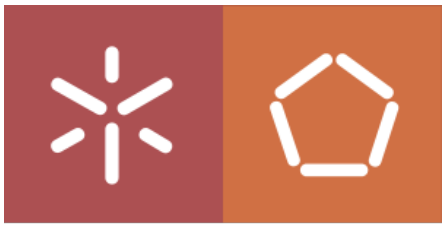


Universidade do Minho
Escola de Engenharia

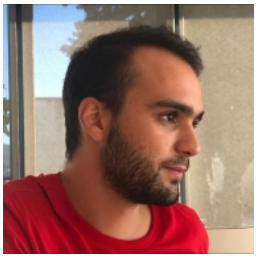
Mestrado em Engenharia Informática

Bases de Dados NoSQL

Trabalho Prático (23/24)

Afonso Miguel Matos Bessa

pg53597



Francisco Luís Rodrigues Claudino

pg50380

João Paulo Machado Abreu

pg53928

João Pedro Dias Faria

pg53939

Ricardo Cardoso Sousa

pg54179

10 de junho de 2024

Conteúdo

1 Introdução **1**

2 Arquitetura **2**

3 Base de dados relacional - Oracle SQL **4**

3.1 Exploração da Base de Dados 5

3.1.1 Queries 6

3.1.2 Functions 6

3.1.3 Procedures 8

3.1.4 Triggers 11

3.1.5 Queries, Functions, Procedures e Triggers implementados 13

4 Base de dados documental - MongoDB **26**

4.1 Coleção Pacientes 27

4.2 Coleção Staff 28

4.3 Coleção Episódios 29

4.4 Triggers em MongoDB 30

4.4.1 Triggers para a adição de identificadores únicos 31

4.4.2 Trigger proveniente da base de dados Oracle 32

4.5 Scripts para gestão dos triggers no MongoDB Atlas 35

4.6 Exploração da Base de Dados 37

4.6.1 Operações Básicas em MongoDB 41

4.6.2 Visão Paciente 42

4.6.3 Visão Staff 44

4.6.4 Visão Episodes 46

5 Base de dados orientada a grafos - Neo4J **51**

5.1 *Constraints* em Neo4j 54

5.2 Triggers em Neo4j 57

5.2.1 Triggers para a adição de identificadores únicos 57

5.2.2 Trigger proveniente da base de dados Oracle 59

5.3 Exploração da Base de Dados 61

5.3.1 Operações Básicas em Neo4J 63

5.3.2 Visão Paciente 63

5.3.3 Visão Staff 66
5.3.4 Visão Episodes 68
5.3.5 Visão Global 70

6 Análise Crítica **73**

7 Conclusão **74**

A Nodos desenvolvidos em Neo4J **75**
A.1 Paciente 75
A.2 Seguro de Saúde 75
A.3 Histórico médico 75
A.4 Contactos de Emergência 76
A.5 Episódios Médicos 76
A.6 Faturas 76
A.7 Prescrição Médica 76
A.8 Medicamento 77
A.9 Departamento 77
A.10 Funcionários 77
A.11 Consulta 77
A.12 Testes Médicos 78
A.13 Hospitalização 78
A.14 Quarto 78
A.15 Contador 78

Capítulo 1

Introdução

No âmbito da unidade curricular de Bases de Dados NoSQL inserida no Mestrado em Engenharia Informática foi desenvolvido um projeto prático com o objetivo de explorar e aplicar diferentes paradigmas de bases de dados. Este projeto tem como foco a migração de dados de um sistema de gestão hospitalar baseado em **SQL** (Oracle SQL) para dois sistemas não relacionais distintos: **MongoDB**, uma base de dados orientada a documentos, e **Neo4j**, uma base de dados orientada a grafos.

A importância desta migração reside na necessidade de adaptar os sistemas de base de dados às crescentes demandas e complexidades dos dados modernos. Enquanto as bases de dados relacionais tradicionais oferecem uma estrutura rígida e tabular, as bases de dados NoSQL proporcionam flexibilidade e escalabilidade para lidar com volumes massivos de dados, estruturas variadas e relações complexas.

Ao longo deste trabalho, foram definidos processos para a migração eficaz dos dados, a implementação de consultas para demonstrar as capacidades operacionais dos sistemas implementados e uma análise crítica comparando as funcionalidades dos sistemas não relacionais com o sistema relacional original.

Este relatório técnico apresenta de forma detalhada o trabalho desenvolvido, desde a análise inicial até à implementação final dos sistemas de base de dados não relacionais, evidenciando as estratégias adotadas, os desafios enfrentados e os resultados obtidos.

Capítulo 2

Arquitetura

A arquitetura do projeto é representada na Figura 2.1. Este ilustra os caminhos de migração do **Oracle SQL** para o **MongoDB** e **Neo4j**, e destaca a fase de exploração para cada sistema de bases de dados.

- **Oracle SQL**: A base de dados relacional de origem, contendo várias tabelas como pacientes, staff, episódios, entre outras, que representam os dados do hospital.
- **MongoDB**: Uma base de dados orientada a documentos onde os dados são armazenados em documentos flexíveis, semelhantes a JSON. Este sistema é utilizado para gerir a natureza semi-estruturada e hierárquica dos dados do hospital.
- **Neo4j**: Uma base de dados orientada a grafos, desenhada para gerir dados altamente conectados. É empregada para representar as relações complexas entre diferentes entidades dentro1 do sistema hospitalar.

Os principais objetivos deste projeto incluem a **Migração de Dados**, a **Exploração do Sistema** e a **Análise Crítica**. A migração de dados consiste em transferir informações de uma base de dados Oracle SQL existente, que armazena dados de um sistema de gestão hospitalar, para MongoDB e Neo4j. Este processo envolve compreender a estrutura dos dados, definir as transformações necessárias e implementar scripts para facilitar a transferência.

A exploração do sistema visa implementar um conjunto de consultas para demonstrar as capacidades operacionais dos novos sistemas não relacionais. Isto envolve investigar como os dados podem ser acedidos, manipulados e utilizados dentro do MongoDB e Neo4j, em comparação com a base de dados Oracle SQL original.

Por fim, a análise crítica foca-se em realizar uma comparação dos modelos de bases de dados relacionais e não relacionais, avaliando as vantagens, limitações e desempenho de cada sistema no tratamento dos dados do hospital.

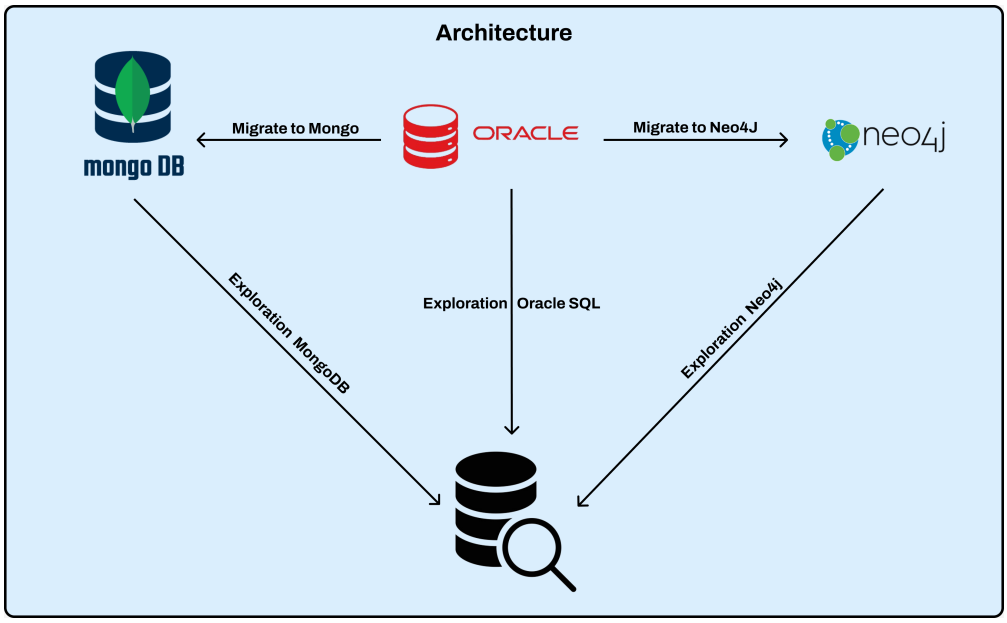


Figura 2.1: Arquitetura

Capítulo 3

Base de dados relacional - Oracle SQL

A linguagem **SQL** (Structured Query Language) é um elemento central na gestão de sistemas de bases de dados relacionais, como o **Oracle SQL**. Esta linguagem permite a definição, manipulação e consulta de dados, sendo fundamental para a administração eficiente e eficaz de grandes volumes de informações. No contexto do Oracle SQL, os comandos SQL possibilitam a criação de estruturas complexas, a realização de consultas detalhadas e a implementação de regras de negócios através de procedimentos armazenados e funções.

Neste capítulo, exploraremos os princípios fundamentais do Oracle SQL aplicados ao desenvolvimento e gestão de uma base de dados para um sistema hospitalar. Abordaremos desde a criação de *tablespaces*, que são *containers* lógicos para armazenamento físico dos objetos de banco de dados, até a definição de utilizadores, permissões e a construção de esquemas complexos que refletem as operações diárias de uma instituição de saúde.

Para iniciar o projeto de um sistema de base de dados para um hospital, o primeiro passo foi criar um *tablespace*. Um *tablespace* é um *container* lógico onde todos os objetos do banco de dados, como tabelas e índices, são armazenados fisicamente. Utilizámos o seguinte comando SQL para criar o *tablespace*:

Assim, para iniciar o projeto de um sistema de base de dados para um sistema hospitalar, o primeiro passo foi criar um *tablespace*. Um *tablespace* é um *container* lógico onde todos os objetos do banco de dados, como tabelas e índices, são armazenados fisicamente.

Usámos o seguinte comando SQL para criar o *tablespace*:

```
1 CREATE TABLESPACE hospital_tables
2 DATAFILE 'hospital_files_01.dbf'
3 SIZE 200M REUSE AUTOEXTEND ON
4 NEXT 100M MAXSIZE UNLIMITED;
```

Aqui, criámos um *tablespace* chamado `hospital_tables` e associámo-lhe um ficheiro de dados denominado `hospital_files_01.dbf`. O tamanho inicial do ficheiro de dados foi de 200 *megabytes*, podendo crescer automaticamente em incrementos de 100 *megabytes* até um tamanho ilimitado.

De seguida, criámos um novo utilizador chamado **hospital** com a senha **hospital**. Definimos o `hospital_tables` como o *tablespace* padrão para este utilizador e concedemos-lhe uma quota ilimitada para utilização do espaço nesse *tablespace*. Utilizámos o seguinte comando:

```
CREATE USER hospital IDENTIFIED BY "hospital" DEFAULT TABLESPACE
↪ hospital_tables
QUOTA UNLIMITED ON hospital_tables;
```

Para que o utilizador hospital pudesse conectar-se ao banco de dados e criar objetos, concedemos os privilégios necessários com o comando:

```
GRANT CONNECT, RESOURCE, CREATE VIEW TO hospital;
```

Após a criação do *tablespace* e do utilizador, o próximo passo foi definir o esquema atual para o utilizador **Hospital** e criar as sequências, tabelas, chaves primárias (PK) e chaves estrangeiras (FK). Para isso, executamos o ficheiro `hospital.sql` que para além das estruturas da base de dados, possui também os comandos para inserir dados iniciais, como informações de pacientes, funcionários, médicos e outras entidades fundamentais para o funcionamento do sistema hospitalar.

3.1 Exploração da Base de Dados

Na etapa de exploração da base de dados relacional, o grupo optou por realizar uma subdivisão do sistema hospitalar em três grandes visões: **Pacientes**, **Staff** e **Episodes**. De referir que esta divisão foi efetuada de forma meramente académica, com o objetivo de facilitar a exploração da Base de Dados. Ajudou a compreender melhor o funcionamento do sistema hospitalar e facilitou o processo de exploração da Base de Daddos Relacional.

A primeira visão, **Pacientes**, abrange todas as informações relacionadas aos pacientes, que são acessíveis e passíveis de inserção, atualização e eliminação por parte dos utilizadores numa plataforma hospitalar. Esta categoria inclui o acesso a informações pessoais, contactos de emergência, detalhes do seguro e histórico médico.

De maneira semelhante, a segunda visão **Staff**, engloba todas as informações relativas aos membros responsáveis pelo funcionamento de uma unidade hospitalar, isto é, todos os médicos, enfermeiros e técnicos, bem como o departamento onde estes desempenham as suas funções.

Por fim, a visão **Episodes** alberga todas as informações internas do sistema hospitalar, incluindo prescrições médicas, episódios de atendimento, faturação, gestão de salas, triagem laboratorial e outras operações administrativas essenciais. Esta categoria é fundamental para assegurar a eficiência e a organização das atividades hospitalares, permitindo uma gestão integrada e eficaz de todos os recursos e processos.

No entanto, para explorar ao máximo cada detalhe existente no sistema hospitalar, será frequentemente necessário envolver uma ou mais categorias de forma a obter uma visão completa e integrada das informações. Esta abordagem cruzada dados entre **Pacientes**, **Staff** e **Episodes** possibilita uma análise mais detalhada e precisa das operações hospitalares, melhorando assim a tomada de decisões e a eficiência dos processos internos.

Virando agora o foco para a exploração em SQL, para realizar a exploração mais detalhada e pormenorizada possível, recorremos essencialmente a quatro tipos distintos de ferramentas: **Queries**, **Functions**, **Procedures** e **Triggers**.

3.1.1 Queries

Primeiramente, as **Queries** são usadas para interrogar a base de dados, permitindo a recuperação de dados específicos de acordo com critérios definidos. Estas consultas são fundamentais para obter informações precisas e rápidas sobre os diversos aspetos do sistema hospitalar, como dados de pacientes, histórico médico, disponibilidade de staff e detalhes de faturação.

Um exemplo de uma Query é apresentado abaixo:

```
SELECT M.M_NAME, M.M_QUANTITY, M.M_COST
FROM Hospital.Medicine M
ORDER BY M.M_NAME;
```

Neste exemplo, a query seleciona o nome, quantidade e custo de todos os medicamentos armazenados na tabela *Medicine*, ordenados por nome.

3.1.2 Functions

As **Functions**, por outro lado, aceitam parâmetros, o que permite uma exploração mais aprofundada e personalizada de acordo com as necessidades do utilizador. Esta flexibilidade é crucial para simular o funcionamento real de um hospital, onde as operações muitas vezes dependem de variáveis dinâmicas e requisitos específicos, como a verificação de disponibilidade de médicos ou a análise de resultados de exames laboratoriais para um dado paciente.

Um exemplo de uma Function é apresentado abaixo:

```
CREATE OR REPLACE TYPE PatientInsurance_PlanRow AS OBJECT (
  IDPATIENT NUMBER(38,0),
  PATIENT_FNAME VARCHAR2(45),
  PATIENT_LNAME VARCHAR2(45),
  BLOOD_TYPE VARCHAR2(3),
  PHONE VARCHAR2(12),
  EMAIL VARCHAR2(50),
  GENDER VARCHAR2(10),
  POLICY_NUMBER VARCHAR2(45),
  BIRTHDAY DATE,
  INSURANCE_PLAN VARCHAR2(45)
);
```

Este excerto de código cria um tipo de objeto chamado *PatientInsurance_PlanRow*, que é uma estrutura para armazenar informações de pacientes, incluindo detalhes como nome, tipo sanguíneo, telefone, email, género, número da apólice de seguro, data de nascimento e plano de seguro. Cada campo possui um tipo de dado específico, por exemplo, *IDPATIENT* é um número, *PATIENT_FNAME* é uma string de até 45 caracteres, *BIRTHDAY* é uma data, entre outros.

```
CREATE OR REPLACE TYPE PatientInsurance_PlanTable IS TABLE OF
↪ PatientInsurance_PlanRow;
```

Aqui, definimos um tipo de tabela de objetos chamado *PatientInsurance_PlanTable*. Esta tabela é uma coleção de objetos do tipo *PatientInsurance_PlanRow*, permitindo armazenar múltiplas linhas de dados de pacientes.

```
CREATE OR REPLACE FUNCTION ListPatientsByInsurancePlan(plan_name IN VARCHAR2)
  RETURN PatientInsurance_PlanTable PIPELINED IS
BEGIN
  FOR rec IN (
    SELECT p.IDPATIENT, p.PATIENT_FNAME, p.PATIENT_LNAME, p.BLOOD_TYPE,
    ↪ p.PHONE,
          p.EMAIL, p.GENDER, p.POLICY_NUMBER, p.BIRTHDAY, i.INSURANCE_PLAN
    FROM HOSPITAL.PATIENT p
    JOIN HOSPITAL.INSURANCE i ON p.POLICY_NUMBER = i.POLICY_NUMBER
    WHERE i.INSURANCE_PLAN = plan_name
  ) LOOP
    PIPE ROW (PatientInsurance_PlanRow(
      rec.IDPATIENT, rec.PATIENT_FNAME, rec.PATIENT_LNAME, rec.BLOOD_TYPE,
      ↪ rec.PHONE,
      rec.EMAIL, rec.GENDER, rec.POLICY_NUMBER, rec.BIRTHDAY,
      ↪ rec.INSURANCE_PLAN
    ));
  END LOOP;
  RETURN;
END ListPatientsByInsurancePlan;
```

Esta função, *ListPatientsByInsurancePlan*, aceita um parâmetro *plan_name* do tipo *VARCHAR2* e retorna uma tabela de objetos *PatientInsurance_PlanTable*. A função usa um loop *FOR* para iterar sobre os resultados de uma consulta que seleciona informações de pacientes a partir das tabelas *PATIENT* e *INSURANCE*, onde o plano de seguro corresponde ao valor fornecido em *plan_name*. Para cada linha resultante, a função usa *PIPE ROW* para adicionar um objeto *PatientInsurance_PlanRow* à tabela resultante, que contém os dados do paciente.

```sql
SELECT * FROM TABLE(ListPatientsByInsurancePlan('Standard Plan'));
```

Esta consulta executa a função *ListPatientsByInsurancePlan* passando o valor *Standard Plan* como argumento. O resultado é uma tabela com todas as informações dos pacientes que estão no plano de seguro.

Cada parte deste código SQL trabalha em conjunto para permitir a criação e utilização de uma função que retorna dados estruturados de pacientes com base no plano de seguro. Isso facilita a personalização e a flexibilidade ao consultar dados em um cenário de hospital real.

3.1.3 Procedures

As **Procedures** são blocos de código armazenados que executam uma série de operações SQL de forma automatizada. Estas são particularmente úteis para realizar tarefas repetitivas e complexas de forma eficiente, como a atualização de registos de pacientes, o processamento de faturações e a gestão de inventário de medicamentos. As Procedures permitem a automação de processos críticos, garantindo a consistência e a integridade dos dados ao longo do tempo.

Embora as Views sejam úteis para simplificar consultas complexas e melhorar a segurança ao restringir colunas visíveis, elas têm limitações significativas em comparação com as Procedures. As Views não permitem lógica condicional ou controlos avançados de fluxo, têm capacidades limitadas para operações de escrita complexas, e podem introduzir overhead de performance em consultas complexas. As Procedures, por outro lado, são mais flexíveis e eficientes para automação de processos críticos, permitindo encapsular lógica de negócios complexa, realizar operações de escrita robustas e otimizar performance, além de facilitar a manutenção e reutilização do código. Portanto, para tarefas repetitivas e complexas, as Procedures são uma escolha mais adequada do que as Views.

