
# Bases de Dados NoSQL
# **Relatório do Trabalho Prático**

Pedro Pereira Sousa PG54721
Nuno Miguel Leite da Costa PG54121
Millena de Freitas Santos PG54107
Ricardo Alves Oliveira PG54177

Bases de Dados NoSQL 2023/2024
Universidade do Minho
June 18, 2024

# Contents

# List of Figures

# List of Tables

# 1 Introdução

No âmbito da unidade curricular de **Base de Dados NoSQL**, foi-nos proposta a conceção e implementação de três SGBD: um relacional e dois não relacionais. Para tal, utilizámos a base de dados relacional de um Sistema de Gestão Hospitalar, cujo script foi disponibilizado pelos docentes com a designação hospital.sql.

No que toca à base de dados não relacional orientada a documentos utilizada, o grupo optou pela utilização do MongoDB. Já a base de dados não relacional orientada a grafos escolhida foi o Neo4j. Para ambas as bases de dados foram realizadas as devidas migrações, criadas as views, procedures e triggers originalmente disponiveis para o sistema relacional.

Adicionalmente, o grupo desenvolveu também diferentes queries com diferentes níveis de complexidade, de modo a avaliar a performance em cada um destes três sistemas. As queries foram desenhadas para abranger uma variedade de operações comuns em sistemas de gestão de bases de dados, desde consultas simples a agregações e operações mais complexas que envolvem múltiplas tabelas/coleções/nodos e relacionamentos.

# 2 Base de Dados Orientada a Documentos - MongoDB

No processo de migração da base de dados relacional para um sistema orientado a documetos, o grupo optou pela utilização do MongoDB. Esta escolha baseia-se na sua performance, familiaridade de utilização (assemelhando-se a documentos JSON), escabilidade e *schema* flexivel.

O processo de migração inclui a criação de um novo *schema* que se adapte à estrutura orientada a documentos, usufruindo das suas vantagens em relação ao esquema relacional. De seguida o grupo decidiu, para além da migração dos dados, criar as respetivas *views*, *procedures* e *triggers*.

## 2.1 Schema

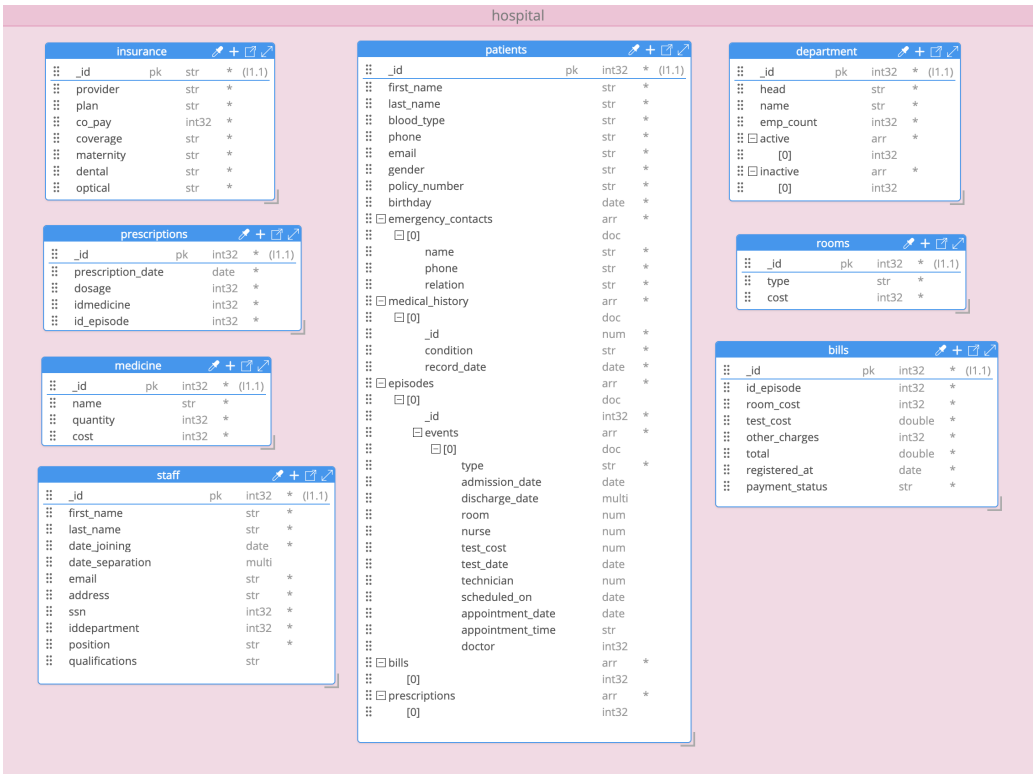


Figure 1: Esquema da arquitetura do MongoDB

O novo schema representado na Figura 1 para a base de dados MongoDB foi cuidadosamente desenhado tendo em conta a estrutura orientada a documentos do MongoDB. Ao contrário das bases de dados relacionais que utilizam tabelas e colunas para armazenar os dados, o MongoDB utiliza coleções de documentos semelhantes a JSON, permitindo uma flexibilidade muito maior na modelação dos dados. Assim, cada coleção foi criada de forma a representar as entidades mais relevantes do sistema de saúde, como pacientes, médicos, departamentos, entre outros.

Esta flexibilidade é particularmente útil num contexto de sistemas de saúde onde diferentes pacientes podem ter diferentes tipos de informações médicas. Para além disso a estrutura dos dados pode evoluir com ao longo do tempo sem a necessidade de migrações complexas. Por sua vez, a estrutura orientada a documentos permite a inclusão de estruturas "aninhadas"(*nested*) como arrays e subdocumentos, o que facilita a representação de dados hierárquicos e relações embutidas nos dados, como o histórico de consultas de um dado paciente ou os detalhes de uma prescrição médica.

## 2.2 Views

```
mongo_db = mongo_client['hospital']
pipeline = [
    {'$unwind': '$episodes'},
    {'$unwind': '$episodes.events'},
    {
        '$match': {
            'episodes.events.type': 'appointment'
        }
    },
    {
        '$lookup': {
            'from': 'staff',
            'localField': 'episodes.events.doctor',
            'foreignField': '_id',
            'as': 'doctor'
        }
    },
    {
        '$unwind': '$doctor'
    },
    {
        '$lookup': {
            'from': 'department',
            'localField': 'doctor.iddepartment',
            'foreignField': '_id',
            'as': 'department'
        }
    },
    {
        '$unwind': '$department'
    },
    {
        '$project': {
            'appointment_scheduled_date': '$episodes.events.scheduled_on',
            'appointment_date': '$episodes.events.appointment_date',
            'appointment_time': '$episodes.events.appointment_time',
            'doctor_id': '$doctor._id',
            'doctor_qualifications': '$doctor.qualifications',
            'department_name': '$department.name',
            'patient_first_name': '$first_name',
            'patient_last_name': '$last_name',
            'patient_blood_type': '$blood_type',
            'patient_phone': '$phone',
            'patient_email': '$email',
            'patient_gender': '$gender'
        }
    }
]
view_data = mongo_db.patients.aggregate(pipeline)
```

Listing 1: Implementação em MongoDB da View

Para replicar a funcionalidade das views SQL no MongoDB, foi utilizada uma pipeline de agregação. Esta pipeline permite realizar operações complexas de consulta e transformação de dados, integrando informações de várias coleções numa única vista. A View implementada tem como objetivo replicar a original, fornecendo uma perspetiva abrangente das marcações dos pacientes.

