied to each patient

6.3 Patient document before the update and trigger execution

```
[{
  "_id": {
    "$oid": "665db9f60e524a17a0115511"
  },
  "patient_id": 2,
  ...
  "episodes": [
    {
      "episode_id": 181,
      "patient_id": 2,
      "prescriptions": [],
      "bills": [
        ...
      ],
      "screenings": [
        ...
      ],
      "appointments": [],
      "hospitalizations": [
        {
          "admissionDate": {
            "$date": "2019-04-17T00:00:00.000Z"
          },
          "dischargeDate": null,
          "room": {
            "room_id": 1,
            "room_type": "Single",
            "room_cost": 100
          },
          "responsibleNurse": {
            "employee_id": 4,
            "fname": "Joe",
            "lname": "Ferguson",
            "dateJoining": {
```

```
            "$date": "2018-08-10T00:00:00.000Z"
          },
          "dateSeparation": null,
          "email": "vsullivan@example.org",
          "address": "DPO AE 27029\"",
          "department": {
            "department_id": 17,
            "dept_head": "Ethan Brown",
            "dept_name": "Gynecology",
            "dept_employeeCount": 5
          },
          "ssn": 531094042,
          "is_active": "Y"
        }
      }
    ]
   }
   ...
  ]
}]
```


6.4 Patient document after the update and trigger execution


```
[{
  "_id": {
    "$oid": "665db9f60e524a17a0115511"
  },
  "patient_id": 2,
  ...
  "episodes": [
    {
      "episode_id": 181,
      "patient_id": 2,
      "prescriptions": [],
      "bills": [
        {
          "bill_id": 48,
          "room_cost": 300,
          "test_cost": 0,
          "add_charges": 3780,
          "total_cost": 4080,
          "register_date": {
            "$date": "2024-06-03T12:41:25.459Z"
          },
          "payment_status": "PENDING"
        }
      ],
      "screenings": [
        ...
      ],
      "appointments": [],
      "hospitalizations": [
        {
          "admissionDate": {
            "$date": "2019-04-17T00:00:00.000Z"
          },
          "dischargeDate": "2024-06-03T12:41:25.342Z",
          "room": {
```

```
          "room_id": 1,
          "room_type": "Single",
          "room_cost": 100
        },
        "responsibleNurse": {
          "employee_id": 4,
          "fname": "Joe",
          "lname": "Ferguson",
          "dateJoining": {
            "$date": "2018-08-10T00:00:00.000Z"
          },
          "dateSeparation": null,
          "email": "vsullivan@example.org",
          "address": "DPO AE 27029\"",
          "department": {
            "department_id": 17,
            "dept_head": "Ethan Brown",
            "dept_name": "Gynecology",
            "dept_employeeCount": 5
          },
          "ssn": 531094042,
          "is_active": "Y"
        }
      }
    ]
  }
  ...
 ]
}]
```