Um exemplo de uma Procedure é apresentado abaixo:

```sql
DECLARE
    max_id NUMBER;
BEGIN
    SELECT COALESCE(MAX(IDPATIENT), 0) INTO max_id FROM Hospital.PATIENT;

    BEGIN
        EXECUTE IMMEDIATE 'DROP SEQUENCE patient_seq_new';
    EXCEPTION
        WHEN OTHERS THEN
            IF SQLCODE != -2289 THEN
                RAISE;
```

```
12              END IF;
13      END;
```

No primeiro bloco, declaramos uma variável *max_id* para armazenar o valor máximo atual da coluna *IDPATIENT* na tabela *Patient*. Usamos a função *COALESCE* para garantir que, caso não haja registos, *max_id* seja definido como 0. Em seguida, usamos *EXECUTE IMMEDIATE* para criar uma nova sequência chamada *patient_seq_new*, que começa a partir do valor *max_id + 1* e incrementa por 1 para cada novo registo. Esta sequência é usada para gerar IDs únicos para novos pacientes, evitando conflitos de chave primária.

```
1  CREATE OR REPLACE PROCEDURE insert_patient (
2      p_patient_fname VARCHAR2,
3      p_patient_lname VARCHAR2,
4      p_blood_type    VARCHAR2,
5      p_phone         VARCHAR2,
6      p_email         VARCHAR2,
7      p_gender        VARCHAR2,
8      p_birthday      DATE,
9      p_policy_number VARCHAR2,
10     p_condition     VARCHAR2,
11     p_record_date   DATE,
12     p_contact_name  VARCHAR2,
13     p_contact_phone VARCHAR2,
14     p_contact_relation VARCHAR2,
15     p_provider      VARCHAR2,
16     p_insurance_plan VARCHAR2,
17     p_co_pay        NUMBER,
18     p_coverage      VARCHAR2,
19     p_maternity     CHAR,
20     p_dental        CHAR,
21     p_optical       CHAR
22 ) IS
23     v_idpatient NUMBER;
24 BEGIN
25     INSERT INTO Hospital.PATIENT (
26         IDPATIENT, PATIENT_FNAME, PATIENT_LNAME, BLOOD_TYPE, PHONE, EMAIL,
           ↪  GENDER, BIRTHDAY, POLICY_NUMBER
27     )
28     VALUES (
```

```
29          patient_seq_new.NEXTVAL, p_patient_fname, p_patient_lname,
            ↪  p_blood_type, p_phone, p_email, p_gender, p_birthday,
            ↪  p_policy_number
30      )
31      RETURNING IDPATIENT INTO v_idpatient;
32
33      INSERT INTO Hospital.MEDICAL_HISTORY (
34          RECORD_ID, CONDITION, RECORD_DATE, IDPATIENT
35      )
36      VALUES (
37          patient_seq_new.NEXTVAL, p_condition, p_record_date, v_idpatient
38      );
39
40      INSERT INTO Hospital.INSURANCE (
41          POLICY_NUMBER, PROVIDER, INSURANCE_PLAN, CO_PAY, COVERAGE, MATERNITY,
            ↪  DENTAL, OPTICAL
42      )
43      VALUES (
44          p_policy_number, p_provider, p_insurance_plan, p_co_pay, p_coverage,
            ↪  p_maternity, p_dental, p_optical
45      );
46
47      INSERT INTO Hospital.EMERGENCY_CONTACT (
48          CONTACT_NAME, PHONE, RELATION, IDPATIENT
49      )
50      VALUES (
51          p_contact_name, p_contact_phone, p_contact_relation, v_idpatient
52      );
53
54      DBMS_OUTPUT.PUT_LINE('Patient and related records inserted
        ↪  successfully.');
55  EXCEPTION
56      WHEN OTHERS THEN
57          DBMS_OUTPUT.PUT_LINE('Error: '  SQLERRM);
58  END;
```

No segundo bloco, criamos a Procedure *insert_patient*, que aceita vários parâmetros como o primeiro nome do paciente, apelido, tipo de sangue, email, telefone, género, número de apólice, data

de nascimento e vários outro relacionado ao Paciente. Dentro da Procedure, usamos o comando *INSERT INTO* para adicionar um novo registo na tabela *Patient*, *Medical_History*, *Insurance* e *Emergency_Contact*, utilizando *patient_seq_new.NEXTVAL* para definir o *IDPATIENT* automaticamente. A data de nascimento é convertida para o formato de data apropriado usando *TO_DATE*. Após a inserção, uma mensagem de sucesso é exibida com *DBMS_OUTPUT.PUT_LINE*. Caso ocorra algum erro, a exceção é capturada e uma mensagem de erro é exibida com os detalhes.

Esta abordagem combinada de Queries, Functions e Procedures proporciona uma exploração robusta e abrangente do sistema hospitalar, garantindo que todas as informações necessárias estejam acessíveis e possam ser manipuladas conforme necessário para suportar a gestão eficiente e eficaz.

3.1.4 Triggers

Finalmente, os **Triggers** em Oracle SQL são um tipo especial de procedimento armazenado que é automaticamente executado (ou "disparado") pelo Oracle Database em resposta a certos eventos em uma tabela ou visão. Os triggers são usados para aplicar regras de negócios de forma automática, manter a integridade dos dados, realizar auditoria de alterações, e implementar outras funcionalidades automáticas que não podem ser facilmente realizadas apenas com comandos SQL simples. Um exemplo de um Trigger implementado é apresentado abaixo:

```
1  CREATE TABLE Hospital.New_Patient_Requests (
2      request_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
3      patient_fname VARCHAR2(45),
4      patient_lname VARCHAR2(45),
5      blood_type    VARCHAR2(3),
6      phone         VARCHAR2(12),
7      email         VARCHAR2(50),
8      gender        VARCHAR2(10),
9      birthday      DATE,
10     policy_number VARCHAR2(45),
11     condition     VARCHAR2(45),
12     record_date   DATE,
13     contact_name  VARCHAR2(45),
14     contact_phone VARCHAR2(30),
15     contact_relation VARCHAR2(45),
16     provider      VARCHAR2(45),
17     insurance_plan VARCHAR2(45),
18     co_pay        NUMBER,
19     coverage      VARCHAR2(20),
20     maternity     CHAR(1),
```

```
    dental        CHAR(1),
    optical       CHAR(1)
);
```

```
CREATE OR REPLACE TRIGGER trg_insert_patient
AFTER INSERT ON Hospital.New_Patient_Requests
FOR EACH ROW
BEGIN
    insert_patient(
        :NEW.patient_fname,
        :NEW.patient_lname,
        :NEW.blood_type,
        :NEW.phone,
        :NEW.email,
        :NEW.gender,
        :NEW.birthday,
        :NEW.policy_number,
        :NEW.condition,
        :NEW.record_date,
        :NEW.contact_name,
        :NEW.contact_phone,
        :NEW.contact_relation,
        :NEW.provider,
        :NEW.insurance_plan,
        :NEW.co_pay,
        :NEW.coverage,
        :NEW.maternity,
        :NEW.dental,
        :NEW.optical
    );
END;
```

O trigger *trg_insert_patient* foi criado para automatizar a inserção de um novo paciente e seus registos relacionados (como historial médico, seguro e contacto de emergência) na base de dados do hospital. Ele é disparado automaticamente após a inserção de um novo pedido de paciente na tabela *Hospital.New_Patient_Requests*.

No primeiro bloco, criamos a tabela *Hospital.New_Patient_Requests* que é definida para armazenar pedidos de novos pacientes. Esta tabela contém informações detalhadas sobre o paciente,

incluindo nome, tipo sanguíneo, contacto, seguro, entre outros. Cada coluna da tabela armazena uma informação específica sobre o paciente.

No segundo bloco, o trigger é definido para ser disparado após uma inserção (*AFTER INSERT*) na tabela *Hospital.New_Patient_Requests*. Para cada linha inserida (*FOR EACH ROW*), o trigger chama o procedimento *insert_patient* previamente mencionado e descrito com os valores dos novos registos inseridos.

É importante referir que os triggers oferecem várias vantagens no contexto dos procedimentos de inserção, atualização e remoção de dados específicos, neste caso na automatização da inserção de um novo paciente e seus registos associados. Algumas das principais vantagens prendem-se com a **Redução de Erros Manuais** que, ao automatizar a inserção dos dados do paciente e os seus registos relacionados, o trigger elimina a possibilidade de erros humanos que poderiam ocorrer durante a inserção manual de dados em múltiplas tabelas. Outra das principais vantages consiste na **Consistência de Dados** pois o trigger garante que todas as informações relacionadas ao paciente sejam inseridas de forma consistente e simultânea nas tabelas apropriadas (*Hospital.PATIENT*, *Hospital.MEDICAL_HISTORY*, *Hospital.INSURANCE*, *Hospital.EMERGENCY_CONTACT*), algo que é fundamental numa Base de Dados Relacional.

3.1.5 Queries, Functions, Procedures e Triggers implementados

De seguida, iremos apresentar todas as explorações implementadas, bem como uma explicação detalhada de cada uma das ferramentas implementadas, para as três visões mencionadas acima, bem como para a visão global da Base de Dados que engloba sempre duas ou mais visões.

Visão Pacientes

Na visão dos pacientes, o nosso foco foi extrair as informações necessárias para o funcionamento de um hospital real. Extraímos todos os dados relacionados a pacientes específicos, além de tratar da inserção, atualização e remoção dessas informações na base de dados.

No processo de inserção, desenvolvemos um procedimento que insere informações de um novo paciente em todas as quatro tabelas da base de dados. Consideramos que não faria sentido inserir os dados em apenas uma tabela, pois isso não reflete a realidade de um hospital.

Para a atualização, permitimos que qualquer tabela relacionada a pacientes possa ser atualizada. Por exemplo, pode ser necessário atualizar apenas o contato de emergência de um paciente ou realizar uma atualização no seu histórico médico.

No que diz respeito à remoção, criamos um procedimento que remove todas as informações do paciente de todas as tabelas relevantes. Além disso, o procedimento redefine o ID do paciente na tabela de Episódios para zero. Optamos por não excluir o episódio, pois a preservação dos dados históricos é fundamental numa base de dados relacional.

Associados aos procedimentos de Inserção, Atualização e Remoção, foram desenvolvidos Triggers

para que se tenha conhecimento de todos o tipo de alterações aos dados da Base de Dados, para garantir a consistência dos dados.

AllInfoPatient: Função que retorna todas as informações sobre um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registos de pacientes e retornar cada linha como um objeto PatientRow.

AllInfoMedicalHistory: Função que retorna todos os registos do historial médico para um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registos de histórico médico e retornar cada linha como um objeto MedicalHistoryRow.

AllInfoInsurance: Função que retorna todas as informações sobre o seguro de um paciente específico dado o número da apólice. A função usa uma tabela pipelined para iterar sobre os registros de seguro e retornar cada linha como um objeto InsuranceRow.

AllInfoEmergencyContact: Função que tetorna todas as informações sobre os contatos de emergência de um paciente específico, dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de contatos de emergência e retornar cada linha como um objeto EmergencyContactRow.

AllInfoPatient: Função que Retorna todas as informações combinadas de um paciente específico, dado o ID do paciente. A função junta dados de várias tabelas (paciente, histórico médico, seguro e contato de emergência) e retorna cada linha como um objeto PatientAllInfoRow.

AllInfoPatientByBloodType: Função que retorna todas as informações dos pacientes que possuem um determinado grupo sanguíneo. A função usa uma tabela pipelined para iterar sobre os registros de pacientes e retornar cada linha como um objeto PatientRow.

AllInfoPatientByGender: Função que retorna todas as informações dos pacientes que possuem um determinado gênero. A função usa uma tabela pipelined para iterar sobre os registros de pacientes e retornar cada linha como um objeto PatientRow.

GetPatientsWithCondition: Função que retorna todas as informações dos pacientes que possuem uma condição médica específica. A função usa uma tabela pipelined para iterar sobre os registros de pacientes e histórico médico, retornando cada linha como um objeto PatientRow.

ListAllRelationsProc: Procedimento que lista todos os tipos de relações de contato de emergência distintos na tabela de contatos de emergência.

ListAllProvidersProc: Procedimento que lista todos os providenciadores de seguro distintos na tabela de seguros.

ListAllInsurancePlansProc: Procedimento que lista todos os planos de seguro distintos na tabela de seguros.

ListAllCoveragesProc: Procedimento que lista todos os tipos de cobertura distintos na tabela de seguros.

ListAllConditionsProc: Procedimento que lista todas as condições médicas distintas na tabela de histórico médico.

ListAllBloodTypesProc: Procedimento que lista todos os tipos de sangue distintos na tabela de pacientes.

ListPatientsByMedicalHistoryDate: Função que retorna todos os pacientes que possuem registos de historial médico em uma data específica.

ListPatientsByInsurancePlan: Função que retorna todos os pacientes que possuem um plano de seguro específico.

ListPatientsByCoverage: Função que retorna todos os pacientes que possuem uma cobertura específica.

ListPatientsByAgeRange: Função que retorna todos os pacientes dentro de uma faixa etária específica.

ListPatientsWithMaternityCoverageProc: Procedimento que retorna todos os pacientes que possuem cobertura de maternidade no seu plano de seguro.

ListPatientsWithDentalCoverage: Função que retorna todos os pacientes que possuem cobertura dental no seu plano de seguro.

ListPatientsWithOpticalCoverageProc: Procedimento que retorna todos os pacientes que possuem cobertura óptica no seu plano de seguro.

GetPatientCountPerBloodTypeProc: Procedimento que conta o número de pacientes para cada tipo sanguíneo distinto na tabela de pacientes.

GetPatientCountPerConditionProc: Procedimento que conta o número de pacientes para cada condição médica distinta na tabela de historial médico.

insert_patient: Procedimento que insere um novo paciente e os registos relacionados de historial médico, seguro e contacto de emergência no banco de dados.

trg_insert_patient: Trigger que chama o procedimento *insert_patient* após a inserção de um novo pedido de paciente na tabela *New_Patient_Requests*.

update_patient: Procedimento que atualiza os detalhes de um paciente na tabela de pacientes.

trg_update_patient: Trigger que chama o procedimento *update_patient* antes de uma atualização na tabela de pacientes.

update_medical_history: Procedimento que atualiza os detalhes do historial médico de um paciente.

trg_update_medical_history: Trigger que chama o procedimento *update_medical_history* antes de uma atualização na tabela de historial médico.

update_insurance: Procedimento que atualiza os detalhes do seguro de um paciente.

trg_update_insurance: Trigger que chama o procedimento *update_insurance* antes de uma atualização na tabela de seguros.

update_emergency_contact: Procedimento que atualiza os detalhes do contacto de emergência de um paciente.

trg_update_emergency_contact: Trigger que chama o procedimento *update_emergency_contact* antes de uma atualização na tabela de contactos de emergência.

delete_patient_and_related: Procedimento que elimina um paciente e todos os registos relacionados de historial médico, seguro e contacto de emergência. Também define o ID do paciente para 0 na tabela de episódios.

trg_delete_patient_and_related: Trigger que chama o procedimento *delete_patient_and_related* antes de uma eliminação na tabela de pacientes.™

Visão Staff

Na visão do staff, o nosso foco foi extrair as informações necessárias para o funcionamento de um hospital real. Extraímos todos os dados relacionados aos funcionários, além de tratar da inserção, atualização e remoção dessas informações na base de dados.

No processo de inserção, desenvolvemos procedimentos que permitem a inclusão de novos membros do staff nas tabelas relevantes da base de dados. Consideramos que é essencial garantir que todas as informações importantes sobre um novo funcionário estejam devidamente registadas, refletindo a realidade operacional de um hospital.

Para a atualização, permitimos que qualquer tabela relacionada ao staff possa ser atualizada. Por exemplo, pode ser necessário atualizar apenas o contacto de um funcionário, ajustar suas qualificações ou alterar sua designação. Nosso objetivo foi garantir que todas as informações possam ser mantidas precisas e atualizadas de maneira eficiente.

No que diz respeito à remoção, criamos procedimentos que removem todas as informações de um funcionário das tabelas relevantes. Além disso, garantimos que quaisquer referências a esse funcionário em outras tabelas, mais concretamente o ID, seja redefinido para 0 ao invés da remoção. Optamos por não excluir dados históricos, como registos de atendimento ou episódios, para preservar a integridade e a continuidade dos dados históricos.

Tal como na visão acima, associados aos procedimentos de Inserção, Atualização e Remoção, foram desenvolvidos Triggers para que se tenha conhecimento de todos o tipo de alterações aos dados da Base de Dados, para garantir a consistência dos dados.

AllInfoStaff: Função que retorna todas as informações sobre um funcionário específico dado o ID do funcionário. A função usa uma tabela pipelined para iterar sobre os registos de funcionários e retornar cada linha como um objeto StaffRow.

AllInfoDepartment: Função que retorna todas as informações sobre um departamento específico dado o ID do funcionário. A função usa uma tabela pipelined para iterar sobre os registos de departamentos e retornar cada linha como um objeto DepartmentRow.

AllInfoNurse: Função que retorna todas as informações sobre uma enfermeira específica dado o ID do funcionário. A função usa uma tabela pipelined para iterar sobre os registos de enfermeiras e retornar cada linha como um objeto NurseRow.

AllInfoDoctor: Função que retorna todas as informações sobre um médico específico dado o ID do médico. A função usa uma tabela pipelined para iterar sobre os registos de médicos e retornar cada linha como um objeto DoctorRow.

AllInfoTechnician: Função que retorna todas as informações sobre um técnico específico dado o ID do funcionário. A função usa uma tabela pipelined para iterar sobre os registos de técnicos e retornar cada linha como um objeto TechnicianRow.

AllInfoStaffByDateJoining: Função que retorna todas as informações sobre funcionários que se juntaram numa data específica. A função usa uma tabela pipelined para iterar sobre os registos de funcionários e retornar cada linha como um objeto StaffRow.

AllInfoStaffByDateSeperation: Função que retorna todas as informações sobre funcionários que se separaram numa data específica. A função usa uma tabela pipelined para iterar sobre os registos de funcionários e retornar cada linha como um objeto StaffRow.

AllInfoStaffByStatus: Função que retorna todas as informações sobre funcionários com um status ativo ou inativo específico. A função usa uma tabela pipelined para iterar sobre os registos de funcionários e retornar cada linha como um objeto StaffRow.

GetDoctorQualifications: Função que retorna todas as qualificações de um médico específico dado o ID do médico. A função usa uma tabela pipelined para iterar sobre os registos de qualificações e retornar cada linha como um objeto QualificationRow.

GetEmployeeCountPerDepartmentProc: Procedimento que conta o número de funcionários por departamento e retorna os resultados em uma coleção de objetos DepartmentEmployeeCountRow.

GetNurseCountPerDepartmentProc: Procedimento que conta o número de enfermeiras por departamento e retorna os resultados em uma coleção de objetos DepartmentNurseCountRow.

GetDoctorCountPerDepartmentProc: Procedimento que conta o número de médicos por departamento e retorna os resultados em uma coleção de objetos DepartmentDoctorCountRow.