A pipeline começa por juntar a coleção de pacientes (patients) com a sub-coleção de episódios médicos (episodes) que estão dentro da coleção de pacientes. Dentro de cada episódio, seleciona-se os eventos (events) que são do tipo "appointment" (marcação). Em seguida, junta-se a coleção de staff para obter os detalhes dos médicos responsáveis pelas marcações, pois os médicos estão armazenados como membros do staff. Depois, associa-se a coleção de departamentos (departments) para identificar o departamento relevante para cada marcação.
O resultado final é uma visão detalhada que inclui informações sobre a data e hora da marcação, o médico responsável, o departamento, e dados demográficos e de contato do paciente. Esta visão detalhada facilita a análise e consulta dos dados de marcações, proporcionando uma visão completa e coesa das interações dos pacientes com o sistema de saúde.

### 2.3 Procedures

```
mongo_db = mongo_client['hospital']
bills_collection = mongo_db['bills']

def sp_update_bill_status(p_bill_id, p_paid_value):
    try:
        bill = bills_collection.find_one({'_id': p_bill_id})
        if not bill:
            raise ValueError("Bill not found")

        v_total = bill['total']

        if p_paid_value < v_total:
            # Update status to FAILURE
            bills_collection.find_one_and_update(
                {'_id': p_bill_id},
                {'$set': {'payment_status': 'FAILURE'}},
                return_document=ReturnDocument.AFTER
            )
            raise ValueError("Paid value is inferior to the total value of the bill.")
        else:
            # Update status to PROCESSED
            updated_bill = bills_collection.find_one_and_update(
                {'_id': p_bill_id},
                {'$set': {'payment_status': 'PROCESSED'}},
                return_document=ReturnDocument.AFTER
            )
            return updated_bill

    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == '__main__':
    bill_id = int(input("Enter the bill ID: "))
    paid_value = float(input("Enter the paid value: "))
    updated_bill = sp_update_bill_status(bill_id, paid_value)
    print(updated_bill)
```

Listing 2: Implementação em MongoDB do Procedure

A implementação de procedures em MongoDB é feita através de um script em Python que interage com a base de dados. No nosso caso, foi implementada a procedure *update_bill_status*, que atualiza o estado de uma fatura com base no valor pago.
O script primeiro obtém a fatura correspondente ao ID fornecido. De seguida, ele verifica se o valor pago é inferior ao valor total da fatura. Se este for o caso, o estado da fatura é atualizado para

'FAILURE', já que o pagamento indicado foi insuficiente. Se o valor pago for igual ou superior ao valor total, o estado é atualizado para 'PROCESSED', indicando que o pagamento foi bem-sucedido. Esta abordagem garante que a integridade dos dados financeiros é mantida ao longo do tempo, evitando erros manuais na atualização do estado das faturas. Para além disso, a implementação em Python permite uma maior flexibilidade e capacidade de integração com outros sistemas e processos automáticos.

## 2.4 Triggers

```
mongo_db = mongo_client['hospital']

# Function to calculate the bill and insert it into the bills collection
def generate_bill(change):
    if 'updateDescription' in change and 'updatedFields' in change['updateDescription']:
        updated_fields = change['updateDescription']['updatedFields']
        if 'episodes' in updated_fields:
            # Get the updated patient document
            patient = mongo_db.patients.find_one({'_id': change['documentKey']['_id']})
            if patient:
                for episode in patient.get('episodes', []):
                    for event in episode.get('events', []):
                        if 'discharge_date' in event:
                            episode_id = episode['_id']
                            room_id = event.get('room')

                            # Calculate room cost
                            room = mongo_db.rooms.find_one({'_id': room_id})
                            room_cost = room['cost'] if room else 0

                            # Calculate test cost
                            test_cost = mongo_db.lab_screening.aggregate([
                                {'$match': {'episode_idepisode': episode_id}},
                                {'$group': {'_id': None, 'total_test_cost': {'$sum':
                                    '$test_cost'}}}
                            ])
                            test_cost = list(test_cost)[0]['total_test_cost'] if test_cost
                                else 0

                            # Calculate other charges
                            other_charges = mongo_db.prescriptions.aggregate([
                                {'$match': {'id_episode': episode_id}},
                                {'$lookup': {
                                    'from': 'medicine',
                                    'localField': 'idmedicine',
                                    'foreignField': '_id',
                                    'as': 'medicine'
                                }},
                                {'$unwind': '$medicine'},
                                {'$group': {'_id': None, 'total_other_charges': {'$sum':
                                    {'$multiply': ['$dosage', '$medicine.cost']}}}}
                            ])
                            other_charges = list(other_charges)[0]['total_other_charges'] if
                                other_charges else 0

                            # Calculate total cost
                            total_cost = room_cost + test_cost + other_charges

                            # Insert bill
                            bill = {
```

```
                            'id_episode': episode_id,
                            'room_cost': room_cost,
                            'test_cost': test_cost,
                            'other_charges': other_charges,
                            'total': total_cost,
                            'payment_status': 'PENDING',
                            'registered_at': datetime.now()
                        }
                        mongo_db.bills.insert_one(bill)
                        print(f"Bill generated for episode {episode_id}")

# Set up change stream listener
with mongo_db.patients.watch([{'$match': {'operationType': 'update'}}]) as stream:
    for change in stream:
        generate_bill(change)
```

Listing 3: Implementação em MongoDB do Trigger

O trigger *generate_bill* foi implementado num script Python que utiliza a funcionalidade de *change streams* do MongoDB para monitorizar as atualizações das hospitalizações (*hospitalizations*).
Quando uma atualização é detetada, este script verifica se a data de saída (*discharge_date*) do paciente foi modificada. Caso isso se verifique o trigger é acionado para calcular os custos associados ao episódio de hospitalização. Este custo deve incluir o custo do quarto, o custo dos exames laboratoriais, etc. Estes custos são então somados para calcular o custo total e uma nova fatura é inserida na coleção de faturas (*bills*). A nova fatura inclui todos os detalhes relevantes sobre o episódio de hospitalização, os custos individuais e o custo total, bem como o estado inicial de pagamento como 'PENDING'.
Esta implementação de triggers em Python garante que as operações necessárias sejam realizadas de forma automatica, mantendo a consistência e integridade dos dados sem a necessidade de intervenção manual para cada fatura. Além disso, esta estratégia permite uma maior flexibilidade na definição e execução das regras de negócio, adaptando-se facilmente às necessidades específicas do sistema de saúde.

# 3 Base de Dados Orientada a Grafos - Neo4j

Como foi mencionado anteriormente, a base de dados não relacional orientada a grafos escolhida foi o *Neo4j*, uma escolha que se baseia no facto de esta base de dados já ter sido abordada e explorada durante as aulas práticas desta unidade curricular, existindo assim um maior conhecimento na sua utilização.
Desta forma, o grupo começou por desenvolver um script responsável por realizar a migração para o *Neo4j*. Neste script, são estabelecidas tanto as conexões à base de dados relacional como à base de dados não relacional, e são construídos os nodos e as relações com a ajuda de um cursor que obtém os vários registos da base de dados. Dessa forma, as tabelas presentes na base de dados deram origem a nodos (um total de 2089 criados), enquanto que as foreign keys deram origem às relações (um total de 3461 criadas) entre os nodos, sendo que na Figura 3 podemos ver o esquema da arquitetura do Neo4j utilizada.

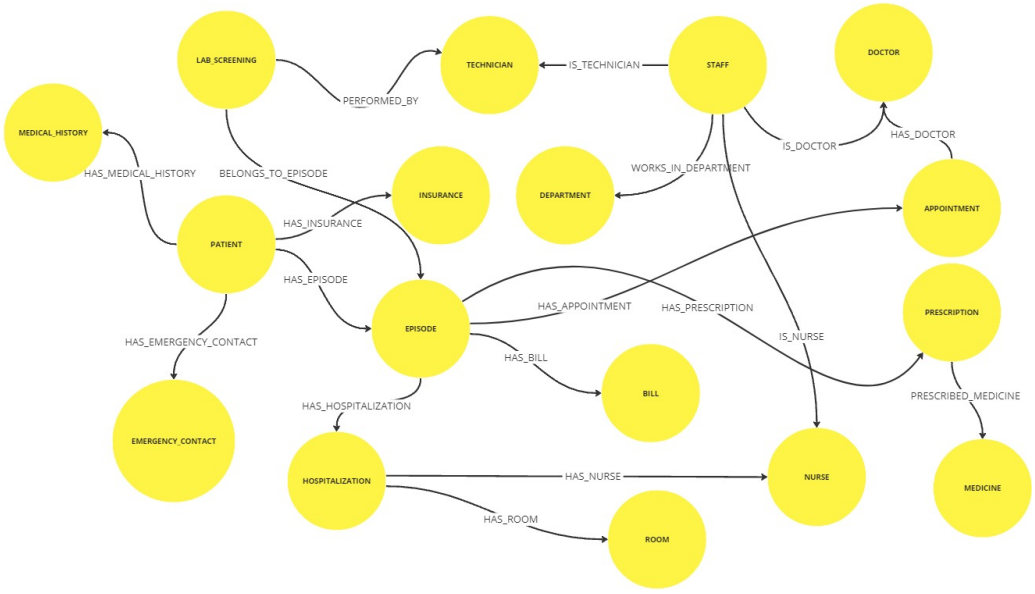


Figure 2: Esquema da arquitetura do Neo4j

## 3.1 Views

Tal como foi feito no MongoDB, também implementamos a funcionalidade das views SQL no Neo4j. Para isso, criamos uma função que executa uma consulta Cypher para integrar informações de várias entidades num único resultado. Inicialmente, executamos uma consulta Cypher que une diferentes nós e relacionamentos no banco de dados Neo4j. A consulta começa por associar as marcações (Appointment) aos médicos (Doctor) através do relacionamento HAS_DOCTOR. Em seguida, associa os médicos aos membros da equipa (Staff) e aos departamentos (Department) onde trabalham, através dos relacionamentos IS_DOCTOR e WORKS_IN_DEPARTMENT.
A consulta também associa os pacientes (Patient) aos episódios médicos (Episode) através do relacionamento HAS_EPISODE. A combinação de todas estas informações permite obter uma visão detalhada que inclui a data e hora da marcação, o médico responsável, o departamento, e dados demográficos e do contacto do paciente. Posteriormente, formatamos e imprimimos os resultados da consulta de forma estruturada, ao utilizar uma tabela para apresentar os dados de maneira clara e organizada, que facilita a análise e consulta das marcações.

## 3.2 Procedures

Em relação às procedures, a implementação destas no Neo4j foi realizada de maneira semelhante à do MongoDB, mas adaptada ao ambiente e à linguagem de consulta específica do Neo4j. No caso do Neo4j, utilizámos a procedure neo_update_bill_status, que atualiza o estado de uma fatura com base no valor pago, tal como foi feito no MongoDB.

# 4 Queries Desenvolvidas

Nesta secção, serão abordadas um conjunto de 13 queries realizadas pelo grupo com o intuito de explorar não só as funcionalidades e implementações de cada uma das três bases de dados, mas também o desempenho de todas, visto que cada uma utiliza uma forma diferente de armazenar os dados. Para assegurar que os resultados das queries têm o mesmo output, o grupo optou por desenvolver um script em Python que corre as queries relativas a cada uma das três bases de dados e, posteriormente, compara os seus resultados, verificando assim se todas as queries apresentam os resultados esperados de forma eficiente e simples.
Nas subseções seguintes, serão apresentadas as implementações das queries para cada uma das bases de dados e os respetivos tempos de execução em milissegundos. É importante realçar dois aspetos sobre os tempos de execução: o primeiro é que o tempo da primeira execução de cada query é descartado, uma vez que apresenta sempre um valor discrepante em relação aos restantes; o segundo é que consideramos o valor final de cada query como a média de 10 medições, utilizando para tal o mesmo script mencionado anteriormente.

## 4.1 Query 1 - Listar por ordem decrescente os medicamentos mais caros

### Oracle

```
SELECT
    idmedicine AS id,
    m_name AS name,
    m_cost AS cost
FROM
    medicine
ORDER BY
    m_cost DESC,
    idmedicine ASC;
```

Listing 4: Implementação em Oracle da query 1

### MongoDB

```
db.medicine.find().sort({ cost: -1 })
```

Listing 5: Implementação em MongoDB da query 1

### Neo4j

```
MATCH (m:Medicine)
RETURN m.id_medicine AS id, m.m_name AS name, m.m_cost AS cost
ORDER BY m.m_cost DESC
```

Listing 6: Implementação em Neo4j da query 1

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 3.20 ms |
| MongoDB | 2.49 ms |
| Neo4j | 8.72 ms |

Table 1: Tempos de execução da query 1

## 4.2 Query 2 - Listar pacientes que têm mais de 3 episódios por ordem decrescente

### Oracle

```
SELECT
    p.idpatient AS id_patient,
    p.patient_fname AS patient_fname,
    p.patient_lname AS patient_lname,
    COUNT(e.idepisode) AS episode_count
FROM
    SYSTEM.patient p
JOIN
    SYSTEM.episode e ON p.idpatient = e.patient_idpatient
GROUP BY
    p.idpatient, p.patient_fname, p.patient_lname
HAVING
    COUNT(e.idepisode) > 3
ORDER BY
    episode_count DESC,
    p.idpatient ASC
```

Listing 7: Implementação em Oracle da query 2

### MongoDB

```
patients_collection = db['patients']
    pipeline = [
        {
            '$project': {
                'first_name': 1,
                'last_name': 1,
                'num_episodes': {'$size': '$episodes'}
            }
        },
        {
            '$match': {
                'num_episodes': {'$gt': 3}
            }
        },
        {
            '$sort': {
                'num_episodes': -1
            }
        }
    ]

    pacientes = patients_collection.aggregate(pipeline)
```

Listing 8: Implementação em MongoDB da query 2

### Neo4j

```
MATCH (p:Patient)-[:HAS_EPISODE]->(e:Episode)
WITH p, COUNT(e) AS episode_count
WHERE episode_count > 3
RETURN p.id_patient AS id_patient, p.patient_fname AS patient_fname, p.patient_lname AS
    patient_lname, episode_count
ORDER BY episode_count DESC, id_patient ASC
```