GetTechniciansCountPerDepartmentProc: Procedimento que conta o número de técnicos por departamento e retorna os resultados em uma coleção de objetos DepartmentTechniciansCountRow.

insert_staff_and_nurse: Procedimento que insere um novo funcionário que é uma enfermeira

e atualiza a contagem de funcionários no departamento correspondente.

trg_insert_staff_and_nurse: Trigger que chama o procedimento *insert_staff_and_nurse* após a inserção de um novo pedido de enfermeira na tabela *New_Staff_Nurse_Requests*.

insert_staff_and_doctor: Procedimento que insere um novo funcionário que é um médico e atualiza a contagem de funcionários no departamento correspondente.

trg_insert_staff_and_doctor: Trigger que chama o procedimento *insert_staff_and_doctor* após a inserção de um novo pedido de médico na tabela *New_Staff_Doctor_Requests*.

insert_staff_and_technician: Procedimento que insere um novo funcionário que é um técnico e atualiza a contagem de funcionários no departamento correspondente.

trg_insert_staff_and_technician: Trigger que chama o procedimento *insert_staff_and_technician* após a inserção de um novo pedido de técnico na tabela *New_Staff_Technician_Requests*.

insert_hospital_department: Procedimento que insere um novo departamento na tabela Hospital.DEPARTMENT.

trg_insert_department: Trigger que chama o procedimento *insert_hospital_department* após a inserção de um novo pedido de departamento na tabela *New_Department_Requests*.

update_staff: Procedimento que atualiza as informações de um funcionário e, se o departamento mudar, atualiza as contagens de funcionários nos departamentos antigo e novo.

trg_update_staff: Trigger que chama o procedimento *update_staff* antes de uma atualização na tabela Hospital.STAFF.

update_doctor: Procedimento que atualiza as qualificações de um médico.

trg_update_doctor: Trigger que chama o procedimento *update_doctor* antes de uma atualização na tabela *Hospital.DOCTOR*.

update_department: Procedimento que atualiza as informações de um departamento.

trg_update_department: Trigger que chama o procedimento *update_department* antes de uma atualização na tabela *Hospital.DEPARTMENT*.

delete_staff_and_nurse: Procedimento que elimina um funcionário que é uma enfermeira, atualiza a contagem de funcionários no departamento correspondente e redefine o enfermeiro responsável na tabela *Hospital.HOSPITALIZATION*.

trg_delete_staff_and_nurse: Trigger que chama o procedimento *delete_staff_and_nurse* antes de uma eliminação na tabela Hospital.STAFF.

delete_staff_and_doctor: Procedimento que elimina um funcionário que é um médico, atualiza a contagem de funcionários no departamento correspondente e redefine o ID do médico na tabela *Hospital.APPOINTMENT*.

trg_delete_staff_and_doctor: Trigger que chama o procedimento *delete_staff_and_doctor* antes de uma eliminação na tabela *Hospital.STAFF*.

delete_staff_and_technician: Procedimento que elimina um funcionário que é um técnico, atualiza a contagem de funcionários no departamento correspondente e redefine o ID do técnico na tabela *Hospital.LAB_SCREENING*.

trg_delete_staff_and_technician: Trigger que chama o procedimento *delete_staff_and_technician* antes de uma eliminação na tabela *Hospital.STAFF*.

DeleteDepartment: Procedimento que elimina um departamento e atualiza os funcionários para definir o ID do departamento como 0.

trg_delete_department: Trigger que chama o procedimento DeleteDepartment antes de uma eliminação na tabela *Hospital.DEPARTMENT*.

Visão Episodes

Na visão dos Episódios, o nosso foco foi a extração de informações abrangentes sobre os diferentes episódios médicos registados no hospital. Desenhamos queries que permitem listar todas as informações detalhadas sobre episódios específicos, incluindo consultas, hospitalizações, prescrições e faturas associadas. Com estas queries, podemos obter uma visão completa e detalhada de cada episódio, garantindo a integração e acessibilidade dos dados críticos para a gestão hospitalar.

Criamos funções que retornam informações sobre episódios médicos para pacientes específicos, tipos de condições, médicos responsáveis, e técnicos de laboratório. Além disso, desenvolvemos funções para listar detalhes financeiros associados aos episódios, como faturas e custos totais. Estas funções utilizam tabelas pipelined para iterar sobre os registos e retornar cada linha como um objeto, facilitando a manipulação e análise dos dados.

Para garantir a integridade e atualização dos dados, incluímos triggers que invocam procedimentos específicos após operações de inserção, atualização e exclusão nas tabelas relacionadas aos episódios. Estes procedimentos asseguram que todas as informações associadas sejam corretamente mantidas e atualizadas, refletindo qualquer mudança feita nos registos de episódios.

A nossa abordagem foi pensada para proporcionar uma visão holística dos episódios, permitindo uma gestão eficiente e detalhada dos dados médicos, essenciais para a operação diária de um hospital. Através destas queries e procedimentos, conseguimos não apenas listar e consultar informações, mas também manter a base de dados consistente e atualizada, apoiando a tomada de decisões e o acompanhamento preciso dos pacientes e seus tratamentos.

AllInfoMedicine: Função que retorna todas as informações sobre um medicamento específico dado o ID do medicamento. A função usa uma tabela pipelined para iterar sobre os registos de medicamentos e retornar cada linha como um objeto *MedicineRow*.

AllInfoPrescription: Função que retorna todas as informações sobre uma prescrição específica

dado o ID da prescrição. A função usa uma tabela pipelined para iterar sobre os registos de prescrições e retornar cada linha como um objeto *PrescriptionRow*.

AllInfoEpisode: Função que retorna todas as informações sobre um episódio específico dado o ID do episódio. A função usa uma tabela pipelined para iterar sobre os registos de episódios e retornar cada linha como um objeto *EpisodeRow*.

AllInfoEpisode: Função que retorna todas as informações sobre um episódio específico, incluindo consultas, hospitalizações, prescrições e facturas associadas, dado o ID do episódio. A função usa uma tabela pipelined para iterar sobre os registos de episódios e retornar cada linha como um objeto *EpisodeRow*.

AllInfoBill: Função que retorna todas as informações sobre uma fatura específica dado o ID da fatura. A função usa uma tabela pipelined para iterar sobre os registos de faturas e retornar cada linha como um objeto *BillRow*.

AllInfoRoom: Função que retorna todas as informações sobre um quarto específico dado o ID do quarto. A função usa uma tabela pipelined para iterar sobre os registos de quartos e retornar cada linha como um objeto *RoomRow*.

AllInfoHospitalization: Função que retorna todas as informações sobre uma hospitalização específica dado o ID do episódio. A função usa uma tabela pipelined para iterar sobre os registos de hospitalizações e retornar cada linha como um objeto *HospitalizationRow*.

AllInfoLabScreening: Função que retorna todas as informações sobre uma triagem laboratorial específica dado o ID da triagem. A função usa uma tabela pipelined para iterar sobre os registos de triagens laboratoriais e retornar cada linha como um objeto *LabScreeningRow*.

GetAllEpisodeInfo: Função que retorna todas as informações detalhadas sobre um episódio específico, incluindo consultas, hospitalizações, prescrições e faturas associadas, dado o ID do episódio. A função usa uma tabela pipelined para iterar sobre os registos de episódios e retornar cada linha como um objeto *EpisodeInfoRow*.

ListRoomsByTypeProc: Procedimento que lista todos os quartos ordenados pelo tipo. Usa uma tabela de objetos *RoomTypeRow* para armazenar e retornar as informações.

ListRoomOccupationsByDateRange: Função que lista todas as ocupações de quartos num intervalo de datas específico. Usa uma tabela pipelined para retornar os registos como objetos *RoomOccupationRow*.

ListCurrentlyOccupiedRoomsProc: Procedimento que lista todos os quartos atualmente ocupados. Usa uma tabela de objetos *OccupiedRoomRow* para armazenar e retornar as informações.

ListDistinctRoomTypesAndCostsProc: Procedimento que lista todos os tipos de quartos distintos e os respetivos custos, ordenados pelo custo. Usa uma tabela de objetos *DistinctRoomTypeRow* para armazenar e retornar as informações.

ListHospitalizationsByDateRange: Função que lista todas as hospitalizações num intervalo de datas específico. Usa uma tabela pipelined para retornar os registos como objetos *HospitalizationByDateRow*.

ListHospitalizationsByRoomType: Função que lista todas as hospitalizações para um tipo de quarto específico. Usa uma tabela pipelined para retornar os registos como objetos *HospitalizationByRoomTypeRow*.

GetTotalAppointments: Função que retorna o número total de consultas na tabela *Hospital.Appointment*.

GetTotalBillingForEpisode: Função que calcula o valor total de faturação para um determinado episódio. A função retorna uma string com o detalhe dos custos de quarto, testes, outras taxas e o total.

ListBillsByPaymentStatus: Função que lista todas as faturas com um status de pagamento específico. Usa uma tabela pipelined para retornar os registos como objetos *BillRowNew*.

GetTotalCostByRegisteredDate: Função que calcula o custo total das faturas registadas entre duas datas específicas.

GetTotalCostOfAllBills: Função que calcula o custo total de todas as faturas na tabela *Hospital.BILL*.

GetLabScreeningsByEpisode: Função que retorna todas as triagens laboratoriais para um episódio específico. Usa uma tabela pipelined para retornar os registos como objetos *LabScreeningRowNew*.

GetMedicinesAndPrescriptionsByEpisode: Função que retorna todas as prescrições e respetivos medicamentos para um episódio específico. Usa uma tabela pipelined para retornar os registos como objetos *MedicinePrescriptionRow*.

GetBillInfoByPatient: Função que retorna todas as informações de faturas para um paciente específico e o custo total acumulado. Usa uma tabela pipelined para retornar os registos como objetos *BillInfoRow*.

ListHospitalizationsOrderedByCost: Procedimento que lista todas as hospitalizações ordenadas pelo custo total associado. Usa uma tabela de objetos *HospitalizationWithCostRow* para armazenar e retornar as informações.

trg_insert_episode_and_related: Trigger que chama o procedimento *insert_episode* após uma inserção na tabela *Hospital.New_Episode_Requests*.

trg_insert_room: Trigger que chama o procedimento *insert_room* após uma inserção na tabela *Hospital.New_Room_Requests*.

trg_insert_medicine: Trigger que chama o procedimento *insert_medicine* após uma inserção

na tabela *Hospital.New_Medicine_Requests*.

trg_update_bill: Trigger que chama o procedimento *update_bill* antes de uma atualização na tabela *Hospital.BILL*.

trg_update_lab_screening: Trigger que chama o procedimento *update_lab_screening* antes de uma atualização na tabela *Hospital.LAB_SCREENING*.

trg_update_appointment: Trigger que chama o procedimento *update_appointment* antes de uma atualização na tabela *Hospital.APPOINTMENT*.

trg_update_hospitalization: Trigger que chama o procedimento *update_hospitalization* antes de uma atualização na tabela *Hospital.HOSPITALIZATION*.

trg_update_room: Trigger que chama o procedimento *update_room* antes de uma atualização na tabela *Hospital.ROOM*.

trg_update_prescription: Trigger que chama o procedimento *update_prescription* antes de uma atualização na tabela *Hospital.PRESCRIPTION*.

trg_update_medicine: Trigger que chama o procedimento *update_medicine* antes de uma atualização na tabela *Hospital.MEDICINE*.

trg_delete_episode: Trigger que chama o procedimento *delete_episode_and_update_related* antes de uma eliminação na tabela *Hospital.EPISODE*.

trg_delete_bill: Trigger que chama o procedimento *delete_bill* antes de uma eliminação na tabela *Hospital.BILL*.

trg_delete_lab_screening: Trigger que chama o procedimento *delete_lab_screening* antes de uma eliminação na tabela *Hospital.LAB_SCREENING*.

trg_delete_appointment: Trigger que chama o procedimento *delete_appointment* antes de uma eliminação na tabela *Hospital.APPOINTMENT*.

trg_delete_hospitalization: Trigger que chama o procedimento *delete_hospitalization* antes de uma eliminação na tabela *Hospital.HOSPITALIZATION*.

trg_delete_room: Trigger que chama o procedimento *delete_room* antes de uma eliminação na tabela *Hospital.ROOM*.

trg_delete_prescription: Trigger que chama o procedimento *delete_prescription* antes de uma eliminação na tabela *Hospital.PRESCRIPTION*.

trg_delete_medicine: Trigger que chama o procedimento *delete_medicine* antes de uma eliminação na tabela *Hospital.MEDICINE*.

Visão Global

Na visão global, o nosso objetivo foi integrar e consolidar informações de diversas áreas do hospital, proporcionando uma visão holística e abrangente dos dados médicos e operacionais. Desenvolvemos queries que abrangem múltiplas visões, facilitando a análise e a gestão dos dados através de uma abordagem integrada.

As funções e procedimentos globais permitem a extração de informações que cruzam diferentes domínios, como pacientes, staff, episódios médicos, hospitalizações, prescrições, exames laboratoriais e faturas. Estas queries foram desenhadas para garantir que os dados sejam consistentes, acessíveis e úteis para a tomada de decisões estratégicas e operacionais.

Criamos funções que retornam informações detalhadas sobre episódios médicos, incluindo consultas, hospitalizações, prescrições e faturas associadas, proporcionando uma visão completa de cada caso clínico. Além disso, desenvolvemos funções para listar custos totais associados a episódios, calcular faturações, e obter detalhes de triagens laboratoriais e prescrições por episódio. Estas funções utilizam tabelas pipelined para iterar sobre os registos e retornar cada linha como um objeto, facilitando a manipulação e análise dos dados.

A nossa abordagem global foi pensada para fornecer uma plataforma robusta e integrada de gestão de dados hospitalares, permitindo uma visão abrangente e detalhada das operações e tratamentos médicos. Com estas queries e procedimentos, conseguimos melhorar a eficiência da gestão hospitalar, apoiar a tomada de decisões informadas, e garantir que todas as informações críticas estejam disponíveis e atualizadas para todos os stakeholders envolvidos.

PrescriptionsForPatient: Função que retorna todas as prescrições para um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de prescrições e retornar cada linha como um objeto *PrescriptionRowNew*.

ListPatientsInRoom: Função que retorna os pacientes alocados a um quarto específico dado o ID do quarto. A função usa uma tabela pipelined para iterar sobre os registros de pacientes e retornar cada linha como um objeto *RoomPatientsRow*.

ListHospitalizationsForPatient: Função que retorna todas as hospitalizações de um determinado paciente dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de hospitalização e retornar cada linha como um objeto *HospitalizationRow*.

ListHospitalizationsByNurse: Função que retorna hospitalizações por enfermeira responsável dado o ID da enfermeira. A função usa uma tabela pipelined para iterar sobre os registros de hospitalização e retornar cada linha como um objeto *HospitalizationByNurseRow*.

ListEpisodesForPatient: Função que retorna todos os episódios médicos de um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de episódios e retornar cada linha como um objeto *EpisodeRowNew*.

ListEpisodesByCondition: Função que retorna episódios médicos por tipo de condição dado

o tipo da condição. A função usa uma tabela pipelined para iterar sobre os registros de episódios e retornar cada linha como um objeto *EpisodeByConditionRow*.

ListEpisodesByDoctor: Função que retorna todos os episódios médicos tratados por um médico específico dado o ID do médico. A função usa uma tabela pipelined para iterar sobre os registros de episódios e retornar cada linha como um objeto *EpisodeByDoctorRow*.

ListLabScreeningsByPatient: Função que retorna todos os exames laboratoriais para um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de exames laboratoriais e retornar cada linha como um objeto *LabScreeningRow*.

ListLabScreeningDetailsByTechnician: Função que retorna exames laboratoriais baseados no técnico responsável dado o ID do técnico. A função usa uma tabela pipelined para iterar sobre os registros de exames laboratoriais e retornar cada linha como um objeto *LabScreeningDetailsRow*.

ListBillDetailsByPatient: Função que retorna todas as faturas para um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de faturas e retornar cada linha como um objeto *BillDetailsRow*.

ListBillAndAppointmentDetailsByDoctor: Função que retorna todas as faturas emitidas por um médico específico dado o ID do médico. A função usa uma tabela pipelined para iterar sobre os registros de faturas e retornar cada linha como um objeto *BillAndAppointmentRow*.

ListAppointmentDoctorDetailsByPatient: Função que retorna todas as consultas agendadas para um paciente específico dado o ID do paciente. A função usa uma tabela pipelined para iterar sobre os registros de consultas e retornar cada linha como um objeto *AppointmentDoctorRow*.

ListAppointmentDoctorStaffDetailsByDoctor: Função que retorna consultas baseadas no médico responsável dado o ID do médico. A função usa uma tabela pipelined para iterar sobre os registros de consultas e retornar cada linha como um objeto *AppointmentDoctorStaffRow*.

ListAppointmentEpisodePatientDetails: Função que retorna consultas agendadas para um médico específico (por dia) dado o ID do médico e a data da consulta. A função usa uma tabela pipelined para iterar sobre os registros de consultas e retornar cada linha como um objeto *AppointmentEpisodePatientRow*.

ListAppointmentsByDate: Função que retorna consultas por data específica. A função usa uma tabela pipelined para iterar sobre os registros de consultas e retornar cada linha como um objeto *AppointmentDatePatientRow*.

ListAppointmentsByDateTime: Função que retorna consultas por data e hora específica. A função usa uma tabela pipelined para iterar sobre os registros de consultas e retornar cada linha como um objeto *AppointmentDateTimePatientRow*.

ListAppointmentPatientInfoProc: Procedimento que lista todos os episódios e o respetivo paciente. O procedimento usa a coleta em massa (bulk collect) para armazenar os resultados em

uma tabela de objetos *AppointmentPatientInfoRow*.

GetDoctorWithMostAppointmentsProc: Procedimento que lista os médicos com mais consultas marcadas, incluindo informações detalhadas do paciente. O procedimento usa a coleta em massa (bulk collect) para armazenar os resultados em uma tabela de objetos *DoctorAppointmentCountRow*.

Capítulo 4

Base de dados documental - MongoDB

Para armazenar os nossos dados numa base de dados não relacional e orientada a documentos, recorremos ao **MongoDB**. Inicialmente, dedicamos bastante tempo à avaliação do esquema da base de dados relacional existente para decidir a melhor forma de representar os dados na nova estrutura orientada a documentos, aproveitando ao máximo as vantagens deste paradigma. Com base nesta análise e considerando as consultas futuras que serão realizadas, decidimos dividir a base de dados em três coleções principais:

1. **Pacientes:** Armazena informações sobre os pacientes, os seus contactos de emergência, histórico médico e informações do seguro.
2. **Staff:** Inclui dados sobre médicos, enfermeiros e técnicos, juntamente com as suas respetivas informações do departamento onde trabalham.
3. **Episódios:** Contém detalhes sobre os episódios de atendimento médico, incluindo consultas, hospitalizações, prescrições, faturas e exames laboratoriais.

Para a migração dos dados de Oracle para MongoDB utilizou-se o script `migrate_to_mongo.py` que efetua uma conexão com a base de dados Oracle, de forma a permitir a recolha dos dados. Após a obtenção dos dados da base de dados relacional, realizamos o devido tratamento e criação das coleções acima propostas e que serão abordadas nos capítulos seguintes. Para finalizar, realizamos uma segunda conexão, agora à base de dados MongoDB e carregamos as coleções resultantes para esta base de dados não relacional.

Este script foi criado para facilitar a implementação dos dados em MongoDB de forma a que seja possível uma transferência eficiente e estruturada dos dados sempre que necessário.