Listing 9: Implementação em Neo4j da query 2

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 6.30 ms |
| MongoDB | 4.54 ms |
| Neo4j | 9.64 ms |

Table 2: Tempos de execução da query 2

## 4.3 Query 3 - Listar pacientes e os seus contactos de emergência

### Oracle

```
SELECT
    p.idpatient AS id_patient,
    p.patient_fname AS patient_fname,
    p.patient_lname AS patient_lname,
    ec.contact_name AS contact_name,
    ec.phone AS phone
FROM
    SYSTEM.patient p
JOIN
    SYSTEM.emergency_contact ec ON p.idpatient = ec.idpatient
ORDER BY
    id_patient ASC
```

Listing 10: Implementação em Oracle da query 3

### MongoDB

```
  projection = {
      '_id': 1,
      'first_name': 1,
      'last_name': 1,
      'emergency_contacts': 1
  }
  patients = db.patients.find(query, projection)
```

Listing 11: Implementação em MongoDB da query 3

### Neo4j

```
MATCH (p:Patient)-[:HAS_EMERGENCY_CONTACT]->(ec:Emergency_Contact)
RETURN p.id_patient AS id_patient, p.patient_fname AS patient_fname, p.patient_lname AS
    patient_lname, ec.contact_name AS contact_name, ec.phone AS phone
ORDER BY p.id_patient ASC
```

Listing 12: Implementação em Neo4j da query 3

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 3.70 ms |
| MongoDB | 3.16 ms |
| Neo4j | 6.75 ms |

Table 3: Tempos de execução da query 3

## 4.4 Query 4 - Listar as salas com o maior custo de hospitalização total

### Oracle

```
SELECT
    r.idroom AS room_id,
    r.room_type AS room_type,
    SUM(b.total) AS total_cost
FROM
    SYSTEM.room r
LEFT JOIN
    SYSTEM.hospitalization h ON r.idroom = h.room_idroom
LEFT JOIN
    SYSTEM.episode e ON h.idepisode = e.idepisode
LEFT JOIN
    SYSTEM.bill b ON e.idepisode = b.idepisode
GROUP BY
    r.idroom, r.room_type
HAVING
    SUM(b.total) IS NOT NULL --  para remover os quatros que no tiveram custos
ORDER BY
    total_cost DESC
```

Listing 13: Implementação em Oracle da query 4

### MongoDB

```
pipeline = [
       {
           '$unwind': '$episodes'
       },
       {
           '$unwind': '$episodes.events'
       },
       {
           '$match': {
              'episodes.events.type': 'hospitalization'
           }
```

```
        },
        {
            '$lookup': {
                'from': 'bills',
                'localField': 'episodes._id',
                'foreignField': 'id_episode',
                'as': 'hospital_bills'
            }
        },
        {
            '$unwind': {
                'path': '$hospital_bills',
                'preserveNullAndEmptyArrays': True
            }
        },
        {
            '$group': {
                '_id': '$episodes.events.room',
                'total_cost': {'$sum': '$hospital_bills.total'}
            }
        },
        {
        '$match': {
            'total_cost': {'$gt': 0}
        }
        },
        {
            '$sort': {
                'total_cost': -1
            }
        }
    ]

    resultados = db.patients.aggregate(pipeline)
```

Listing 14: Implementação em MongoDB da query 4

### Neo4j

```
MATCH
    (r:Room)<-[:HAS_ROOM]-(h:Hospitalization)<-[:HAS_HOSPITALIZATION]-(e:Episode)-[:HAS_BILL]->
(b:Bill)
RETURN r.id_room AS room_id, r.room_type AS room_type, SUM(b.total) AS total_cost
ORDER BY total_cost DESC
```

Listing 15: Implementação em Neo4j da query 4

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 5.80 ms |
| MongoDB | 57.70 ms |
| Neo4j | 7.51 ms |

Table 4: Tempos de execução da query 4

### 4.5 Query 5 - Contar o número de pacientes únicos por tipo de sala

#### Oracle

```
SELECT
    r.room_type AS room_type,
    COUNT(DISTINCT p.idpatient) AS unique_patient_count
FROM
    SYSTEM.room r
JOIN
    SYSTEM.hospitalization h ON r.idroom = h.room_idroom
JOIN
    SYSTEM.episode e ON h.idepisode = e.idepisode
JOIN
    SYSTEM.patient p ON e.patient_idpatient = p.idpatient
GROUP BY
    r.room_type
ORDER BY
    unique_patient_count DESC
```

Listing 16: Implementação em Oracle da query 5

#### MongoDB

```
pipeline = [
       { '$unwind': '$episodes' },
       { '$unwind': '$episodes.events' },
       {
           '$match': {
               'episodes.events.type': 'hospitalization'
           }
       },
       {
           '$lookup': {
               'from': 'rooms',
               'localField': 'episodes.events.room',
               'foreignField': '_id',
               'as': 'room'
           }
       },
       { '$unwind': '$room' },
       {
           '$group': {
               '_id': '$room.type',
               'pacientes': { '$addToSet': '$_id' }
           }
       },
       {
           '$project': {
               '_id': 1,
               'num_pacientes': { '$size': '$pacientes' }
           }
       },
       {
           '$sort': {
               'num_pacientes': -1
           }
       }
   ]
```

```
    resultados = db.patients.aggregate(pipeline)
```

Listing 17: Implementação em MongoDB da query 5

### Neo4j

```
MATCH
    (r:Room)<-[:HAS_ROOM]-(h:Hospitalization)<-[:HAS_HOSPITALIZATION]-(e:Episode)<-[:HAS_EPISODE]-
(p:Patient)
RETURN r.room_type AS room_type, COUNT(DISTINCT p.id_patient) AS unique_patient_count
ORDER BY unique_patient_count DESC
```

Listing 18: Implementação em Neo4j da query 5

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 5.10 ms |
| MongoDB | 11.04 ms |
| Neo4j | 7.24 ms |

Table 5: Tempos de execução da query 5

## 4.6 Query 6 - Listar os tipos de salas e o custo médio por tipo

### Oracle

```
SELECT
    r.room_type AS room_type,
    AVG(r.room_cost) AS average_cost
FROM
    SYSTEM.room r
GROUP BY
    r.room_type
ORDER BY
    average_cost DESC
```

Listing 19: Implementação em Oracle da query 6

### MongoDB

```
pipeline = [
        {
            '$group': {
                '_id': '$type',
                'custo_medio': { '$avg': '$cost' }
            }
        },
        {
            '$sort': {
                'custo_medio': -1
            }
        }
```

```
    ]

    resultados = db.rooms.aggregate(pipeline)
```

Listing 20: Implementação em MongoDB da query 6

### Neo4j

```
MATCH (r:Room)
RETURN r.room_type AS room_type, AVG(r.room_cost) AS average_cost
ORDER BY average_cost DESC
```

Listing 21: Implementação em Neo4j da query 6

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 2.10 ms |
| MongoDB | 1.79 ms |
| Neo4j | 3.90 ms |

Table 6: Tempos de execução da query 6

## 4.7 Query 7 - Contar o número de funcionários por departamento

### Oracle

```
SELECT
    d.dept_name AS department_name,
    COUNT(s.emp_id) AS staff_count
FROM
    SYSTEM.staff s
JOIN
    SYSTEM.department d ON s.iddepartment = d.iddepartment
GROUP BY
    d.dept_name
ORDER BY
    staff_count DESC
```

Listing 22: Implementação em Oracle da query 7

### MongoDB

```
pipeline = [
        {
            '$project': {
                '_id': 1,
                'name': 1,
                'total_count': {
                    '$sum': [
                        { '$size': '$active' },
                        { '$size': '$inactive' }
                    ]
                }
```

```
            }
        },
        {
            '$sort': { 'total_count': -1 }
        }
    ]

    resultados = db.department.aggregate(pipeline)
```

Listing 23: Implementação em MongoDB da query 7

### Neo4j

```
MATCH (s:Staff)-[:WORKS_IN_DEPARTMENT]->(d:Department)
RETURN d.dept_name AS department_name, COUNT(s) AS staff_count
ORDER BY staff_count DESC
```

Listing 24: Implementação em Neo4j da query 7

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 2.83 ms |
| MongoDB | 1.95 ms |
| Neo4j | 5.27 ms |

Table 7: Tempos de execução da query 7

## 4.8 Query 8 - Funcionário com mais tempo de serviço ativo

### Oracle

```
SELECT
    s.emp_id AS emp_id,
    s.emp_fname AS first_name,
    s.emp_lname AS last_name,
    (ROUND((SYSDATE - s.date_joining) / 365.25, 2)) AS years_at_hospital
FROM
    SYSTEM.staff s
WHERE
    s.is_active_status = 'Y'
ORDER BY
    (SYSDATE - s.date_joining) DESC
FETCH FIRST 1 ROWS ONLY
```

Listing 25: Implementação em Oracle da query 8

### MongoDB

```
pipeline = [
        {
            "$match": {
                "date_separation": None
            }
```

```
        },
        {
            "$project": {
                "_id": 1,
                "first_name": 1,
                "last_name": 1,
                "date_joining": 1
            }
        },
        {
            "$addFields": {
                "tempo_ativo_dias": {
                    "$divide": [
                        {
                            "$subtract": [
                                datetime.now(),
                                "$date_joining"
                            ]
                        },
                        1000 * 60 * 60 * 24
                    ]
                }
            }
        },
        {
            "$addFields": {
                "tempo_ativo_anos": {
                    "$divide": ["$tempo_ativo_dias", 365.25]
                }
            }
        },
        {
            "$sort": {
                "tempo_ativo_anos": -1
            }
        },
        {
            "$limit": 1
        }
    ]

    resultado = list(db.staff.aggregate(pipeline))
```

Listing 26: Implementação em MongoDB da query 8

### Neo4j

```
MATCH (s:Staff)
WHERE s.is_active_status = 'Y'
WITH s, duration.inDays(date(s.date_joining), date()).days AS days_at_hospital
RETURN s.emp_id AS emp_id, s.emp_fname AS first_name, s.emp_lname AS last_name,
       ROUND(days_at_hospital / 365.25, 2) AS years_at_hospital
ORDER BY days_at_hospital DESC
LIMIT 1
```

Listing 27: Implementação em Neo4j da query 8

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 1.60 ms |
| MongoDB | 3.14 ms |
| Neo4j | 5.30 ms |

Table 8: Tempos de execução da query 8

## 4.9 Query 9 - O paciente com mais condições médicas e as suas condições

### Oracle

```
SELECT
    p.idpatient AS id_patient,
    p.patient_fname AS first_name,
    p.patient_lname AS last_name,
    COUNT(mh.record_id) AS condition_count,
    LISTAGG(mh.condition, ', ') WITHIN GROUP (ORDER BY mh.condition) AS conditions
FROM
    SYSTEM.patient p
JOIN
    SYSTEM.MEDICAL_HISTORY mh ON p.idpatient = mh.idpatient
GROUP BY
    p.idpatient, p.patient_fname, p.patient_lname
ORDER BY
    condition_count DESC
FETCH FIRST 1 ROWS ONLY
```

Listing 28: Implementação em Oracle da query 9

### MongoDB

```
    pipeline = [
        {
            "$project": {
                "_id": 1,
                "first_name": 1,
                "last_name": 1,
                "medical_history_count": {
                    "$size": "$medical_history"
                },
                "medical_history_conditions": "$medical_history.condition"
            }
        },
        {
            "$sort": {
                "medical_history_count": -1
            }
        },
        {
            "$limit": 1
        }
    ]

    resultado = list(db.patients.aggregate(pipeline))
```

Listing 29: Implementação em MongoDB da query 9

### Neo4j

```
MATCH (p:Patient)-[:HAS_MEDICAL_HISTORY]->(mh:Medical_History)
WITH p, COUNT(mh) AS condition_count, COLLECT(mh.condition) AS conditions
RETURN p.id_patient AS id_patient, p.patient_fname AS first_name, p.patient_lname AS
    last_name,
       condition_count, REDUCE(s = '', condition IN conditions | s + CASE WHEN s = '' THEN
           '' ELSE ', ' END + condition) AS conditions
ORDER BY condition_count DESC
LIMIT 1
```

Listing 30: Implementação em Neo4j da query 9

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 2.88 ms |
| MongoDB | 4.49 ms |
| Neo4j | 6.91 ms |

Table 9: Tempos de execução da query 9

## 4.10 Query 10 - Listar todas as hospitalizações em uma sala específicas

### Oracle

```
SELECT
    r.idroom AS room_id,
    r.room_type AS room_type,
    COUNT(h.idepisode) AS hospitalization_count
FROM
    SYSTEM.room r
LEFT JOIN
    SYSTEM.hospitalization h ON r.idroom = h.room_idroom
GROUP BY
    r.idroom, r.room_type
ORDER BY
    hospitalization_count DESC, room_id ASC
```

Listing 31: Implementação em Oracle da query 10

### MongoDB

```
rooms_collection = db['rooms']

    pipeline = [
        {
            "$lookup": {
                "from": "patients",
                "let": { "room_id": "$_id" },
                "pipeline": [
                    { "$unwind": "$episodes" },
                    { "$unwind": "$episodes.events" },
```

```
                {
                    "$match": {
                        "$expr": {
                            "$and": [
                                { "$eq": ["$episodes.events.room", "$$room_id"] },
                                { "$eq": ["$episodes.events.type", "hospitalization"] }
                            ]
                        }
                    }
                },
                { "$count": "count" }
            ],
            "as": "hospitalizations"
        }
    },
    {
        "$project": {
            "_id": 1,
            "type": 1,
            "hospitalizations": { "$ifNull": [{ "$arrayElemAt":
                ["$hospitalizations.count", 0] }, 0] }
        }
    },
    {
        "$sort": {
            "hospitalizations": -1
        }
    }
]

result = list(rooms_collection.aggregate(pipeline))
```

Listing 32: Implementação em MongoDB da query 10

### Neo4j

```
MATCH (r:Room)
OPTIONAL MATCH (r)<-[:HAS_ROOM]-(h:Hospitalization)
RETURN r.id_room AS room_id, r.room_type AS room_type, COUNT(h) AS hospitalization_count
ORDER BY hospitalization_count DESC, room_id ASC
```

Listing 33: Implementação em Neo4j da query 10

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 4.42 ms |
| MongoDB | 38.30 ms |
| Neo4j | 10.60 ms |