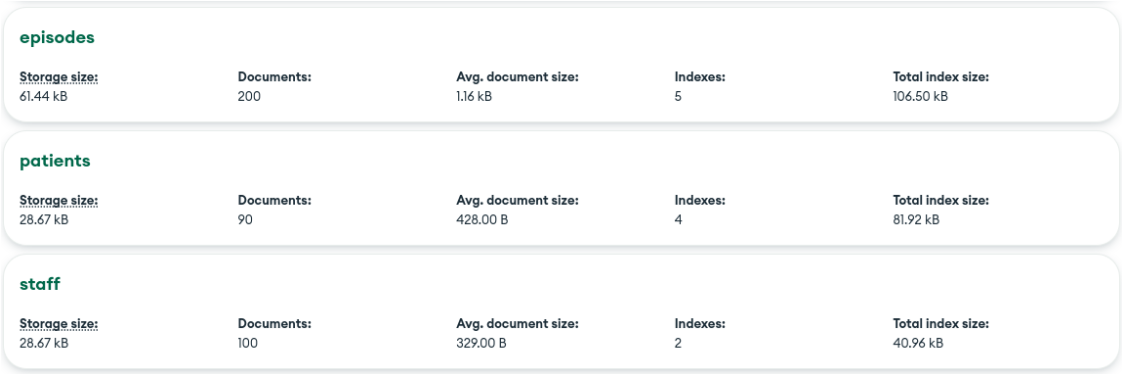


Figura 4.1: Coleções criadas na base de dados MongoDB

4.1 Coleção Pacientes

A coleção de Pacientes no MongoDB foi projetada para armazenar informações abrangentes sobre cada paciente, aproveitando a flexibilidade da estrutura de documentos JSON. Esta coleção inclui dados pessoais como nome, data de nascimento, tipo sanguíneo, contacto telefónico e e-mail. Além disso, armazena detalhes de seguros de saúde, contactos de emergência e histórico médico. No final um objeto paciente pertencente a esta coleção terá uma estrutura semelhante à seguinte:

```
{
  "_id": {"$oid": "6658a65bf6d1a33ea5c68337"},
  "id_patient": 2,
  "patient_fname": "Jane",
  "patient_lname": "Smith",
  "blood_type": "O-",
  "phone": "987-654-3210",
  "email": "jane.smith@example.com",
  "gender": "Female",
  "birthday": {"$date": "1990-03-20T00:00:00.000Z"},
  "insurance": {
    "policy_number": "POL002",
    "provider": "XYZ Insurance",
    "insurance_plan": "Premium Plan",
    "co_pay": 30,
    "coverage": "Partial Coverage",
    "maternity": false,
    "dental": true,
    "optical": true
  },
```

```
  "emergency_contact": [
    {
      "contact_name": "Jane Smith",
      "phone": "222-333-4444",
      "relation": "Mother"
    }
  ],
  "medical_history": [
    {
      "record_id": 2,
      "condition": "Allergy",
      "record_date": {"$date": "2023-03-05T00:00:00.000Z"}
    }
  ]
}
```

4.2 Coleção Staff

A coleção de Staff no MongoDB foi desenvolvida para gerir de forma eficiente os dados relacionados aos funcionários do sistema de saúde, incluindo médicos, enfermeiros e técnicos. Esta coleção armazena informações detalhadas sobre cada funcionário, como nome, data de admissão, endereço, e-mail e número de segurança social. Também inclui detalhes específicos sobre o departamento ao qual pertencem, suas qualificações e o estado de atividade atual. Na base esta coleção faz uso da semelhança entre as tabelas de *doctor*, *nurse* e *technician* agrupando-as com a tabela *staff* e *department*, utilizando o campo ***role*** para identificação do funcionário de saúde em questão (*DOCTOR*, *NURSE*, *TECHNICIAN*). O atributo ***date_of_separation*** apenas se encontra presente nos funcionários que já abandonaram um determinado departamento, quando tal não acontece este campo não é acrescentado ao objeto JSON. Em seguida é apresentado um exemplo da estrutura implementada para o caso de um médico que ainda se mantém ativo no departamento em questão:

```
{
  "_id": {"$oid": "6658a65bf6d1a33ea5c6838a"},
  "emp_id": 6,
  "emp_fname": "Lisa",
  "emp_lname": "Hayes",
  "date_joining": {"$date": "2023-05-10T00:00:00.000Z"},
  "email": "mprice@example.com",
  "address": "Trevorfurt, IN 02637\"",
```

```
    "ssn": 685569160,
    "is_active_status": true,
    "department": {
      "id_department": 1,
      "department_head": "John Smith",
      "department_name": "Cardiology_1"
    },
    "role": "DOCTOR",
    "qualifications": "PhD"
}
```

4.3 Coleção Episódios

A coleção de Episódios no MongoDB foi criada para armazenar informações detalhadas sobre os episódios de atendimento dos pacientes, incluindo consultas, hospitalizações, prescrições, faturas e exames laboratoriais. Como esta coleção serve como uma ligação essencial entre as coleções de Pacientes e Staff, foi necessário encontrar uma maneira eficiente de traduzir essas relações. Uma das abordagens consideradas foi a criação de redundância nas coleções, aproveitando a flexibilidade do paradigma não relacional para replicar dados e adicionar informações úteis diretamente na coleção de Episódios. No entanto, para evitar a redundância excessiva, optou-se por utilizar identificadores de objetos de outras coleções, funcionando de maneira similar às chaves estrangeiras no paradigma relacional. Esta abordagem permite manter a integridade referencial e facilita consultas mais eficientes e organizadas, garantindo, assim, uma gestão mais eficaz dos dados.

Para implementar esta abordagem, foi necessário armazenar em duas listas os IDs dos objetos de Pacientes e Staff que foram adicionados. Em Python, essa tarefa foi facilitada pelo uso da função `collection.insert_many()`. Esta função não apenas insere os elementos na coleção correspondente, mas também retorna um objeto contendo a lista de todos os IDs inseridos. Assim, antes de migrar os dados para a coleção de episódios, bastou obter duas listas contendo todos os IDs dos pacientes e membros do staff previamente inseridos. Com estas listas carregadas, foi possível estabelecer as relações necessárias entre as coleções de forma eficiente e precisa, garantindo que, por exemplo, cada episódio estivesse corretamente associado aos seus respetivos paciente e profissional de saúde. Se um indivíduo desta coleção fosse hospitalizado, recebesse prescrições de medicamentos e fosse submetido a exames médicos, a sua estrutura seria representada da seguinte forma na coleção:

```
{
    "_id": {"$oid": "66603d8cc0c3dd4601195502"},
    "id_episode": 175,
```

```
    "id_patient": {"$oid": "66603d8cc0c3dd460119543d"},
    "hospitalization": {
      "admission_date": {"$date": "2020-11-03T00:00:00.000Z"},
      "discharge_date": {"$date": "2020-11-04T00:00:00.000Z"},
      "responsible_nurse": {"$oid": "66603d8cc0c3dd46011954ca"},
      "room": {
        "id_room": 40,
        "room_type": "Executive",
        "room_cost": 420
      }
    },
    "prescriptions": [
      {
        "id_prescription": 169,
        "prescription_date": {"$date": "2020-11-03T00:00:00.000Z"},
        "dosage": 31,
        "medicine": {
          "id_medicine": 1,
          "m_name": "Paracetamol",
          "m_quantity": 50,
          "m_cost": 10
        }
      }
    ],
    "lab_screenings": [
      {
        "lab_id": 9,
        "test_cost": 93.19,
        "test_date": {"$date": "2020-11-03T00:00:00.000Z"},
        "id_technician": {"$oid": "66603d8cc0c3dd46011954f6"}
      }
    ]
  }
```


4.4 Triggers em MongoDB

Durante o processo de migração de dados da base de dados Oracle para MongoDB, foi necessário definir diversos *triggers* para otimizar a inserção de novos documentos e manter a integridade dos

dados. Na nossa base de dados, procurávamos evitar o uso manual de campos de identificação durante a inserção de novos documentos preservando os campos de identificação originais, provenientes da base de dados Oracle, e mantendo a utilização de campos de identificação do mesmo tipo na base de dados MongoDB (além da utilização de identificadores do tipo ObjectId para o campo '_id'). Para tal, utilizamos *triggers* responsáveis por gerar automaticamente esses identificadores, garantindo a integridade e a unicidade dos dados.

Para alcançar este objetivo, optamos pela criação de uma coleção auxiliar denominada '*counters*', responsável por armazenar o valor máximo de cada identificador inteiro existente, entre todos os valores já utilizados até ao momento. Cada documento desta coleção é utilizado para a geração do próximo valor de cada identificador único (similar às sequências em Oracle) durante a inserção de novos documentos ou substituição de documentos existentes nas restantes coleções da base de dados. Assim, na inserção de um novo documento, o contador correspondente, presente neste coleção '*counters*', é incrementado, de modo obter o próximo valor do identificador, de acordo com a coleção na qual foi inserido um novo documento e o identificador em questão.

Além dos *triggers* relacionados à criação de identificadores únicos, foi necessário adaptar um *trigger* existente na base de dados original Oracle para o ambiente MongoDB. Este *trigger* específico foi ajustado para garantir que sua funcionalidade fosse preservada após a migração, mantendo a consistência e a continuidade das operações automatizadas.

Em MongoDB, para a configuração e *deployment* destes triggers, decidimos utilizar um *cluster* criado na plataforma *MongoDB Atlas*. Assim, de modo a automatizar a criação de triggers nesta plataforma, desenvolvemos diversos *scripts* Python responsáveis pela criação, atualização, remoção e retoma (após uma falha no cluster, por exemplo) dos triggers na mesma, que utilizam pedidos *HTTP* a APIs disponibilizadas pelo MongoDB de modo a efetuar as diversas operações descritas.

4.4.1 Triggers para a adição de identificadores únicos

Na sequencia da criação destes *triggers* foi observada a existência de dois casos de geração de identificadores diferentes. Estes dois casos distintos dividiram a implementação em dois tipos principais: *triggers* responsáveis pela geração de valores para os identificadores de cada documento e, *triggers* responsáveis pela geração de valores para identificadores de objetos armazenados em listas/arrays presentes nos documentos das restantes coleções.

Assim sendo, para os campos 'id_patient', 'id_episode' e 'emp_id', das coleções 'patients', 'episodes' e 'staff, respetivamente, foram criados três *triggers* idênticos, acionados sempre que seja necessário gerar um novo identificador como valor de algum destes três campos, ou, caso o utilizador decida especificar o valor do identificador manualmente e possa, eventualmente, ser necessário atualizar o valor do contador correspondente ao campo em questão, para o valor do identificador inserido manualmente (caso este seja superior ao valor do contador). Deste modo, estes *triggers* são executados apenas nos casos em que existe a inserção de um novo documento em qualquer

uma das restantes coleções, sempre que um documento é substituído por um novo (*replace)* e este último não apresenta identificador inteiro ou, o valor do identificador é alterado, através de uma operação de *update* ou *replace*, para um valor superior ao armazenado no contador corresponde e, portanto, é necessário atualizar o valor do contador. Para obter este comportamento e evitar que os *triggers* fossem executados, por exemplo, quando são efetuados *updates* a outros campos de cada documento, foi necessário definir uma expressão `$match` no MongoDB Atlas.

Para o campo 'record_id' existente nos objetos armazenados na lista 'medical_history', presente nos documentos da coleção 'patients' e, para os campos 'id_bill', 'id_prescription' e 'lab_id' existentes nos objetos armazenados nas listas 'bills', 'prescriptions' e 'lab_screenings', respetivamente, da coleção 'episodes' foram criados quatro *triggers* idênticos, acionados sempre que seja necessário gerar um novo identificador como valor de algum destes quatro campos, ou, caso o utilizador decida especificar o valor do identificador manualmente e possa, eventualmente, ser necessário atualizar o valor do contador correspondente ao campo em questão, para o valor do identificador inserido manualmente (caso este seja superior ao valor do contador).

Deste modo, estes *triggers* são executados apenas nos casos em que existe a inserção de um novo documento em qualquer uma das restantes coleções, sempre que um documento é substituído por um novo (*replace)*, um novo objeto é inserido numa das listas referidas ou, o valor do identificador de um objeto é alterado, através de uma operação de *update*. Tal como nos triggers anteriores, para obter este comportamento e evitar que os *triggers* fossem executados, por exemplo, quando são efetuados *updates* a outros campos de cada documento, foi necessário definir mais uma expressão `$match` no MongoDB Atlas.

4.4.2 Trigger proveniente da base de dados Oracle

O *trigger* proveniente da base de dados Oracle em questão têm como objetivo principal gerar automaticamente uma fatura (*bill*) quando um paciente recebe uma alta hospitalar, ou seja, quando o campo `discharge_date` é adicionado ao objeto '`hospitalization`' de um documento da coleção '`episodes`' com valor diferente de `null`, ou então, o seu valor é atualizado de `null` para um valor diferente de `null`. Assim sendo, este *trigger* é, em primeiro lugar, apenas atualizado quando o documento em questão é atualizado (`update`) ou substituído (`replace`). Para obter o comportamento e evitar que este *trigger* fosse executado, por exemplo, quando são efetuados *updates* a outros campos de cada documento, utilizamos a seguinte expressão de `$match` no MongoDB Atlas, de modo a filtrar os casos em que não é pretendido gerar uma nova fatura.

```
1 {
2     "fullDocumentBeforeChange.hospitalization.discharge_date": null,
3     "fullDocument.hospitalization.discharge_date": {
4         "$ne": null
5     }
```

```
}
```

Tal como em Oracle, este *trigger* é responsável por monitorizar o valor do campo de alta hospitalar e, ao detetar uma atualização para um valor diferente de `null`, inicia uma série de processos para calcular os custos associados ao paciente durante sua estadia. Esses custos incluem taxas de quarto (`room_cost`), custos de exames laboratoriais (`test_cost`) e outros encargos médicos (`other_charges`). A soma total desses custos resulta na criação de uma nova fatura, armazenada na lista `bills`. O código desenvolvido para a implementação deste *trigger* é apresentado de seguida (a variável `service_name` presente no código abaixo é substituído pelo nome do serviço do MongoDB Atlas).

```
function stringify(object) {
    return JSON.stringify(object).replaceAll('"', "'").replaceAll(':', ': ').replaceAll(',', ', ');
}

function isIterable(value) {
    return Symbol.iterator in Object(value);
}

exports = async function(changeEvent) {
    try {
        const fullDocument = changeEvent.fullDocument;
        const docId = changeEvent.documentKey._id;

        const serviceName = '{{ service_name }}';
        const databaseName = changeEvent.ns.db;
        const collectionName = changeEvent.ns.coll;

        const episodesCollection = context.services.get(serviceName).db(databaseName).collection(collectionName);

        // Calculate the room cost for the associated hospitalization
        let roomCost = 0;
        const hospitalization = fullDocument.hospitalization;

        if (hospitalization) {
          const room = hospitalization.room;

```

```
27            if (room && room.room_cost) {
28              roomCost = room.room_cost;
29            }
30          }
31
32          // Calculate the test cost for the associated hospitalization
33          let testsCost = 0;
34          const lab_screenings = fullDocument.lab_screenings;
35
36          if (isIterable(lab_screenings)) {
37              for (test of lab_screenings) {
38                  if (test.test_cost) {
39                      testsCost += test.test_cost;
40                  }
41              }
42          }
43
44          // Calculate the other charges for prescriptions for the associated
            ↪ hospitalization
45          let prescriptionsCost = 0;
46          const prescriptions = fullDocument.prescriptions;
47
48          if (isIterable(prescriptions)) {
49              for (prescription of prescriptions) {
50                  const medicine = prescription.medicine;
51                  const dosage = prescription.dosage;
52
53                  if (medicine && dosage && medicine.m_cost) {
54                      prescriptionsCost += medicine.m_cost * dosage;
55                  }
56              }
57          }
58
59          // Calculate the total cost of the bill for the associated episode
60          const totalCost = roomCost + testsCost + prescriptionsCost;
61
62          // Insert the bill with the total cost for the associated episode
```

```
63          const newBill = {
64            room_cost: roomCost,
65            test_cost: testsCost,
66            other_charges: prescriptionsCost,
67            total: totalCost,
68            registered_at: new Date(),
69            payment_status: 'PENDING'
70          }
71
72          const billsListName = 'bills';
73          const documentQuery = { _id: docId };
74
75          await episodesCollection.updateOne(documentQuery, { $push: {
            ↪  [billsListName]: newBill } });
76
77          console.log(`Adicionada fatura ${stringify(newBill)} ao documento
            ↪  ${stringify(documentQuery)}, na lista '|\$|{billsListName}'.`);
78
79      } catch (err) {
80          console.error('Erro ao executar o trigger: ', err.message);
81      }
82  };
```

Este *trigger* garante que a transição do OracleDB para o MongoDB mantenha a mesma eficiência e precisão na criação de faturas, assegurando a continuidade das operações financeiras no ambiente hospitalar de uma forma otimizada.

4.5 Scripts para gestão dos triggers no MongoDB Atlas

Tal como referido, foram criados scripts `Python` responsáveis pela criação, atualização, remoção e retoma dos *triggers* na plataforma MongoDB Atlas. Assim sendo, os quatros scripts desenvolvidos são apresentados na seguinte lista.

- '`create_triggers.py`' - Script responsável pela criação de *triggers* na plataforma MongoDB Atlas. Os argumentos deste script são:
  - '`-pubk`' ou '`--public-key`' - Chave pública da organização existente no MongoDB Atlas a utilizar;

  - '`-privk`' ou '`--private-key`' - Chave privada da organização existente no MongoDB Atlas a utilizar;
  - '`-pn`' ou '`--project-name`' - Nome do projeto MongoDB Atlas a utilizar;
  - '`-cn`' ou '`--cluster-name`' - Nome do cluster MongoDB Atlas a utilizar;
  - '`-f`' ou '`--file`' - Caminho para o ficheiro JSON de configuração dos triggers a utilizar;

- '`update_triggers.py`' - Script responsável pela atualização de *triggers* já existentes na plataforma MongoDB Atlas. Os argumentos deste script são:
  - '`-pubk`' ou '`--public-key`' - Chave pública da organização existente no MongoDB Atlas a utilizar;
  - '`-privk`' ou '`--private-key`' - Chave privada da organização existente no MongoDB Atlas a utilizar;
  - '`-pn`' ou '`--project-name`' - Nome do projeto MongoDB Atlas a utilizar;
  - '`-cn`' ou '`--cluster-name`' - Nome do cluster MongoDB Atlas a utilizar;
  - '`-f`' ou '`--file`' - Caminho para o ficheiro JSON de configuração dos triggers a utilizar;

- '`delete_triggers.py`' - Script responsável pela remoção de *triggers* existentes na plataforma MongoDB Atlas. Os argumentos deste script são:
  - '`-pubk`' ou '`--public-key`' - Chave pública da organização existente no MongoDB Atlas a utilizar;
  - '`-privk`' ou '`--private-key`' - Chave privada da organização existente no MongoDB Atlas a utilizar;
  - '`-pn`' ou '`--project-name`' - Nome do projeto MongoDB Atlas a utilizar;
  - '`-f`' ou '`--file`' - Caminho para o ficheiro JSON de configuração dos triggers a utilizar;

- '`delete_triggers.py`' - Script responsável pela retoma (*resume*) de *triggers* suspensos na plataforma MongoDB Atlas. Os argumentos deste script são:
  - '`-pubk`' ou '`--public-key`' - Chave pública da organização existente no MongoDB Atlas a utilizar;
  - '`-privk`' ou '`--private-key`' - Chave privada da organização existente no MongoDB Atlas a utilizar;
  - '`-pn`' ou '`--project-name`' - Nome do projeto MongoDB Atlas a utilizar;
  - '`-f`' ou '`--file`' - Caminho para o ficheiro JSON de configuração dos triggers a utilizar;

4.6 Exploração da Base de Dados

Após a migração para o MongoDB, conforme explicado anteriormente, iniciámos a exploração desta base de dados orientada a documentos. Para isso, usamos duas abordagens distintas. No primeiro caso, para consultar os dados concretos, isto é, passar por exemplo um ID de episódio foram usadas **funções**. Por outro lado, caso a consulta fosse com o objetivo de obter dados de maneira mais geral como o custo total dos exames laboratoriais e/ou o número total desses exames foram usadas **expressões**.