Table 10: Tempos de execução da query 10

## 4.11 Query 11 - Listar pacientes com mais consultas

### Oracle

```sql
SELECT
    p.idpatient,
    p.patient_fname,
    p.patient_lname,
    COUNT(a.iddoctor) AS appointment_count
FROM
    SYSTEM.patient p
JOIN
    SYSTEM.episode e ON p.idpatient = e.patient_idpatient
JOIN
    SYSTEM.appointment a ON e.idepisode = a.idepisode
GROUP BY
    p.idpatient, p.patient_fname, p.patient_lname
ORDER BY
    appointment_count DESC
```

Listing 34: Implementação em Oracle da query 11

### MongoDB

```
patients_collection = db['patients']

    # Query em MongoDB
    pipeline = [
        {
            "$project": {
                "_id": 1,
                "first_name": 1,
                "last_name": 1,
                "num_appointments": {
                    "$sum": {
                        "$map": {
                            "input": "$episodes",
                            "as": "episode",
                            "in": {
                                "$size": {
                                    "$filter": {
                                        "input": "$$episode.events",
                                        "as": "event",
                                        "cond": { "$eq": ["$$event.type", "appointment"] }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        },
        {
        '$match': {
            'num_appointments': {'$gt': 0}
        }
    },
        { "$sort": { "num_appointments": -1 } }
    ]

    result = list(patients_collection.aggregate(pipeline))
```

Listing 35: Implementação em MongoDB da query 11

### Neo4j

```
MATCH (p:Patient)-[:HAS_EPISODE]->(e:Episode)-[:HAS_APPOINTMENT]->(a:Appointment)
RETURN p.id_patient AS idpatient, p.patient_fname AS patient_fname, p.patient_lname AS
    patient_lname, COUNT(a) AS appointment_count
ORDER BY appointment_count DESC
```

Listing 36: Implementação em Neo4j da query 11

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 4.00 ms |
| MongoDB | 4.23 ms |
| Neo4j | 6.88 ms |

Table 11: Tempos de execução da query 11

## 4.12 Query 12 - Listar o custo total da fatura por paciente

### Oracle

```
SELECT
    p.idpatient,
    p.patient_fname,
    p.patient_lname,
    SUM(b.total) as sum_total_bill
FROM
    SYSTEM.patient p
JOIN
    SYSTEM.episode e ON p.idpatient = e.patient_idpatient
JOIN
    SYSTEM.bill b ON e.idepisode = b.idepisode
GROUP BY
    p.idpatient, p.patient_fname, p.patient_lname
ORDER BY
    sum_total_bill DESC
```

Listing 37: Implementação em Oracle da query 12

### MongoDB

```
patients_collection = db['patients']

    # Pipeline de  agregao  para calcular o custo total de bill por paciente
    pipeline = [
        {
            "$unwind": "$bills"
        },
        {
            "$lookup": {
                "from": "bills",
                "localField": "bills",
```

```
            "foreignField": "_id",
            "as": "bill_details"
        }
    },
    {
        "$unwind": "$bill_details"
    },
    {
        "$group": {
            "_id": "$_id",
            "first_name": { "$first": "$first_name" },
            "last_name": { "$first": "$last_name" },
            "total_bill_cost": { "$sum": "$bill_details.total" }
        }
    },
    {
        "$sort": {
            "total_bill_cost": -1
        }
    }
]


result = list(patients_collection.aggregate(pipeline))
```

Listing 38: Implementação em MongoDB da query 12

### Neo4j

```
MATCH (p:Patient)-[:HAS_EPISODE]->(e:Episode)-[:HAS_BILL]->(b:Bill)
RETURN p.id_patient AS idpatient, p.patient_fname AS patient_fname, p.patient_lname AS
    patient_lname, SUM(b.total) AS sum_total_bill
ORDER BY sum_total_bill DESC
```

Listing 39: Implementação em Neo4j da query 12

### Resultados

| Base de Dados | Tempo final |
|---|---|
| Oracle | 2.80 ms |
| MongoDB | 5.90 ms |
| Neo4j | 9.79 ms |

Table 12: Tempos de execução da query 12

## 4.13 Query 13 - Obter a média da estadia hospitalar

### Oracle

```
SELECT
    ROUND(AVG(h.discharge_date - h.admission_date), 2) AS average_length_of_stay
FROM
    SYSTEM.hospitalization h
```

Listing 40: Implementação em Oracle da query 13

### MongoDB

```
patients_collection = db['patients']
   pipeline = [
       {
           "$unwind": "$episodes"
       },
       {
           "$unwind": "$episodes.events"
       },
       {
           "$match": {
               "episodes.events.type": "hospitalization",
               "episodes.events.admission_date": { "$exists": True },
               "episodes.events.discharge_date": { "$exists": True }
           }
       },
       {
           "$project": {
               "duracao_estadia": {
                   "$divide": [
                       {
                           "$subtract": [
                               { "$toDate": "$episodes.events.discharge_date" },
                               { "$toDate": "$episodes.events.admission_date" }
                           ]
                       },
                       86400000 # Converter milissegundos para dias
                   ]
               }
           }
       },
       {
           "$group": {
               "_id": None,
               "media_total_duracao_estadia": { "$avg": "$duracao_estadia" }
           }
       },
       {
           "$project": {
               "_id": 0,
               "media_total_duracao_estadia": { "$round": ["$media_total_duracao_estadia",
                   2] }
           }
       }
   ]

   result = list(patients_collection.aggregate(pipeline))
```

Listing 41: Implementação em MongoDB da query 13

### Neo4j

```
MATCH (h:Hospitalization)
RETURN ROUND(AVG(TOFLOAT(duration.inDays(h.admission_date, h.discharge_date).days)), 2) as
    avg_hospitalization_stay
```

Listing 42: Implementação em Neo4j da query 13

**Resultados**

| Base de Dados | Tempo final |
|---|---|
| Oracle | 1.21 ms |
| MongoDB | 2.95 ms |
| Neo4j | 4.61 ms |

Table 13: Tempos de execução da query 13

### 4.14 Script para comparar os resultados das queries e calcular tempos de execução

Como mencionado anteriormente, o grupo desenvolveu um script denominado `compare.py`, que permite não só comparar os resultados das 13 diferentes queries, mas também calcular o tempo de execução final de cada query, correspondendo a uma média de 10 medições. Para utilizar este script, recorremos inicialmente a 3 scripts Python, denominados `mongo_queries.py`, `neo4j.py` e `sql.py`, onde, no interior de cada um, temos funções para executar cada uma das queries nas respetivas bases de dados, retornando o resultado em forma de dicionário. No `compare.py`, os três dicionários são comparados para cada query e, caso sejam iguais, considera-se que todas as queries deram o mesmo resultado. Além disso, dentro do mesmo script, temos uma função que permite ignorar a primeira execução das queries e calcular o tempo de execução médio com base em 10 medições.
Após executar o `compare.py`, é criada uma pasta `out` onde, para cada query, temos 4 ficheiros `.txt` diferentes: o `queryX_mongodb.txt`, onde é possível ver o resultado da query MongoDB e o respetivo tempo final de execução; o `queryX_neo4j.txt`, onde é possível ver o resultado da query Neo4j e o respetivo tempo final de execução; o `queryX_sql.txt`, onde é possível ver o resultado da query SQL e o respetivo tempo final de execução; e, por fim, o `queryX.txt`, onde é indicado se os resultados das queries são iguais.