Para obter informações detalhadas de um episódio específico, utilizámos uma função simples que aceita um argumento, neste caso, o *episodeId*. A função `getAllInfoByEpisodeId` foi implementada para buscar todos os documentos na coleção *episodes* onde o campo *id_episode* coincide com o *episodeId* fornecido. Abaixo encontra-se a implementação desta função:

```
function getAllInfoByEpisodeId(episodeId) {
    return db.episodes.find({
        id_episode: episodeId
    }).toArray();
}

getAllInfoByEpisodeId(89);
```

Esta função permite-nos recuperar rapidamente todas as informações associadas a um episódio específico, facilitando a análise detalhada e a verificação de dados individuais conforme necessário.

Outro exemplo seria:

```
function getPatientsByRoom(roomId) {
    return db.episodes.aggregate([
        {
            $match: { "hospitalization.room.id_room": roomId }
        },
        {
            $lookup: {
                from: "patients",
                localField: "id_patient",
                foreignField: "_id",
                as: "patient_info"
            }
        },
        {
```

```
            $unwind: "$patient_info"
        },
        {
            $project: {
                _id: 0,
                patient_id: "$patient_info._id",
                patient_fname: "$patient_info.patient_fname",
                patient_lname: "$patient_info.patient_lname",
                room: "$room"
            }
        }
    ]).toArray();
}

getPatientsByRoom(1)
```

Nesta expressão de agregação, utilizamos as seguintes etapas do pipeline de agregação:

- `$match`: Filtra os documentos da coleção `episodes` para incluir apenas aqueles onde o campo `hospitalization.room.id_room` coincide com o *roomId* fornecido.
- `$lookup`: Faz uma junção (*join*) com a coleção `patients` para obter informações adicionais sobre os pacientes, utilizando o campo `id_patient` como referência.
- `$unwind`: Decompõe o *array* `patient_info` para incluir as informações do paciente diretamente no documento.
- `$project`: Reformata o resultado final para incluir apenas os campos desejados: `patient_id`, `patient_fname`, `patient_lname` e `room`.

Esta função permite-nos obter uma lista detalhada de pacientes que estão internados num determinado quarto, proporcionando uma visão clara e organizada das informações dos pacientes.

Além disso, para obter *insights* mais abrangentes e sumarizados sobre os dados, utilizámos expressões agregadas. Um exemplo disso é a agregação que calcula o custo total dos exames laboratoriais (*lab_screenings*) e o número total desses exames. A agregação é realizada com a seguinte expressão:

```
db.episodes.aggregate([
    { $unwind: "$lab_screenings" },
    { $group: {
        _id: null,
```

```
        totalCost: { $sum: "$lab_screenings.test_cost" },
        totalCount: { $sum: 1 }
    }},
    { $project: {
        _id: 0,
        totalCost: 1,
        totalCount: 1
    }}
])
```

Nesta expressão, utilizamos várias etapas do pipeline de agregação: o `$unwind` decompõe o *array* `lab_screenings` em múltiplos documentos, cada um contendo um único elemento do *array*; o `$group` agrupa todos os documentos, somando os custos dos exames e contando o número total de exames; e o `$project` reformata o resultado para exibir apenas os campos `totalCost` e `totalCount`.

Outro exemplo seria:

```
db.episodes.aggregate([
        {
            $unwind: "$appointment"
        },
        {
            $group: {
                _id: "$appointment.id_doctor",
                totalAppointments: { $sum: 1 }
            }
        },
        {
            $sort: { totalAppointments: -1 }
        },
        {
            $lookup: {
                from: "staff",
                localField: "_id",
                foreignField: "_id",
                as: "doctor_info"
            }
        },
        {
            $unwind: "$doctor_info"
```

```
        },
        {
            $lookup: {
                from: "episodes",
                localField: "_id",
                foreignField: "appointment.id_doctor",
                as: "appointments"
            }
        },
        {
            $project: {
                _id: 0,
                doctor_id: "$_id",
                doctor_fname: "$doctor_info.emp_fname",
                doctor_lname: "$doctor_info.emp_lname",
                doctor_email: "$doctor_info.email",
                totalAppointments: 1,
            }
        }
    ])
```

Nesta expressão de agregação, utilizamos as seguintes etapas do pipeline de agregação:

- `$unwind`: Esta etapa decompõe o *array* `appointment` em múltiplos documentos, cada um contendo um único elemento do *array*.
- `$group`: Agrupa os documentos pelo campo `id_doctor` dentro de `appointment`, contando o número total de consultas (`totalAppointments`) para cada médico.
- `$sort`: Ordena os resultados pelo número total de consultas em ordem decrescente.
- `$lookup`: Faz uma junção (*join*) com a coleção `staff` para obter informações adicionais sobre o médico, utilizando o campo `_id` como referência.
- `$unwind`: Decompõe o *array* `doctor_info` para incluir as informações do médico diretamente no documento.
- `$lookup`: Faz outra junção (*join*) com a coleção `episodes` para obter todas as consultas relacionadas ao médico específico.
- `$project`: Reformata o resultado final para incluir apenas os campos desejados: `doctor_id`, `doctor_fname`, `doctor_lname`, `doctor_email` e `totalAppointments`.

É importante mencionar que a utilização de uma base de dados orientada a documentos, como o MongoDB, oferece várias vantagens para o bom desempenho da *expressão* fornecida. As vantagens do MongoDB para a execução da expressão fornecida são várias, especialmente no contexto da agregação de dados e das funcionalidades específicas que o MongoDB oferece. Vamos detalhar algumas dessas vantagens:

1. **Modelo de Dados Flexível:** MongoDB utiliza um modelo de dados baseado em documentos, que permite a representação de dados complexos e hierárquicos de forma mais natural e eficiente. No caso desta expressão, a capacidade de armazenar arrays e subdocumentos permite a estruturação dos dados de uma maneira que simplifica o processo de agregação.

2. **Pipeline de Agregação Poderosa:** A framework de agregação do MongoDB oferece um pipeline poderosa e flexível para processar e transformar dados. A expressão fornecida utiliza várias etapas do pipeline de agregação, como $unwind, $group, $sort, $lookup e $project, cada uma desempenhando uma função específica e eficiente no processamento dos dados.

3. **Desempenho e Escalabilidade:** MongoDB é projetado para ser escalável horizontalmente, o que significa que pode lidar com grandes volumes de dados e consultas complexas de maneira eficiente. A indexação adequada pode otimizar ainda mais a performance das consultas, especialmente aquelas que envolvem operações de agregação complexas.

Estas ferramentas e técnicas são essenciais para a exploração eficiente de bases de dados no MongoDB, permitindo-nos manipular grandes volumes de dados e extrair informações relevantes de maneira rápida e eficaz.

4.6.1 Operações Básicas em MongoDB

Numa base de dados NoSQL como o MongoDB, não é necessário implementar funções separadas para realizar operações básicas como inserir, apagar ou atualizar documentos. O MongoDB fornece métodos internos para lidar diretamente com estas operações. Aqui está um breve resumo de como pode realizar estas operações no MongoDB:

- **Inserir Documentos:** Para inserir um documento numa coleção, pode usar os métodos `insertOne` ou `insertMany`.

- **Apagar Documentos:** Para apagar documentos, pode usar os métodos `deleteOne` ou `deleteMany`.

- **Atualizar Documentos:** Para atualizar documentos, pode usar os métodos `updateOne`, `updateMany` ou `replaceOne`.

O MongoDB, sendo uma base de dados NoSQL, é projetado para manipular dados de forma mais flexível e direta, eliminando a necessidade de mecanismos de controlo adicionais nas operações de **Criação** (*CREATE*), **Atualização** (*UPDATE*) e **Remoção** (*DELETE*).

De seguida, iremos apresentar e explicar de forma breve cada uma das funções e expressões implementadas.

4.6.2 Visão Paciente

Funções

getPatientById: Obtém informações de um paciente específico, dado um *id_patient*. Recebe ID como argumento e retorna os dados do paciente.

getMedicalHistoryById: Obtém o histórico médico de um paciente específico. Recebe ID como argumento e retorna todo o histórico médico do paciente.

getInsuranceById: Obtém as informações de seguro de um paciente específico. Recebe ID como argumento e retorna os dados do seguro do paciente.

getEmergencyContactById: Obtém o contacto de emergência de um paciente específico. Recebe ID como argumento e retorna os dados do contacto de emergência.

getPatientsByBloodType: Obtém pacientes por tipo sanguíneo. Recebe *bloodType* como argumento e retorna os pacientes com o tipo sanguíneo especificado.

getPatientsByGender: Obtém pacientes por género. Recebe *gender* como argumento e retorna os pacientes com o género especificado.

getPatientsByCondition: Obtém pacientes por condição médica. Recebe *condition* como argumento e retorna os pacientes com a condição médica especificada.

getPatientsByMedicalRecordDate: Obtém pacientes com registros médicos em uma data específica. Recebe *date* como argumento e retorna os pacientes com registos nessa data.

countStaffByRole: Conta o número de enfermeiros, doutores e técnicos. Retorna um objeto com as contagens de cada papel.

getPatientsByMedicalRecordDateRange: Obtém pacientes com registros médicos em um intervalo de datas. Recebe *startDate* e *endDate* como argumentos e retorna os pacientes com registos nesse intervalo.

getPatientsByBirthday: Obtém pacientes com uma data de aniversário específica. Recebe *birthday* como argumento e retorna os pacientes com essa data de aniversário.

getPatientsByInsuranceProvider: Obtém pacientes por provedor de seguro. Recebe *provider* como argumento e retorna os pacientes com o provedor de seguro especificado.

getPatientsByInsurancePlan: Obtém pacientes por plano de seguro. Recebe *plan* como argumento e retorna os pacientes com o plano de seguro especificado.

getPatientsByCoverage: Obtém pacientes por tipo de cobertura. Recebe *coverage* como argumento e retorna os pacientes com o tipo de cobertura especificado.

listPatientsByAgeRange: Lista pacientes por intervalo de idades. Recebe *minAge* e *maxAge* como argumentos e retorna os pacientes dentro desse intervalo de idades.

listPatientsWithMaternityCoverage: Obtém pacientes com cobertura de maternidade. Recebe boolean *maternity* como argumento e retorna os pacientes com cobertura de maternidade.

listPatientsWithDentalCoverage: Obtém pacientes com cobertura dental. Recebe boolean *dental* como argumento e retorna os pacientes com cobertura dental.

listPatientsWithOpticalCoverage: Obtém pacientes com cobertura ótica. Recebe boolean *optical* como argumento e retorna os pacientes com cobertura ótica.

findPatientsByEmergencyContactRelation: Obtém pacientes por relação de contacto de emergência. Recebe *relation* como argumento e retorna os pacientes com essa relação de contacto.

getPatientByName: Obtém um paciente pelo primeiro nome e sobrenome. Recebe *firstName* e *lastName* como argumentos e retorna o paciente com os nomes especificados.

getPatientByPhone: Obtém um paciente pelo número de telefone. Recebe *phone* como argumento e retorna o paciente com o número de telefone especificado.

getPatientsByEmergencyContact: Obtém pacientes com um contacto de emergência específico. Recebe *firstName* e *lastName* como argumentos e retorna os pacientes com o contacto de emergência especificado.

getPatientsByRecordId: Obtém pacientes pelo *record_id* do histórico médico. Recebe *recordId* como argumento e retorna os pacientes com o *record_id* especificado.

getEpisodesByCondition: Obtém episódios médicos por condição médica. Recebe *condition* como argumento e retorna os episódios médicos com a condição especificada.

Expressões

Buscar Todos os Tipos de Relações em Contactos de Emergência: Obtém todos os tipos de relações em contactos de emergência. Desagrupa os contactos de emergência e agrupa por tipo de relação, retornando os tipos de relação.

Buscar Todos os Tipos de Provedores de Seguro: Obtém todos os tipos de provedores de seguro. Agrupa por provedores de seguro, retornando os tipos de provedores.

Buscar Todos os Tipos de Planos de Seguro: Obtém todos os tipos de planos de seguro.

Agrupa por planos de seguro, retornando os tipos de planos.

Buscar Todos os Tipos de Coverage: Obtém todos os tipos de cobertura de seguro. Agrupa por tipos de cobertura, retornando os tipos de cobertura.

Buscar Todos os Tipos de Condições Médicas: Obtém todos os tipos de condições médicas. Desagrupa o histórico médico e agrupa por condição médica, retornando os tipos de condições.

Buscar Todos os Tipos Sanguíneos: Obtém todos os tipos sanguíneos. Agrupa por tipo sanguíneo, retornando os tipos sanguíneos.

Buscar quantos Pacientes existem para cada BloodType: Conta o número de pacientes para cada tipo sanguíneo. Agrupa por tipo sanguíneo e retorna a contagem de pacientes para cada tipo.

Buscar quantos Pacientes existem para cada Condition: Conta o número de pacientes para cada condição médica. Desagrupa o histórico médico e agrupa por condição médica, retornando a contagem de pacientes para cada condição.

4.6.3 Visão Staff

Funções

getStaffInfo: Obtém todas as informações de um membro do staff, dado um *emp_id*. Recebe *emp_id* como argumento e retorna os dados do membro do staff.

getDepartmentInfo: Obtém as informações do departamento de um membro do staff para um dado *emp_id*. Recebe *emp_id* como argumento e retorna os dados do departamento ou *null* se não houver.

AllInfoStaffByDateJoining: Obtém todas as informações dos membros do staff que se juntaram em uma data específica. Recebe *date_joining* como argumento e retorna os dados dos membros do staff.

AllInfoStaffByDateSeparation: Obtém todas as informações dos membros do staff que se separaram em uma data específica. Recebe *date_separation_str* como argumento e retorna os dados dos membros do staff.

getStaffByActiveStatus: Obtém todos os membros do staff com um status de atividade específico. Recebe *is_active_status* como argumento e retorna os dados dos membros do staff.

getQualificationsByDoctor: Obtém as qualificações de um doutor específico. Recebe *emp_id* como argumento e retorna as qualificações do doutor ou *null* se não houver.

getStaffByName: Obtém um membro do staff pelo primeiro nome e sobrenome. Recebe *firstName* e *lastName* como argumentos e retorna os dados do membro do staff.

getStaffByEmail: Obtém um membro do staff pelo email. Recebe *email* como argumento e retorna os dados do membro do staff.

getStaffBySSN: Obtém um membro do staff pelo (*SSN*). Recebe *ssn* como argumento e retorna os dados do membro do staff.

Expressões

Buscar toda a informação das Enfermeiras: Obtém todas as informações dos membros do staff que têm o papel de *NURSE*. Retorna uma lista com os dados das enfermeiras.

Buscar toda a informação dos Médicos: Obtém todas as informações dos membros do staff que têm o papel de *DOCTOR*. Retorna uma lista com os dados dos médicos.

Buscar toda a informação dos Técnicos: Obtém todas as informações dos membros do staff que têm o papel de *TECHNICIAN*. Retorna uma lista com os dados dos técnicos.

Buscar quantos Enfermeiros existem: Conta o número de membros do staff com o papel de *NURSE*. Retorna a contagem de enfermeiros.

Buscar quantos Doutores existem: Conta o número de membros do staff com o papel de *DOCTOR*. Retorna a contagem de doutores.

Buscar quantos Técnicos existem: Conta o número de membros do staff com o papel de *TECHNICIAN*. Retorna a contagem de técnicos.

Buscar quantos Departments existem: Conta o número de departamentos diferentes. Agrupa por *id_department* e retorna a contagem de departamentos.

Buscar todos os tipos de Qualifications: Obtém todos os tipos de qualificações dos doutores. Desagrupa as qualificações e agrupa por tipo de qualificação, retornando os tipos de qualificações.

Número de Empregados por Departamento: Conta o número de empregados por departamento. Agrupa por *id_department* e retorna a contagem de empregados para cada departamento.

Nurses por Departamento: Conta o número de enfermeiras por departamento. Agrupa por *id_department* e retorna a contagem de enfermeiras para cada departamento.

Número de Doctors por Departamento: Conta o número de doutores por departamento. Agrupa por *id_department* e retorna a contagem de doutores para cada departamento.

Número de Technicians por Departamento: Conta o número de técnicos por departamento. Agrupa por *id_department* e retorna a contagem de técnicos para cada departamento.

Contar o Número Total de Staff: Conta o número total de membros do staff. Retorna a contagem total de membros.

Buscar todos os Staff que Estão Ativos: Obtém todos os membros do staff que estão ativos. Retorna uma lista com os dados dos membros ativos.

Buscar todos os Staff que não estão Ativos: Obtém todos os membros do staff que não estão ativos. Retorna uma lista com os dados dos membros inativos.

Contar quantos Staff Estão Ativos: Conta o número de membros do staff que estão ativos. Retorna a contagem de membros ativos.

Contar quantos Staff não estão Ativos: Conta o número de membros do staff que não estão ativos. Retorna a contagem de membros inativos.