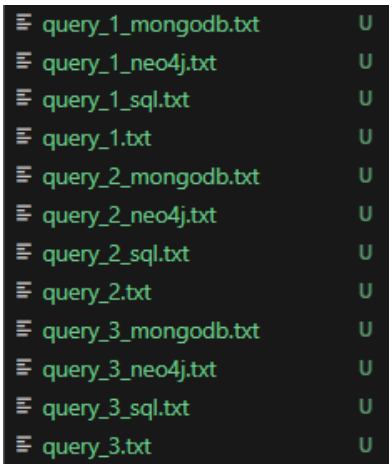


Figure 3: Uma parte da pasta out

# 5 Discussão

Neste capítulo, analisamos as diferenças de performance, vantagens e desvantagens entre os tipos de dados utilizados no projeto: MongoDB, uma base de dados orientada a documentos, e Neo4j, uma base de dados orientada a grafos. Ambas foram escolhidas pela sua relevância ao projeto proposto, pela familiaridade com ambas e pelas características distintas que oferecem para a gestão de dados complexos como era o caso de estudo.

## 5.1 Performance

Ao comparar as performances de MongoDB e Neo4j, várias considerações devem ser tidas em conta. A implementação das varias queries foi realizada de forma a obter resultados que pudessem ser comparados diretamente entre os tres esquemas de bases de dados. Estas queries foram executadas múltiplas vezes e, para assegurar consistência e precisão desta analise, tivemos em conta a média dos tempos de execução. A partir destes testes chegamos as seguintes observacoes.

### 5.1.1 MongoDB

- Em geral, MongoDB apresentou tempos de execução rápidos para consultas simples e agregações básicas devido à sua natureza de armazenamento de documentos.
- No entanto, para consultas que exigem operações mais complexas e relacionamentos entre diferentes estruturas de dados, os tempos de execução foram mais elevados do que o esquema relacional ou orientado a grafos.

### 5.1.2 Neo4j

- Neo4j demonstrou superioridade em consultas que envolvem múltiplos relacionamentos e navegação entre nós (nodes).
- A capacidade de Neo4j em realizar operações complexas de grafos de maneira eficiente resultou em tempos de execução menores para consultas específicas deste tipo.

A comparação dos tempos de execução das queries desenvolvidas mostra claramente que, enquanto MongoDB se destaca como uma alternativa sólida de forma geral, o Neo4j oferece vantagens significativas em cenários que representam relações entre entidades, onde a modelagem de grafos é benéfica.

## 5.2 Vantagens e Desvantagens

### 5.2.1 MongoDB

**Vantagens**:

- *Flexibilidade*: MongoDB permite um esquema flexível, ideal para dados sem estrutura definida e que podem mudar com frequência.
- *Escalabilidade*: Projetado para escalar horizontalmente, este motor de bases de dados lida bem com grandes volumes de informação e transações.
- *Facilidade de uso*: Sintaxe similar a documentos JSON, tornando-o acessível, fácil de compreender e utilizar.

**Desvantagens**:

- *Desempenho em consultas complexas*: MongoDB pode sofrer um pouco com desempenho quando se trata de consultas complexas que exigem múltiplos relacionamentos entre documentos.

### 5.2.2 Neo4j

**Vantagens**:

- *Modelagem de grafos*: Excelente para dados que beneficiam de uma estrutura orientada a grafos, como relações entre entidades, recomendações e outras aplicações que envolvem relacionamentos complexos.
- *Desempenho em consultas complexas*: Realiza operações complexas de maneira eficiente, com um desempenho satisfatório mesmo em consultas que envolvem múltiplos níveis de relacionamentos.
- *Linguagem Cypher*: Linguagem de consulta intuitiva e específica para grafos.

**Desvantagens**:

- *Escalabilidade*: Embora Neo4j escale bem verticalmente, este pode enfrentar desafios ao escalar horizontalmente para conjuntos muito grandes de dados distribuídos.
- *Complexidade de implementação*: Pode ser mais complexo de configurar e alterar este tipo de bases de dados.

## 5.3 Considerações Finais

A escolha entre MongoDB e Neo4j deve ser guiada pelos requisitos específicos do projeto. Assim sendo, o MongoDB é vantajoso para aplicações que necessitam de flexibilidade nos seus dados e alta escalabilidade horizontal. Por outro lado, Neo4j destaca-se em cenários onde a modelagem de grafos e a eficiência em consultas sobre as relações são cruciais.

# 6 Conclusão e Trabalho Futuro

Dado por concluído o projeto prático, consideramos que o trabalho realizado cumpre com as expectativas definidas no inicio do desenvolvimento e com os requisitos propostos. Este trabalho inclui, de forma eficaz, as temáticas abordadas ao longo do semestre na unidade curricular de Bases de Dados NoSQL para garantir uma solução robusta independente do tipo SGBD.
Relativamente ao trabalho futuro, consideramos que seria uma mais-valia a implementar a sicronização eficiente entre as várias bases de dados, permitindo usufruir das principais vantagens de cada uma delas para as diferentes necessidades do sistema. Para além disto, os próximos passos devem envolver uma comunicação direta com os stakeholders do sistema de modo a compreender de forma detalhada as suas necessidades relativas ao sistema em desenvolvimento. Isto permitirá a criação de novas views, trigger e procedures relevantes ao projeto.

# 7 Adenda

Tendo em conta o *feedback* recebido durante a apresentação, foi adicionada esta adenda com o intuito de melhorar a implementação da base de dados não relacional orientada a grafos. A primeira versão do Neo4j focou-se numa migração direta das tabelas para nodos e das chaves estrangeiras para relações, baseando-se na estrutura do esquema original para facilitar a interpretação. No entanto, esta abordagem não tirava pleno proveito das vantagens oferecidas por uma modelação orientada a grafos. Para optimizar o desempenho e a flexibilidade na manipulação dos dados, foi necessária uma reavaliação do esquema inicial. Esta segunda versão do Neo4j envolveu ajustes específicos para maximizar os benefícios da modelação orientada a grafos, sem perder a integridade da informação da base de dados relacional original. Assim sendo, as principais mudanças introduzidas foram as seguintes:

- Os nodos **Prescription** e **Lab_Screening** deram origem a relações com propriedades: **PRESCRIBED** e **PERFORMED** respetivamente. Esta mudança deve-se ao facto de considerarmos que as tabelas **Prescription** e **Lab_Screening** no modelo lógico da base de dados relacional devem ser tratadas como "Attributed JOIN Tables". Assim, optámos por transformar essas tabelas em relações com atributos para melhor representar a estrutura e os dados relacionais.

- Os nodos **Doctor**, **Technician** e **Nurse** deixaram de existir, sendo agora utilizados apenas nodos **Staff**. Desta forma, no nodo staff foi introduzida a propriedade **position**, que indica a posição que o membro da equipa ocupa no hospital (ou seja, pode ser médico, técnico ou enfermeiro). No caso dos médicos, os nodos ainda possuem o atributo **qualifications**. Assim, considerámos que poderíamos reduzir o número de nodos dessas três tabelas e apenas guardar a informação delas nos nodos **Staff**, melhorando a eficiência e a gestão dos dados no nosso sistema.