4.6.4 Visão Episodes

Funções

getAllInfoByEpisodeId: Esta função retorna todas as informações sobre um episódio específico, dado um `episodeId`. Recebe um `episodeId` como argumento e retorna todos os documentos que correspondem a esse ID na coleção `episodes`.

getAllInfoByEpisodeId_withNames: Retorna todas as informações sobre um episódio específico, incluindo informações do paciente, dado um `episodeId`. Recebe um `episodeId` e retorna documentos agregados com detalhes do paciente.

getAllInfoByPrescriptionId: Retorna todas as informações sobre uma prescrição específica dentro dos episódios. Recebe um `prescriptionId` e retorna os documentos que contêm essa prescrição.

getPrescriptionById: Obtém detalhes específicos de uma prescrição. Recebe um `prescriptionId` e retorna os detalhes da prescrição sem outros dados do episódio.

getAllInfoByPatientId: Retorna todas as informações sobre os episódios de um paciente específico. Recebe um `id_patient` e retorna os episódios desse paciente.

getPatientEpisodes: Conta o número de episódios para um paciente específico. Recebe um `id_patient` e retorna a contagem de episódios.

getPrescriptionsByEpisodeId: Obtém todas as prescrições de um episódio específico. Recebe um `episodeId` e retorna as prescrições desse episódio.

getBillsByEpisodeId: Retorna todas as faturas de um episódio específico. Recebe um `episodeId` e retorna as faturas desse episódio.

getBillById: Obtém uma fatura específica por `id_bill`. Recebe um `id_bill` e retorna a fatura correspondente.

getLabScreeningsByEpisodeId: Retorna todos os exames laboratoriais de um episódio específico. Recebe um `episodeId` e retorna os exames laboratoriais desse episódio.

getHospitalizationByEpisodeId: Obtém informações de hospitalização para um episódio específico. Recebe um `episodeId` e projeta apenas o campo `hospitalization` no resultado.

getRoomById: Obtém informações de uma sala específica com base no seu ID. Recebe `roomId` como argumento e retorna os dados da sala correspondente.

getPrescriptionsByPatientId: Obtém todas as prescrições associadas a um paciente específico. Recebe `patientId` como argumento e retorna uma lista de prescrições desse paciente.

getBillsByPatientId: Obtém todas as faturas relacionadas a um paciente. Recebe `patientId` como argumento e retorna as faturas correspondentes.

getLabScreeningsByPatientId: Obtém todos os exames laboratoriais de um paciente. Recebe `patientId` como argumento e retorna os exames laboratoriais desse paciente.

getHospitalizationByPatientId: Obtém informações de hospitalização para um paciente específico. Recebe `patientId` como argumento e retorna os dados de hospitalização.

getRoomsByPatientId: Obtém todas as salas associadas às hospitalizações de um paciente. Recebe `patientId` como argumento e retorna os dados das salas.

getMedicinesByPatientId: Obtém todas as informações de medicamentos prescritos para um paciente. Recebe `patientId` como argumento e retorna os dados dos medicamentos.

getMedicinesByEpisodeId: Obtém todos os medicamentos prescritos em um episódio específico. Recebe `episodeId` como argumento e retorna os dados dos medicamentos desse episódio.

getMedicineById: Obtém os dados de um medicamento específico. Recebe `medicineId` como argumento e retorna os dados do medicamento.

getMedicineByPrescriptionId: Obtém informações dos medicamentos associados a uma prescrição específica. Recebe `prescriptionId` como argumento e retorna os dados dos medicamentos dessa prescrição.

getMedicineByName: Esta função obtém informações de um medicamento específico pelo seu nome. Recebe *medicineName* como argumento e retorna os dados do medicamento.

getPrescriptionsByDate: Obtém todas as prescrições em uma data específica. Recebe *date* como argumento e retorna as prescrições dessa data.

getPrescriptionsBetweenDates: Obtém todas as prescrições entre duas datas. Recebe *startDate* e *endDate* como argumentos e retorna as prescrições nesse intervalo de datas.

getPrescriptionsBetweenDosages: Obtém todas as prescrições entre duas dosagens. Recebe *minDosage* e *maxDosage* como argumentos e retorna as prescrições dentro desse intervalo de dosagens.

getTotalBillsByPatientId: Soma o total das faturas para um paciente específico. Recebe *patientId* como argumento e retorna o valor total das faturas.

getTotalCostsByPatientId: Soma todas as contas (room cost, test cost, e other charges) para um paciente específico. Recebe *patientId* como argumento e retorna os custos totais.

getLabScreeningsByDateRange: Obtém todos os exames laboratoriais dentro de um intervalo de datas. Recebe *startDate* e *endDate* como argumentos e retorna os exames laboratoriais nesse intervalo.

getLabScreeningsByPriceRange: Obtém todos os exames laboratoriais dentro de um intervalo de custo. Recebe *minPrice* e *maxPrice* como argumentos e retorna os exames laboratoriais dentro desse intervalo de custo.

getHospitalizationsByDateRange: Obtém registros de hospitalização com base em um intervalo de datas de admissão e alta. Recebe *startDate* e *endDate* como argumentos e retorna as hospitalizações nesse intervalo de datas.

getAppointmentsByPatientId: Obtém todas as consultas de um paciente específico. Recebe *patientId* como argumento e retorna as consultas desse paciente.

getAppointmentsByPatientId: Obtém todas as consultas de um paciente específico. Recebe *patientId* como argumento e retorna as consultas desse paciente.

getAppointmentsByEpisodeId: Obtém todas as consultas de um episódio específico. Recebe *episodeId* como argumento e retorna as consultas desse episódio.

getAppointmentsByScheduleOnDate: Obtém todas as consultas agendadas para uma data específica. Recebe *date* como argumento e retorna as consultas agendadas nessa data.

getAppointmentsByAppointmentDate: Obtém todas as consultas para uma data específica. Recebe *date* como argumento e retorna as consultas dessa data.

getAppointmentsByAppointmentTime: Obtém todas as consultas para um horário específico. Recebe *time* como argumento e retorna as consultas desse horário.

getAppointmentsByDoctorId: Obtém todas as consultas de um médico específico. Recebe *doctorId* como argumento e retorna as consultas desse médico.

getBillsByDateRange: Obtém todas as faturas registradas entre duas datas específicas. Recebe *startDate* e *endDate* como argumentos e retorna as faturas registradas nesse intervalo de datas.

getPatientsByPaymentStatus: Obtém pacientes com um *payment_status* específico. Recebe *status* como argumento e retorna os dados dos pacientes e seus episódios com o status de pagamento especificado.

getTotalCostByRegisteredDate: Calcula o custo total das faturas registradas entre duas datas. Recebe *startDate* e *endDate* como argumentos e retorna o custo total das faturas nesse intervalo de datas.

getNurseInfoByEpisodeId: Obtém informações da enfermeira responsável por um episódio específico. Recebe *episodeId* como argumento e retorna os dados da enfermeira responsável.

getNursesByPatientId: Obtém todas as enfermeiras que já cuidaram de um determinado paciente. Recebe *patientId* como argumento e retorna os dados das enfermeiras.

getDoctorInfoByEpisodeId: Obtém informações do médico responsável por um episódio específico. Recebe *episodeId* como argumento e retorna os dados do médico responsável.

getDoctorsByPatientId: Obtém todas as informações dos médicos que atenderam um paciente específico. Recebe *patientId* como argumento e retorna os dados dos médicos.

getTechnicianInfoByEpisodeId: Obtém informações do técnico responsável por um episódio específico. Recebe *episodeId* como argumento e retorna os dados do técnico responsável.

getTechniciansByPatientId: Obtém todas as informações dos técnicos que atenderam um paciente específico. Recebe *patientId* como argumento e retorna os dados dos técnicos.

getPatientInfoByIdFromEpisodes: Retorna informações de um paciente específico, dado um *id_patient*. Recebe *patientId* como argumento e retorna os dados do paciente.

getPatientsByRoom: Lista os pacientes alocados a um quarto específico. Recebe *roomId* como argumento e retorna os dados dos pacientes e informações do quarto.

getHospitalizationsByNurse: Lista hospitalizações por enfermeira responsável. Recebe *nurseId* como argumento e retorna os dados das hospitalizações e da enfermeira.

getAllEpisodesForPatient: Lista todos os episódios médicos de um paciente específico. Recebe *patientId* como argumento e retorna os episódios do paciente.

getEpisodesByDoctor: Lista todos os episódios médicos de um médico específico. Recebe *doctorId* como argumento e retorna os episódios do médico.

getLabScreeningsByTechnician: Lista exames baseados no técnico responsável. Recebe *technicianId* como argumento e retorna os exames e informações do técnico.

getAppointmentsByDateAndTime: Busca consultas por data e horário específico. Recebe *date* e *time* como argumentos e retorna as consultas e informações dos pacientes.

getAppointmentsByDoctorByDay: Lista as consultas de um médico específico por dia. Recebe *doctorId* como argumento e retorna as consultas agrupadas por dia, com detalhes dos pacientes e do médico.

Expressões

Contar *Appointments* por Médico: Conta o número total de consultas para cada médico. Agrupa os resultados pelo campo `id_doctor` e retorna o número total de consultas para cada médico.

Calcular o Total de Medicamentos e Custo Total: Calcula o total de medicamentos e o custo total de todos os medicamentos prescritos. Agrupa os resultados e retorna a quantidade total e o custo total.

Calcular o Total de Cada Medicamento Usado e o Custo Total: Calcula a quantidade total e o custo total para cada tipo de medicamento usado. Agrupa os resultados pelo nome do medicamento e retorna a quantidade total e o custo total para cada medicamento.

Calcular o Custo Total dos Testes e a Contagem de Testes Realizados: Calcula o custo total dos testes e a contagem de testes realizados. Agrupa os resultados e retorna o custo total e o número total de testes.

Buscar Todas as Faturas com Detalhes dos Pacientes: Obtém todas as faturas com detalhes dos pacientes associados. Realiza uma junção com a coleção `patients` para incluir informações dos pacientes.

Buscar Todos os Episódios que Ainda Não Acabaram: Obtém todos os episódios que ainda não foram encerrados. Retorna os episódios com a data de alta nula.

Buscar Informações de Todos os Pacientes: Obtém informações de todos os pacientes associados aos episódios. Retorna os dados agregados dos pacientes.

Buscar Todos os Episódios e o Respetivo Paciente: Obtém todos os episódios e os dados do paciente associado. Retorna as informações dos episódios e dos pacientes.

Lista os Médicos com Mais Consultas Marcadas: Lista os médicos com mais consultas marcadas. Agrupa e ordena os médicos pelo número total de consultas, retornando os dados dos médicos e o total de consultas.

Lista os Médicos com Mais Consultas Marcadas, com Informação Detalhada do Paciente: Lista os médicos com mais consultas marcadas, incluindo detalhes dos pacientes. Agrupa e ordena os médicos pelo número total de consultas, retornando os dados dos médicos e informações detalhadas dos pacientes.

Capítulo 5

Base de dados orientada a grafos - Neo4J

Além da migração para uma base de dados não relacional orientada a documentos, também implementamos um modelo na base de dados **Neo4j**, conhecida por sua orientação a grafos. Após uma análise aprofundada do nosso esquema relacional inicial, identificamos as relações mais significativas que podem ser representadas de forma eficiente neste paradigma não relacional.

O Neo4j permite a modelagem de dados em termos de nós e relacionamentos, proporcionando uma representação intuitiva das interconexões entre os dados. Com base nessa abordagem, identificamos e selecionamos os nós mais relevantes para representar corretamente as informações no Neo4j. Além disso, escolhemos as relações mais significativas para o nosso esquema relacional, traduzindo-as para estruturas de grafos que preservam a integridade das relações existentes. O schema da nossa base de dados ficou da seguinte forma:

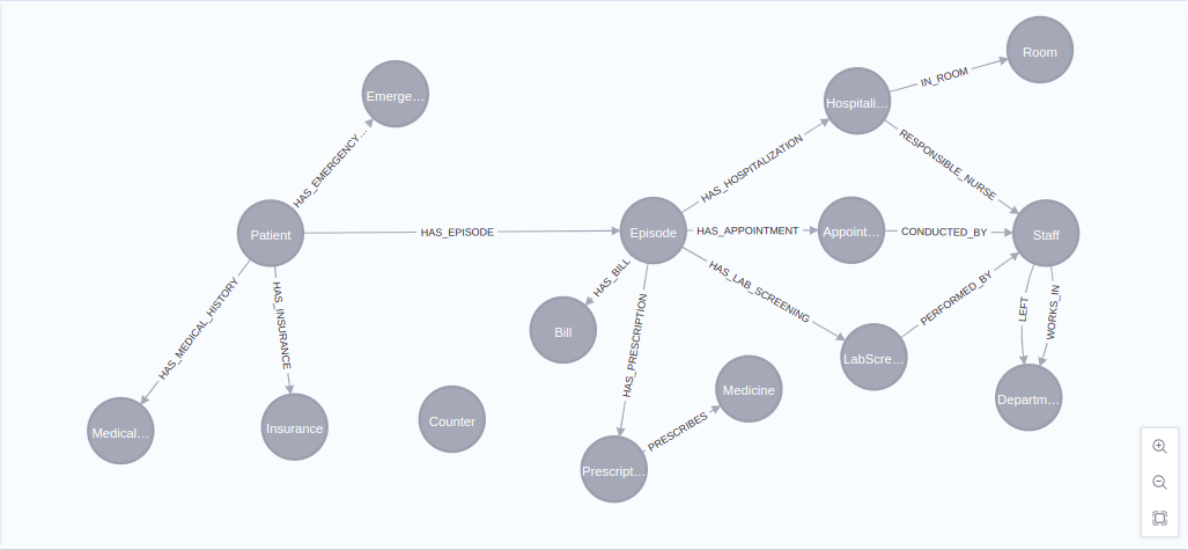


Figura 5.1: Esquema da nossa base de dados Neo4j

O esquema final no Neo4j inclui várias entidades (nós) e suas inter-relações, modelando de forma precisa o contexto hospitalar. A nossa estruturação de dados final têsm os 15 nós seguintes:

1. `Paciente (Patient)`: Este nó representa todos os pacientes que já tiveram algum episódio hospitalar, exibindo os campos originais presentes na base de dados SQL. A.1

2. `Seguro de Saúde (Insurance)`: Este nó é representativo de vários seguros de saúde e

contém os mesmos campos encontrados na tabela correspondente na base de dados SQL. A.2

3. `Histórico Médico (Medical History)`: Este nó contém o histórico médico detalhado de cada paciente, mantendo os campos originais presentes na tabela SQL correspondente. A.3

4. `Contactos de Emergência (Emergency Contacts)`: Este nó representa os contactos de emergência dos pacientes, com os mesmos campos encontrados na tabela SQL correspondente. A.4

5. `Episódios Médicos (Episodes)`: Este nó é responsável por registar todos os episódios médicos de cada paciente, mantendo os campos originais da tabela correspondente presentes na base de dados SQL. A.5

6. `Faturas (Bill)`: Este nó representa as faturas associadas aos serviços prestados aos pacientes, mantendo os mesmos campos da tabela SQL correspondente. A.6

7. `Prescrição Médica (Prescription)`: Este nó contém informações sobre as prescrições médicas emitidas aos pacientes, mantendo a estrutura da tabela correspondente em SQL. A.7

8. Medicamento (Medicine): Este nó representa os medicamentos disponíveis no hospital, mantendo os mesmos campos da tabela SQL correspondente. A.8

9. `Departamento (Department)`: Este nó representa os diferentes departamentos do hospital, mantendo os parâmetros da tabela correspondente em SQL.A.9

10. `Funcionários (Staff)`: Este nó reúne todas as tabelas referentes aos funcionários do nosso hospital, incluindo médicos, enfermeiros e técnicos. Dado que as estruturas das tabelas SQL para esses três tipos de funcionários são muito semelhantes, decidimos adotar uma abordagem semelhante ao MongoDB, abstraindo estas três tabelas e reunindo todos os elementos em um único nó denominado `staff`. Neste nó, apenas adicionamos um campo `role`, responsável por determinar a função do funcionário (DOCTOR, NURSE, TECHNICIAN). Para os funcionários com o papel de médico (`DOCTOR`), também adicionamos um campo `qualification`, característico da tabela `doctor` no SQL. A.10

11. `Consulta (Appointment):` Este nó regista as consultas marcadas pelos pacientes, mantendo os campos originais presentes na base de dados SQL. A.11

12. `Testes Médicos (Lab Screenings)`: Este nó contém informações sobre os testes médicos realizados nos pacientes, mantendo os mesmos campos da tabela SQL correspondente. A.12

13. `Hospitalização (Hospitalization)`: Este nó regista informações sobre a hospitalização dos pacientes, mantendo os campos originais presentes na base de dados SQL. A.13

14. `Quarto (Room)`: Este nó representa os quartos presentes no hospital. Os parâmetros deste nó são os mesmos da tabela correspondente na base de dados SQL.A.14

15. `Contador (Counter)`: Este nó auxiliar serve como um contador de IDs para os vários tipos de nós que possuem identificadores. Utilizando este nó, é possível obter um ID único sempre que um dos restantes tipos de nós seja adicionado, mantendo, desta forma, a integridade dos dados na base de dados Neo4j.A.15

No esquema final temos 16 relações entre os nós anteriormente mencionados. As relações são as seguintes:

1. `HAS_APPOINTMENT`: Representa a relação entre um paciente (*patient*) e uma consulta agendada (*appointment*). Um nó de paciente possui essa relação com um nó de consulta, indicando que o paciente tem uma consulta marcada.

2. `HAS_BILL`: Reflete a relação entre um episódio médico (*episode*) e uma fatura associada (*bill*) a esse episódio. Um nó de episódio médico está conectado a um nó de fatura, indicando que ao episódio encontra-se associada uma fatura.

3. `HAS_EMERGENCY_CONTACT`: Indica a relação entre um paciente (*patient*) e um contato de emergência (*emergency contact*). Um nó de paciente está conectado a um ou mais nós de contato de emergência, representando que o paciente possui um ou mais contatos de emergência registados.

4. `HAS_EPISODE`: Reflete a relação entre um paciente (*patient*) e um episódio médico (*episode*). Um nó de paciente possui essa relação com um nó de episódio médico, indicando que o paciente está envolvido nesse episódio e correlacionando assim o paciente com os elementos médicos.

5. `HAS_HOSPITALIZATION`: Representa a relação entre um episódio médico (*episode*) e uma hospitalização (*hospitalization*) associada a esse episódio. Um nó de episódio médico está conectado a um nó de hospitalização, indicando que o episódio médico recorreu a uma hospitalização do paciente a qual o episódio se encontra associado.

6. `HAS_INSURANCE`: Indica a relação entre um paciente (*patient*) e um seguro de saúde (*insurance*). Um nó de paciente possui essa relação com um nó de seguro de saúde, indicando qual o seguro de saúde do paciente em questão.

7. `HAS_MEDICAL_HISTORY`: Reflete a relação entre um paciente (*patient*) e seu histórico médico (*medical history*). Um nó de paciente está conectado a um ou mais nós de histórico médico, indicando que o paciente tem um ou mais nós de histórico médico registados.

8. `HAS_PRESCRIPTION`: Denota a relação entre um episódio médico (*episode*) e uma prescrição (*prescription*) associada a esse episódio. Um nó de episódio médico está conectado a um ou mais nós de prescrição, indicando que o episódio possui uma ou mais prescrições.

9. `IN_ROOM`: Representa a relação entre uma hospitalização (*hospitalization*) e o quarto (*room*) onde o paciente está internado. Um nó de hospitalização está conectado a um nó de quarto, indicando o quarto onde a hospitalização ocorre. Um nó de quarto pode ter associado várias hospitalizações desde que em horários que não coincidam.

10. `LEFT`: Indica a relação entre um paciente (*patient*) e sua hospitalização (*hospitalization*), semelhante ao que ocorre com a relação HAAS_HOSPITALIZATION, no entanto, neste caso apenas os pacientes que já abandonaram uma hospitalização apresentam esta relação, ou seja, pacientes com alta hospitalar. Um nó de paciente possui essa relação com um nó de hospitalização, representando que o paciente teve alta do hospital.

11. `PRESCRIBES`: Reflete a relação entre um médico (*doctor*) e uma prescrição (*prescription*) que ele emite. Um nó de médico está conectado a um ou mais nós de prescrição, indicando que o médico emitiu a prescrição.

12. `RESPONSIBLE_NURSE`: Indica a relação entre uma hospitalização (*hospitalization*) e a enfermeira responsável (*nurse*) por essa hospitalização. Um nó de hospitalização está conectado a um nó de enfermeira, representando que a enfermeira é responsável por essa hospitalização.

13. `CONDUCTED_BY`: Representa a relação entre uma consulta agendada (*appointment*) e o médico que a realizou. Um nó de consulta está conectado a um nó de médico, indicando que o médico conduziu a consulta.

14. `WORKS_IN`: Denota a relação entre um funcionário (médico, enfermeiro, técnico) e o departamento (*department*) em que ele trabalha. Um nó de funcionário possui essa relação com um nó de departamento, indicando que o funcionário trabalha no departamento.

15. `HAS_LAB_SCREENING`: Representa a relação entre um episódio médico (*episode*) e um exame laboratorial (*lab screening*) associado a esse episódio. Um nó de episódio médico está conectado a um ou mais nós de exame laboratorial, indicando que o episódio médico incluiu a realização de um ou mais exames exame.

16. `PERFORMED_BY`: Reflete a relação entre um exame laboratorial (*lab screening*) e um técnico (*technician*) responsável por realizá-lo. Um nó de exame laboratorial está conectado a um nó de técnico, indicando que o técnico foi o responsável pela execução do exame laboratorial. O mesmo técnico pode ser responsável pela realização de vários exames laboratoriais.

5.1 *Constraints* em Neo4j

A migração de dados de Oracle para Neo4j envolveu a definição de várias *constraints* para assegurar a integridade e a unicidade dos dados na base de dados orientada a grafos. As *constraints*

adicionadas garantem que os dados inseridos no Neo4j respeitam a unicidade dos ids, quando os mesmos existem, prevenindo duplicações e mantendo a consistência das informações armazenadas. A seguir, detalhamos as 14 *constraints* implementadas:

1. Pacientes (*Patient*): Para garantir que cada paciente é único na base de dados, foi criada uma *constraint* para assegurar a unicidade do identificador de paciente (`id_patient`), assim, adicionamos uma *constraint* ao identificador deste nó garantindo que não existem dois pacientes com o mesmo ID.

```
CREATE CONSTRAINT FOR (p:Patient) REQUIRE p.id_patient IS UNIQUE
```

2. Seguros de Saúde (*Insurance*): De modo a evitar duplicações nas apólices de seguro, foi adicionada uma constraint de unicidade para o número da apólice (policy_number).

```
CREATE CONSTRAINT FOR (i:Insurance) REQUIRE i.policy_number IS UNIQUE
```

3. Contatos de Emergência (*Emergency Contact*): Para garantir que para este tipo de nó cada combinação de nome de contato, telefone e relação é única, foi criada uma *constraint* composta incluindo estes três parâmetros.

```
CREATE CONSTRAINT unique_contact FOR (c:EmergencyContact) REQUIRE
↪ (c.contact_name, c.phone, c.relation) IS UNIQUE
```

4. Histórico Médico (*Medical History*): Cada registo de histórico médico (`id_record`) deve ser único para assegurar a integridade dos dados históricos dos pacientes, adicionamos, assim, uma *constraint* no identificador deste nó como forma de garantir que não existem dois registos de histórico médico iguais.

```
CREATE CONSTRAINT FOR (m:MedicalHistory) REQUIRE m.id_record IS UNIQUE
```

5. Departamentos (*Department*): Os departamentos são identificados de forma única pelo seu identificador de departamento (`id_department`), ao colocarmos uma *constraint* única no seu identificador garantimos que não é possível existir dois departamentos com o mesmo ID.

```
CREATE CONSTRAINT FOR (dep:Department) REQUIRE dep.id_department IS
↪ UNIQUE
```

6. Funcionários (*Staff*): Para assegurar a unicidade dos funcionários na base de dados, foi adicionada uma *constraint* no identificador do funcionário (`id_emp`), desta forma garantimos que não existem dois funcionários com o mesmo ID.

```
CREATE CONSTRAINT FOR (s:Staff) REQUIRE s.id_emp IS UNIQUE
```

7. Quartos (*Room*): Cada quarto ligado à hospitalização é identificado de forma única pelo seu identificador de quarto (`id_room`), pelo que, apenas é necessário incluir uma *constraint* neste identificador, garantindo que não existam dois quartos com o mesmo ID.

```
CREATE CONSTRAINT FOR (r:Room) REQUIRE r.id_room IS UNIQUE
```

8. Faturas (*Bill*): As faturas são únicas e identificadas pelo seu identificador de fatura (`id_bill`), assim sendo, apenas é necessário incluir uma *constraint* neste identificador, garantindo que não existam duas faturas com o mesmo ID.

```
CREATE CONSTRAINT FOR (b:Bill) REQUIRE b.id_bill IS UNIQUE
```

9. Medicamentos (*Medicine*): Cada medicamento é identificado de forma única pelo seu identificador de medicamento (`id_medicine`), pelo que, apenas é necessário incluir uma *constraint* neste identificador, garantindo que não existam dois medicamentos com o mesmo ID.

```
CREATE CONSTRAINT FOR (m:Medicine) REQUIRE m.id_medicine IS UNIQUE
```

10. Prescrições (*Prescription*): De modo a evitar a existência de duplicações nas prescrições, foi criada uma constraint no identificador da prescrição (`id_prescription`), garantindo que não existem duas prescrições com o mesmo ID.

```
CREATE CONSTRAINT FOR (p:Prescription) REQUIRE p.id_prescription IS
↪ UNIQUE
```

11. Exames Laboratoriais (*Lab Screening*): Cada exame laboratorial é identificado de forma única pelo seu identificador de exame (`id_lab`), pelo que, apenas é necessário incluir uma *constraint* neste identificador, garantindo que não existam dois exames laboratoriais com o mesmo ID.

```
CREATE CONSTRAINT FOR (l:LabScreening) REQUIRE l.id_lab IS UNIQUE
```

12. Consultas (*Appointment*): Sendo que o nó não apresenta um identificador único, foi utilizado uma *constraint* composta para garantir que as consultas são únicas. Uma consulta é identificadas pela combinação da data, hora e identificador do médico responsável, garantindo que um mesmo médico não pode dar duas consultas numa mesma data e hora.

```
CREATE CONSTRAINT unique_appointment FOR (a:Appointment)
↪ REQUIRE(a.appointment_date, a.appointment_time, a.id_doctor) IS
↪ UNIQUE
```

13. Hospitalizações (*Hospitalization*): Uma vez que este nó não apresenta um identificador único e para assegurar a unicidade das hospitalizações, foi criada uma *constraint* composta que

considera a data de admissão e o identificador do episódio médico, garantindo que duas hospitalizações na mesma data têm de ser feitas por episódios médicos diferentes, ou seja, para pacientes diferentes.

```
CREATE CONSTRAINT unique_hospitalization FOR (h:Hospitalization)
↪ REQUIRE (h.admission_date, h.id_episode) IS UNIQUE
```

14. Episódios Médicos (*Episode*): Cada episódio médico é identificado de forma única pelo seu identificador de episódio (`id_episode`), pelo que apenas é necessário incluir uma *constraint* neste identificador, garantindo que não existam dois episódios com o mesmo ID.

```
CREATE CONSTRAINT FOR (e:Episode) REQUIRE e.id_episode IS UNIQUE
```

5.2 Triggers em Neo4j

Durante o processo de migração de dados da base de dados Oracle para Neo4j, foi necessário definir diversos *triggers* para otimizar a inserção de novos nós e manter a integridade dos dados. Na nossa base de dados, procurávamos evitar o uso manual de campos de identificação durante a inserção de novos nós. Para isso, utilizamos *triggers* que gerassem automaticamente esses identificadores, garantindo a integridade e a unicidade dos dados.

Para alcançar este objetivo, introduzimos um nó auxiliar (*Counter*) que armazena o número máximo dos identificadores utilizados até o momento. Este nó facilita a geração de novos identificadores únicos para futuros nós. Assim, após a inserção de um novo nó, os *triggers* eram acionados para obter o próximo identificador a partir do nó *Counter*, incrementar o contador presente neste nó, de acordo com o seu tipo, e atribuir o novo identificador ao nó recém-criado.

Além dos *triggers* relacionados à criação de identificadores únicos, foi necessário adaptar um *trigger* existente na base de dados original Oracle para o ambiente Neo4j. Este *trigger* específico foi ajustado para garantir que sua funcionalidade fosse preservada após a migração, mantendo a consistência e a continuidade das operações automatizadas.

Em Neo4j, para a correta criação dos triggers, foi necessário recorrer à biblioteca **APOC** (*Awesome Procedures on Cypher*). A biblioteca APOC fornece acesso a procedimentos e funções definidos pelos utilizadores que estendem o uso da linguagem de consulta *Cypher* para novas áreas. Utilizando os recursos avançados oferecidos pela APOC, conseguimos automatizar e otimizar a criação e gestão de identificadores únicos para novos nós.

5.2.1 Triggers para a adição de identificadores únicos

Na sequencia da criação destes *triggers* foi observado a existência de algumas consistências entre o nome do nó (`label`) e o nome do ID (`id_ + label`). Estas consistências dividiram a implementação

em dois tipos principais: *triggers* dinâmicos para *labels* comuns e *triggers* específicos para *labels* com nomes de identificador diferentes do nome da *label*.

Os dados de transações em Neo4j são transformados em estruturas de dados apropriadas para serem consumidas como parâmetros nas instruções. Recorremos, portanto, à variável `$createdNodes` que permite, quando um nó é criado no Neo4j, acionar o nosso *trigger*. O parâmetro `$createdNodes` é uma lista que contém os nós recém-criados durante a transação. Esta lista é então utilizada pelos *triggers* para executar operações específicas, como, no nosso caso, a criação de identificadores únicos.

Assim sendo, para as *labels*: *Patient*, *Room*, *Department*, *Episode*, *Medicine*, *Prescription* e *Bill*, foi criado um único *trigger* que fosse acionados com a inserção de nós das *labels* mencionadas.

```
CALL apoc.trigger.install(
    'hospital',
    'dynamic_id_trigger',
    '
    UNWIND $createdNodes AS n
    WITH n, head(apoc.node.labels(n)) AS label
    WITH n, label, \'id_\' + toLower(label) AS dynamicProperty
    WITH n, label, apoc.map.get(n, dynamicProperty, false) as idValue,
    ↪ dynamicProperty
    MATCH (c:Counter {type: label})
    WHERE label in ["Patient","Room","Department","Episode","Medicine","Presc
    ↪ ription","Bill"] AND NOT
    ↪ idValue
    CALL apoc.cypher.run("
        MATCH (c:Counter {type: $label})
        RETURN c.count + 1 AS count
    ", {label: label}) YIELD value
    CALL apoc.create.setProperty(n, dynamicProperty, value.count) YIELD node
    SET n = node
    SET c.count = value.count
    ',
    {phase: 'afterAsync'}
);
```

Como podemos observar o procedimento seguido foi o seguinte:

1. Instalar o *trigger* denominado `dynamic_id_trigger` na base de dados **hospital**.
2. Descompactar os nós criados (*unwind*).

3. Obter a *label* que sofreu uma transação.

4. Verifica se o nó não possui a propriedade do identificador (`apoc.map.get(n, dynamicProperty, false)`).

5. Verifica se o nó é um dos nós onde existe uma consistência entre a *label* e o nome do ID, só estes nós é que serão alterados com este *trigger*.

6. Incrementa o contador correspondente à *label* e atribui o valor ao novo nó.

Para os restantes *triggers*, o processo é semelhante, a única diferença reside na forma como são identificados os nós e atribuídos os identificadores únicos. Cada *trigger* é configurado para operar em um tipo específico de nó, verificando se esse nó já possui um identificador associado. Se não, é gerado um novo identificador único com base no contador (*Counter*) correspondente ao tipo de nó. Esse identificador é então atribuído ao nó antes de atualizar o contador para refletir a adição do novo nó na base de dados. Assim, todos os *triggers* garantem que os nós criados recebam identificadores únicos de forma automática e consistente, mantendo a integridade dos dados na base de dados Neo4j.

5.2.2 Trigger proveniente da base de dados Oracle

O *trigger* proveniente da base de dados Oracle em questão têm como objetivo principal gerar automaticamente uma fatura (*Bill*) quando um paciente recebe uma alta hospitalar, ou seja, quando ao nó hospitalização (*hospitalization*), que é responsável por manter a informação sobre a hospitalização do paciente, é acrescentado o campo referente à data da alta hospitalar (`discharge_date`).

No ambiente Oracle, este *trigger* era implementado para monitorizar a inserção do campo de alta hospitalar e, ao detetar essa inserção, iniciava uma série de processos para calcular os custos associados ao paciente durante sua estadia. Esses custos incluíam taxas de quarto (`room_cost`), custos de exames laboratoriais (`test_cost`) e outros encargos médicos (`other_charges`). A soma total desses custos resultava na criação de uma nova fatura, que então era vinculada ao episódio de hospitalização do paciente.

Para a criação deste *trigger* foi necessário recorrer à propriedade `$assignedNodeProperties` da biblioteca *apoc.trigger* para monitorizar mudanças nos nós. Esta propriedade é um mapa que contém chaves para listas de mapas, onde cada mapa inclui a chave *(key)*, o valor antigo (*old*), o valor novo (*new*) e o nó (*node*). O *trigger* é ativado quando a propriedade `discharge_date` é atribuída a um nó *Hospitalization*.

O trigger obtido em Neo4j foi o seguinte:

```
1    CALL apoc.trigger.install(
2    'hospital',
3    'trg_generate_bill',
```

```
    '
    UNWIND keys(\$assignedNodeProperties) AS k
    WITH k
    WHERE k = \'discharge_date\'
    UNWIND $assignedNodeProperties[k] AS map
    WITH map.node AS h, map.old AS old, map.new AS new
    WHERE "Hospitalization" IN LABELS(h) AND old IS NULL AND new IS NOT NULL
    MATCH (e:Episode)-[:HAS_HOSPITALIZATION]->(h)
    MATCH (h)-[:IN_ROOM]->(r:Room)
    WITH e, h, COALESCE(r.room_cost, 0) AS v_room_cost
    OPTIONAL MATCH (e)-[:HAS_LAB_SCREENING]->(ls:LabScreening)
    WITH e, h, v_room_cost, COALESCE(SUM(ls.test_cost), 0) AS v_test_cost
    OPTIONAL MATCH
    ↪  (e)-[:HAS_PRESCRIPTION]->(p:Prescription)-[:PRESCRIBES]->(m:Medicine)
    WITH e, h, v_room_cost, v_test_cost, COALESCE(SUM(m.m_cost * p.dosage),
    ↪  0) AS v_other_charges
    WITH e, v_room_cost, v_test_cost, v_other_charges, (v_room_cost +
    ↪  v_test_cost + v_other_charges) AS v_total_cost
    MATCH (c:Counter {type: \'Bill\'})
   CALL apoc.cypher.run("
        MATCH (c:Counter {type: \'Bill\'})
        RETURN c.count + 1 AS count
    ",{}) YIELD value
     CREATE (b:Bill {
        id_bill: value.count,
        room_cost: v_room_cost,
        test_cost: v_test_cost,
        other_charges: v_other_charges,
        total: v_total_cost,
        id_episode: e.id_episode,
        registered_at: datetime(),
        payment_status: "PENDING"
    })
    MERGE (e)-[:HAS_BILL]->(b)
    SET c.count = value.count
',
    {phase: 'afterAsync'}
```

```
38 );
```

O procedimento levado a cabo pelo *trigger* apresentado acima é o seguinte:

1. Instalação do *trigger*.
2. Monitorização da Propriedade do nó de hospitalização, de forma a percecionar quando uum nó de hospitalização é atualizado com a inserção do campo *discharge_date* .
3. Cálculo de Custos associados com a hospitalização, recolhendo os custos do quarto associado à hospitalização (`_room_cost`), possíveis custos referentes à realização de testes laboratoriais (`v_test_cost`), e ainda possíveis despesas com a prescrição de medicação (`v_other_charges`). No final realizando o custo total da hospitalização, que estará presente na fatura, através da soma dos vários valores anteriormente referidos.
4. Criação de uma fatura (nó *Bill*), recorrendo às variáveis, `v_room_cost`, `v_test_cost` e `v_other_charges`.
5. Associação da fatura ao episódio correspondente à hospitalização.

Este trigger garante que a transição do OracleDB para o Neo4j mantenha a mesma eficiência e precisão na criação de faturas, assegurando a continuidade das operações financeiras no ambiente hospitalar de uma forma otimizada.

5.3 Exploração da Base de Dados

Após a migração para o Neo4J, conforme explicado anteriormente, iniciámos a exploração desta base de dados orientada a grafos. Para isso, utilizamos **queries** ou **expressões** tanto para passar um ID de uma tabela específica e obter a informação desejada, bem como, passar mais do que um parâmetro para obter dados de forma mais precisa e que englobassem mais do que um nó.

Por exemplo, para obter todas as informações de um paciente específico, incluindo detalhes sobre seus episódios médicos, consultas agendadas, faturas, hospitalizações e exames laboratoriais, utilizamos uma *query* que recebe como parâmetro o *id_patient*. A *query* percorre todos os nós que possuem estas informações apenas para o ID passado como parâmetro, fazendo uso das ligações entre eles para poder obter informação sobre cada um dos campos acima mencionados. Abaixo encontra-se a implementação desta *query*:

```
1 MATCH (p:Patient {id_patient: 89})
2 OPTIONAL MATCH (p)-[:HAS_EPISODE]->(e:Episode)
3 OPTIONAL MATCH (e)-[:HAS_APPOINTMENT]->(appointment:Appointment)
4 OPTIONAL MATCH (e)-[:HAS_BILL]->(bill:Bill)
5 OPTIONAL MATCH (e)-[:HAS_HOSPITALIZATION]->(hospitalization:Hospitalization)
6 OPTIONAL MATCH (e)-[:HAS_LAB_SCREENING]->(lab:LabScreening)
```

```
7  RETURN p,
8         COLLECT(DISTINCT appointment) AS appointments,
9         COLLECT(DISTINCT bill) AS bills,
10        COLLECT(DISTINCT hospitalization) AS hospitalizations,
11        COLLECT(DISTINCT lab) AS labScreenings
```

Analisando a *query* de forma concreta, temos:

```
1  MATCH (p:Patient {id_patient: 89})
```

Encontra o nó Patient com a campo *id_patient* igual a 89.

```
1  OPTIONAL MATCH (p)-[:HAS_EPISODE]->(e:Episode)
```

Realiza um *match* opcional (que pode ou não existir) para encontrar todos os episódios médicos (*Episode*) associados ao paciente. A relação *HAS_EPISODE* conecta o paciente aos seus episódios médicos.

```
1  OPTIONAL MATCH (e)-[:HAS_APPOINTMENT]->(appointment:Appointment)
```

Realiza um *match* opcional para encontrar todas as consultas (*Appointment*) associadas aos episódios médicos do paciente. A relação *HAS_APPOINTMENT* conecta os episódios médicos às suas consultas.

```
1  OPTIONAL MATCH (e)-[:HAS_BILL]->(bill:Bill)
```

Realiza um *match* opcional para encontrar todas as faturas (*Bill*) associadas aos episódios médicos do paciente. A relação *HAS_BILL* conecta os episódios médicos às suas faturas.

```
1  OPTIONAL MATCH (e)-[:HAS_HOSPITALIZATION]->(hospitalization:Hospitalization)
```

Realiza um *match* opcional para encontrar todas as hospitalizações (*Hospitalization*) associadas aos episódios médicos do paciente. A relação *HAS_HOSPITALIZATION* conecta os episódios médicos às suas hospitalizações.

```
1  OPTIONAL MATCH (e)-[:HAS_LAB_SCREENING]->(lab:LabScreening)
```

Realiza um *match* opcional para encontrar todos os exames laboratoriais (*LabScreening*) associados aos episódios médicos do paciente. A relação *HAS_LAB_SCREENING* conecta os episódios médicos aos seus exames laboratoriais.