- Os nodos **Emergency_Contact** foram convertidos numa lista dentro dos nodos **Patient**. Para isso, foi necessário serializar a informação dos contactos de emergência utilizando strings JSON. Esta alteração deve-se ao facto da relação entre a tabela Patient e a tabela Emergency_Contact ser 1:N.

- Os nodos **Appointment** e **Hospitalizations** passaram a ter os seus atributos dentro dos nodos **Episode**. Esta mudança ocorreu porque identificámos que desta forma era possível reduzir o número total de nodos na nossa base de dados, e os nodos **Episode** passariam a conter mais informação do que apenas os dois IDs que tinham anteriormente. Além disso, esta alteração deve-se ao facto de Episode ter uma relação 1:1 com as tabelas Appointment e Hospitalizations, permitindo assim uma estrutura de dados mais eficiente e simplificada.

Estas alterações acabaram por dar origem ao novo esquema da base de dados Neo4j, representado abaixo na Figura 4.

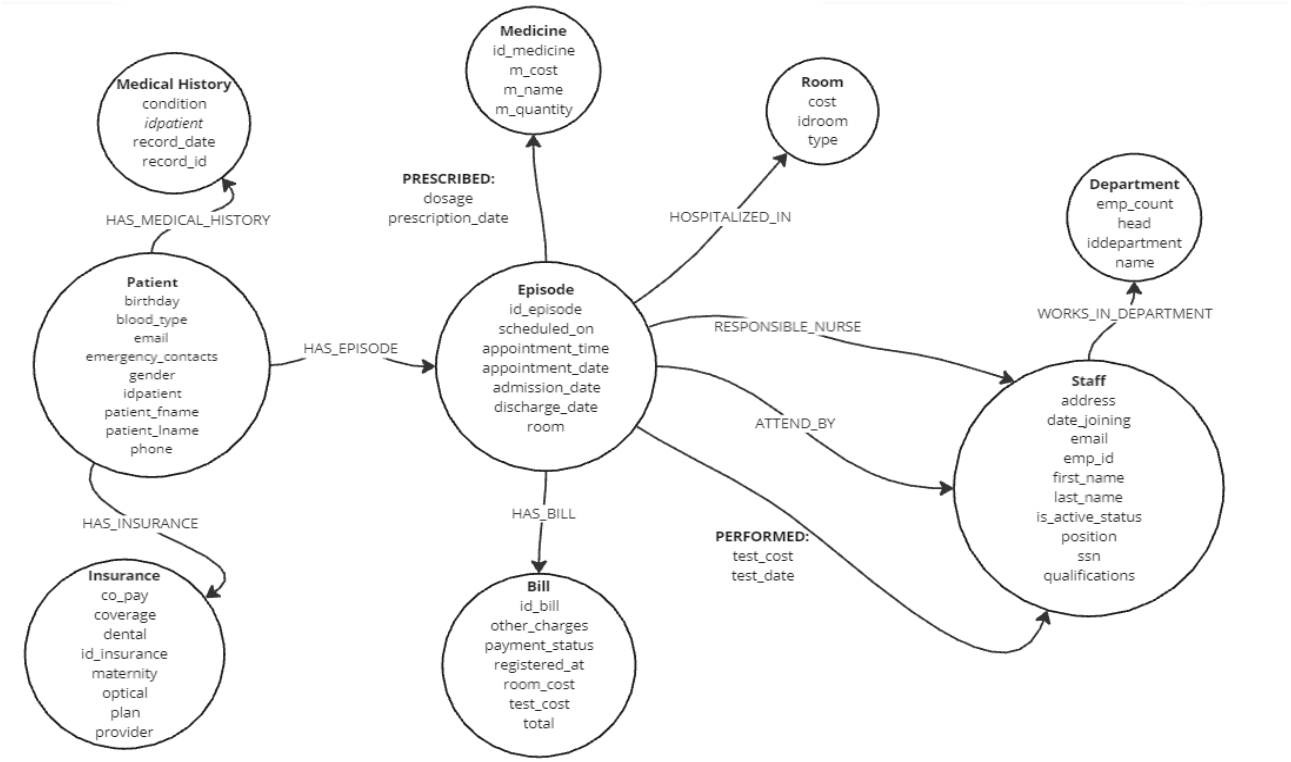


Figure 4: Esquema da arquitetura do Neo4j

Relativamente às *Procedures*, estas permaneceram iguais à última versão, devido ao facto da estrutura do nodo Bill não ter mudado, e portanto não tendo havido necessidade de realizar mudanças.
No que toca à *View* apresentada anteriormente, foram introduzidas alterações devido ao facto de que alguns nodos deixaram de existir, e outros *merged* num só nodo. Portanto, a maneira como alguns atributos eram acessados teve que mudar para acomodar a nova estrutura, mas a ação feita pela query é a mesma.
Um ponto a destacar é que optamos por não remover os ficheiros relativos à implementação antiga do Neo4j e adicionamos apenas dois novos ficheiros: o *neo4j_novo.py* na pasta Migrations, responsável pela migração da base de dados relacional para a base de dados orientada a grafos, e o ficheiro *neo4j_novas_queries.py* na pasta Queries, que contém as novas queries desenvolvidas.

## 7.1 Resultados e Análise

Com um novo esquema, reescrevemos as queries mencionadas anteriormente conforme a nova estrutura do grafo e posteriormente recolhemos novamente os tempos de execução para cada uma das três bases de dados. Como podemos observar na Tabela 14, os resultados do Neo4j apresentaram melhorias nos seus tempos de execução, sendo estes muito próximos dos tempos das outras duas bases de dados. Em alguns casos, os resultados do Neo4j foram até melhores do que os das outras duas bases de dados, como por exemplo nas queries 4, 5 e 7.

| **Queries Novas** | **Oracle (ms)** | **Mongo (ms)** | **Neo4j (ms)** |
|---|---|---|---|
| 1- Listar por ordem decrescente os medicamentos mais caros | 2.99 | 1.82 | 3.31 |
| 2- Listar pacientes que têm mais de 3 episódios por ordem decrescente | 2.67 | 2.52 | 3.00 |
| 3- Listar pacientes e os seus contactos de emergência | 2.20 | 2.21 | 7.31 |
| 4- Listar as salas com o maior custo de hospitalização total | 4.30 | 57.70 | 3.49 |
| 5- Contar o número de pacientes únicos por tipo de sala | 3.00 | 8.63 | 2.98 |
| 6- Listar os tipos de salas e o custo médio por tipo | 1.59 | 1.75 | 1.76 |
| 7- Contar o número de funcionários por departamento | 2.21 | 2.52 | 1.89 |
| 8- Funcionário com mais tempo de serviço ativo | 0.99 | 2.21 | 2.45 |
| 9- O paciente com mais condições médicas e as suas condições | 1.89 | 2.62 | 1.94 |
| 10- Listar todas as hospitalizações em uma sala específica | 2.38 | 21.87 | 3.79 |
| 11- Listar pacientes com mais consultas | 2.49 | 2.55 | 4.06 |
| 12- Listar o custo total da fatura por paciente | 2.57 | 4.89 | 2.96 |
| 13- Obter a média da estadia hospitalar | 0.84 | 2.20 | 2.73 |

Table 14: Comparação de desempenho das queries novas entre Oracle, Mongo e Neo4j

Os novos resultados apresentaram melhorias significativas, quando comparados com a versão anterior. Estes consolidam as conclusões discutidas no Capítulo 5.