É importante mencionar que a utilização de uma base de dados orientada a grafos, como o Neo4j, oferece várias vantagens para o bom desempenho da *query* fornecida. Vantagens como a **Consulta**

Rápida de Relacionamentos que envolvem a navegação através de relações entre entidades, faz com que esta *query* que envolve múltiplos relacionamentos (paciente para episódios, episódios para consultas, faturas, hospitalizações e exames), seja executada de forma muito eficiente. Outra vantagem que é importante mencionar é a **Travessia de Grafo** onde o Neo4J utiliza técnicas de travessia de grafo que permitem encontrar e retornar dados relacionados de forma rápida, mesmo com grandes volumes de dados.

5.3.1 Operações Básicas em Neo4J

Numa base de dados NoSQL como o Neo4J, não é necessário implementar funções separadas para realizar operações básicas como inserir, apagar ou atualizar nós. O Neo4J fornece métodos internos para lidar diretamente com estas operações. Aqui está um breve resumo de como pode realizar estas operações no Neo4J:

- **Inserir Nós e Relacionamentos:** Para inserir um nó ou um relacionamento, pode usar-se o comando `CREATE`.
- **Apagar Nós e Relacionamentos:** Para apagar nós ou relacionamentos, pode usar-se o comando `DELETE`.
- **Atualizar Nós e Relacionamentos:** Para atualizar propriedades de nós ou relacionamentos, pode usar o comando `SET`. Para remover propriedades, pode usar o comando `REMOVE`.

O Neo4J, sendo uma base de dados NoSQL, é projetado para manipular dados de forma mais flexível e direta, eliminando a necessidade de mecanismos de controlo adicionais nas operações de **Criação** (*CREATE*), **Atualização** (*UPDATE*) e **Remoção** (*DELETE*).

5.3.2 Visão Paciente

Buscar todos os pacientes: Esta query retorna todos os pacientes registados na base de dados. Cada paciente é representado por um nó `Patient`.

Buscar paciente por ID: Retorna o nó `Patient` que tem um ID específico. Utiliza a propriedade `id_patient` para encontrar o paciente desejado.

Buscar historial médico para um dado ID: Retorna o nó `Patient` e todos os nós `MedicalHistory` relacionados a ele através da relação `HAS_MEDICAL_HISTORY`. Cada nó `MedicalHistory` contém informações sobre condições médicas passadas do paciente.

Buscar seguro para um dado ID: Retorna os detalhes do seguro do paciente específico. Encontra o nó `Patient` com o ID fornecido e segue a relação `HAS_INSURANCE` para obter o nó `Insurance` associado.

Buscar contacto de emergência para um dado ID: Retorna os detalhes do contacto de emergência do paciente específico. Encontra o nó `Patient` com o ID fornecido e segue a relação `HAS_EMERGENCY_CONTACT` para obter o nó `EmergencyContact` associado.

Buscar pacientes por tipo de sangue: Retorna todos os pacientes com um tipo de sangue específico. Utiliza a propriedade `blood_type` do nó `Patient` para encontrar todos os pacientes com o tipo sanguíneo fornecido.

Buscar pacientes por género: Retorna todos os pacientes de um género específico (masculino ou feminino). Utiliza a propriedade `gender` do nó `Patient` para filtrar os pacientes pelo género.

Buscar pacientes pela condição médica: Retorna todos os pacientes que têm uma condição médica específica. Encontra todos os nós `Patient` que têm uma relação `HAS_MEDICAL_HISTORY` com um nó `MedicalHistory` onde a condição corresponde à fornecida.

Buscar todos os tipos de relações em contactos de emergência: Retorna todos os tipos de relações registadas nos nós `EmergencyContact`. Cada relação descreve a natureza do contacto de emergência com o paciente (por exemplo, pai, mãe).

Buscar todos os tipos de provedores de seguro: Retorna todos os provedores de seguro disponíveis na base de dados. Cada provedor é identificado pela propriedade `provider`.

Buscar todos os tipos de planos de seguro: Retorna todos os diferentes planos de seguro registados nos nós `Insurance`. Cada plano é identificado pela propriedade `insurance_plan`.

Buscar todos os tipos de cobertura: Retorna todos os diferentes tipos de cobertura registados nos nós `Insurance`. Cada tipo de cobertura é identificado pela propriedade `coverage`.

Buscar historial médico: Retorna todos os registros de historial médico na base de dados. Cada registro é representado por um nó `MedicalHistory`.

Buscar seguro: Retorna todos os registros de seguro na base de dados. Cada registro é representado por um nó `Insurance`.

Buscar contacto de emergência: Retorna todos os registros de contactos de emergência na base de dados. Cada registro é representado por um nó `EmergencyContact`.

Buscar todos os tipos de condições médicas: Retorna todos os diferentes tipos de condições médicas registadas nos nós `MedicalHistory`. Cada condição é identificada pela propriedade `condition`.

Buscar todos os tipos sanguíneos: Retorna todos os diferentes tipos sanguíneos registados nos nós `Patient`. Cada tipo sanguíneo é identificado pela propriedade `blood_type`.

Buscar número de pacientes para cada tipo sanguíneo: Retorna o número de pacientes para cada tipo sanguíneo, ordenado pela quantidade de pacientes. Conta quantos nós `Patient` existem para cada valor de `blood_type`.

Buscar número de pacientes para cada condição médica: Retorna o número de pacientes

para cada condição médica, ordenado pela quantidade de pacientes. Conta quantos nós `Patient` têm uma relação `HAS_MEDICAL_HISTORY` com um nó `MedicalHistory` para cada valor de `condition`.

Buscar pacientes por provedor de seguro específico: Retorna todos os pacientes que têm seguro com um provedor específico. Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde o provedor corresponde ao fornecido.

Buscar pacientes por plano de seguro específico: Retorna todos os pacientes que têm um plano de seguro específico. Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde o plano corresponde ao fornecido.

Buscar pacientes por cobertura específica: Retorna todos os pacientes que têm uma cobertura de seguro específica. Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde a cobertura corresponde à fornecida.

Buscar pacientes dentro de um intervalo de idades: Retorna todos os pacientes nascidos entre datas específicas. Utiliza a propriedade `birthday` do nó `Patient` para filtrar os pacientes dentro do intervalo de datas fornecido.

Buscar pacientes com cobertura de maternidade: Retorna todos os pacientes que têm cobertura de maternidade no seguro. Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde a cobertura de maternidade é verdadeira.

Buscar pacientes com cobertura dental: Retorna todos os pacientes que têm cobertura dental no seguro. Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde a cobertura dental é verdadeira.

Buscar pacientes com cobertura óptica: Retorna todos os pacientes que têm cobertura óptica no seguro. Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde a cobertura óptica é verdadeira.

Buscar pacientes com várias combinações de cobertura: Retorna pacientes que têm várias combinações de coberturas (por exemplo, dental e óptica). Encontra os nós `Patient` que têm uma relação `HAS_INSURANCE` com um nó `Insurance` onde múltiplas coberturas são verdadeiras.

Buscar pacientes por relação de contacto de emergência: Retorna todos os pacientes cujo contacto de emergência tem uma relação específica (por exemplo, pai). Encontra os nós `Patient` que têm uma relação `HAS_EMERGENCY_CONTACT` com um nó `EmergencyContact` onde a relação corresponde à fornecida.

Buscar paciente pelo primeiro e último nome: Retorna o paciente com um primeiro e último nome específico. Utiliza as propriedades `patient_fname` e `patient_lname` do nó `Patient` para encontrar o paciente.

Buscar paciente pelo número de telefone: Retorna o paciente com um número de telefone específico. Utiliza a propriedade `phone` do nó `Patient` para encontrar o paciente.

Contar o número de pacientes: Retorna a contagem total de pacientes registados na base de dados. Conta todos os nós `Patient`.

Buscar pacientes com um contacto de emergência específico: Retorna todos os pacientes cujo contacto de emergência é uma pessoa específica. Encontra os nós `Patient` que têm uma relação `HAS_EMERGENCY_CONTACT` com um nó `EmergencyContact` onde o nome do contacto corresponde ao fornecido.

Buscar pacientes pelo ID do historial médico: Retorna todos os pacientes que têm um historial médico com um ID específico. Encontra os nós `Patient` que têm uma relação `HAS_MEDICAL_HISTORY` com um nó `MedicalHistory` onde o ID do historial corresponde ao fornecido.

5.3.3 Visão Staff

Buscar toda a Informação de um Staff: Retorna todas as informações do funcionário cujo ID é fornecido. Utiliza a propriedade `id_emp` para encontrar o funcionário desejado.

Buscar o Department para um dado ID: Retorna o funcionário com um ID específico e o departamento em que ele trabalha. Também pode buscar diretamente um departamento pelo seu ID.

Buscar toda a informação das Enfermeiras: Retorna todas as informações de todos os funcionários cujo papel é *NURSE*. Utiliza a propriedade `role` para filtrar os funcionários.

Buscar toda a informação dos Médicos: Retorna todas as informações de todos os funcionários cujo papel é *DOCTOR*. Utiliza a propriedade `role` para filtrar os funcionários.

Buscar toda a informação dos Técnicos: Retorna todas as informações de todos os funcionários cujo papel é *TECHNICIAN*. Utiliza a propriedade `role` para filtrar os funcionários.

Buscar quantos Enfermeiros existem: Retorna a contagem de todos os funcionários cujo papel é *NURSE*. Utiliza a propriedade `role` para contar os funcionários.

Buscar quantos Doutores existem: Retorna a contagem de todos os funcionários cujo papel é *DOCTOR*. Utiliza a propriedade `role` para contar os funcionários.

Buscar quantos Técnicos existem: Retorna a contagem de todos os funcionários cujo papel é *TECHNICIAN*. Utiliza a propriedade `role` para contar os funcionários.

Buscar quantos enfermeiros, doutores e técnicos existem: Retorna a contagem de funcionários agrupados por seus papéis (*NURSE*, *DOCTOR*, *TECHNICIAN*). Utiliza a propriedade `role` para agrupar e contar os funcionários.

Buscar quantos Departamentos existem: Retorna a contagem de todos os departamentos na base de dados. Conta todos os nós `Department`.

Buscar Staff por Data de Admissão: Retorna todos os funcionários que foram admitidos na data fornecida. Utiliza a propriedade `date_joining` para filtrar os funcionários.

Buscar Staff por Data de Separação: Retorna todos os funcionários que se separaram na data fornecida. Utiliza a propriedade `date_separation` para filtrar os funcionários.

Buscar Staff Ativo ou Inativo: Retorna todos os funcionários que estão ativos ou inativos. Utiliza a propriedade `is_active_status` para filtrar os funcionários.

Qualificações de um Doctor por ID: Retorna as qualificações do doutor cujo ID é fornecido. Utiliza as propriedades `role` e `id_emp` para encontrar o doutor.

Todos os tipos de Qualificações: Retorna todas as qualificações distintas dos doutores. Utiliza a propriedade `qualification` para listar as qualificações únicas.

Número de Empregados por Departamento: Retorna a contagem de funcionários por departamento. Utiliza a relação `WORKS_IN` para agrupar e contar os funcionários por departamento.

Enfermeiros por Departamento: Retorna a contagem de enfermeiros por departamento. Utiliza a propriedade `role` e a relação `WORKS_IN` para agrupar e contar os enfermeiros por departamento.

Número de Doutores por Departamento: Retorna a contagem de doutores por departamento. Utiliza a propriedade `role` e a relação `WORKS_IN` para agrupar e contar os doutores por departamento.

Número de Técnicos por Departamento: Retorna a contagem de técnicos por departamento. Utiliza a propriedade `role` e a relação `WORKS_IN` para agrupar e contar os técnicos por departamento.

Contar Quantos Staff Estão Ativos: Retorna a contagem de todos os funcionários que estão ativos. Utiliza a propriedade `is_active_status` para contar os funcionários ativos.

Contar Quantos Staff não estão Ativos: Retorna a contagem de todos os funcionários que não estão ativos. Utiliza a propriedade `is_active_status` para contar os funcionários inativos.

Buscar Staff pelo Primeiro Nome e Sobrenome: Retorna o funcionário cujo primeiro nome e sobrenome correspondem aos fornecidos. Utiliza as propriedades `emp_fname` e `emp_lname` para encontrar o funcionário.

Buscar Staff pelo Email: Retorna o funcionário cujo email corresponde ao fornecido. Utiliza a propriedade `email` para encontrar o funcionário.

Contar o Número Total de Staff: Retorna a contagem total de funcionários registados na base de dados. Conta todos os nós `Staff`.

Buscar Staff pelo SSN: Retorna o funcionário cujo número de segurança social (SSN) corresponde ao fornecido. Utiliza a propriedade `ssn` para encontrar o funcionário.

5.3.4 Visão Episodes

Buscar toda a informação de um Episódio por ID: Retorna todas as informações do episódio cujo ID é fornecido. Utiliza a propriedade `id_episode` para encontrar o episódio desejado.

Buscar toda a informação sobre Prescrições Hospitalares: Retorna todas as informações de todas as prescrições registadas na base de dados.

Quantos episódios para um dado paciente: Retorna a contagem de episódios associados a um paciente específico. Utiliza a relação `HAS_EPISODE` para contar os episódios de um paciente.

Buscar as prescrições para um dado ID de episódio: Retorna todas as prescrições associadas a um episódio específico. Utiliza a relação `HAS_PRESCRIPTION` para obter as prescrições de um episódio.

Buscar bill para um dado ID de episódio: Retorna todas as contas associadas a um episódio específico. Utiliza a relação `HAS_BILL` para obter as contas de um episódio.

Buscar bill por ID de conta: Retorna a conta cujo ID é fornecido. Utiliza a propriedade `id_bill` para encontrar a conta desejada.

Buscar exames laboratoriais para um dado ID de episódio: Retorna todos os exames laboratoriais associados a um episódio específico. Utiliza a relação `HAS_LAB_SCREENING` para obter os exames de um episódio.

Buscar exame laboratorial por ID de laboratório: Retorna o exame laboratorial cujo ID é fornecido. Utiliza a propriedade `id_lab` para encontrar o exame desejado.

Buscar hospitalização para um dado ID de episódio: Retorna todas as hospitalizações associadas a um episódio específico. Utiliza a relação `HAS_HOSPITALIZATION` para obter as hospitalizações de um episódio.

Buscar sala por ID de sala: Retorna a sala cujo ID é fornecido. Utiliza a propriedade `id_room` para encontrar a sala desejada.

Buscar prescrições por ID de paciente: Retorna todas as prescrições associadas a um paciente específico. Utiliza a propriedade `patient_id` e a relação `HAS_PRESCRIPTION` para obter as prescrições do paciente.

Buscar bills por ID de paciente: Retorna todas as contas associadas a um paciente específico. Utiliza a propriedade `patient_id` e a relação `HAS_BILL` para obter as contas do paciente.

Buscar exames laboratoriais por ID de paciente: Retorna todos os exames laboratoriais associados a um paciente específico. Utiliza a propriedade `patient_id` e a relação `HAS_LAB_SCREENING` para obter os exames do paciente.

Buscar hospitalização por ID de paciente: Retorna todas as hospitalizações associadas a um paciente específico. Utiliza a propriedade `patient_id` e a relação `HAS_HOSPITALIZATION` para

obter as hospitalizações do paciente.

Buscar sala por ID de paciente: Retorna todas as salas associadas a um paciente específico. Utiliza a propriedade `patient_id`, a relação `HAS_HOSPITALIZATION`, e a relação `IN_ROOM` para obter as salas do paciente.

Buscar medicamento por ID de paciente: Retorna todos os medicamentos associados a um paciente específico. Utiliza a propriedade `patient_id`, a relação `HAS_PRESCRIPTION`, e a relação `PRESCRIBES` para obter os medicamentos do paciente.

Buscar prescrições por ID de paciente e data: Retorna todas as prescrições associadas a um paciente específico e uma data específica. Utiliza a propriedade `patient_id`, a relação `HAS_PRESCRIPTION`, e a propriedade `prescription_date` para filtrar as prescrições.

Buscar hospitalizações por ID de paciente e data de admissão: Retorna todas as hospitalizações associadas a um paciente específico e uma data de admissão específica. Utiliza a propriedade `patient_id`, a relação `HAS_HOSPITALIZATION`, e a propriedade `admission_date` para filtrar as hospitalizações.

Buscar todas as prescrições de um paciente específico: Retorna todas as prescrições de um paciente específico, independentemente do episódio. Utiliza a propriedade `patient_id` e a relação `HAS_PRESCRIPTION`.

Buscar todas as contas de um paciente específico: Retorna todas as contas de um paciente específico, independentemente do episódio. Utiliza a propriedade `patient_id` e a relação `HAS_BILL`.

Buscar todos os exames laboratoriais de um paciente específico: Retorna todos os exames laboratoriais de um paciente específico, independentemente do episódio. Utiliza a propriedade `patient_id` e a relação `HAS_LAB_SCREENING`.

Buscar todas as hospitalizações de um paciente específico: Retorna todas as hospitalizações de um paciente específico, independentemente do episódio. Utiliza a propriedade `patient_id` e a relação `HAS_HOSPITALIZATION`.

Buscar todas as salas de um paciente específico: Retorna todas as salas de um paciente específico, independentemente do episódio. Utiliza a propriedade `patient_id`, a relação `HAS_HOSPITALIZATION`, e a relação `IN_ROOM`.

Buscar todas as informações de um episódio específico: Retorna todas as informações detalhadas de um episódio específico, incluindo prescrições, contas, hospitalizações e exames laboratoriais. Utiliza a propriedade `id_episode` para encontrar o episódio e suas relações associadas.

Buscar todas as informações de todos os episódios: Retorna todas as informações detalhadas de todos os episódios, incluindo prescrições, contas, hospitalizações e exames laboratoriais. Utiliza as relações associadas aos episódios.

Buscar informações do médico responsável por um episódio específico: Retorna todas as informações do médico responsável por um episódio específico. Utiliza a relação `HAS_APPOINTMENT` e a relação `CONDUCTED_BY` para obter o médico.

Buscar todas as informações dos médicos que atenderam um paciente específico: Retorna todas as informações dos médicos que atenderam um paciente específico. Utiliza a relação `HAS_EPISODE`, a relação `HAS_APPOINTMENT`, e a relação `CONDUCTED_BY` para obter os médicos.

Buscar informações do técnico responsável por um episódio específico: Retorna todas as informações do técnico responsável por um episódio específico. Utiliza a relação `HAS_LAB_SCREENING` e a relação `PERFORMED_BY` para obter o técnico.

Buscar todas as informações dos técnicos que atenderam um paciente específico: Retorna todas as informações dos técnicos que atenderam um paciente específico. Utiliza a relação `HAS_EPISODE`, a relação `HAS_LAB_SCREENING`, e a relação `PERFORMED_BY` para obter os técnicos.

Retornar informações de um paciente específico dado um ID de paciente: Retorna todas as informações de um paciente específico, incluindo todos os episódios, consultas, contas, hospitalizações e exames laboratoriais. Utiliza a propriedade `id_patient` e as relações associadas.

Retornar informações de todos os pacientes: Retorna todas as informações detalhadas de todos os pacientes, incluindo todos os episódios, consultas, contas, hospitalizações e exames laboratoriais. Utiliza as relações associadas aos pacientes.

5.3.5 Visão Global

Listar todas as prescrições para um paciente específico: Retorna todas as prescrições associadas a um paciente específico. Utiliza a relação `HAS_EPISODE` para encontrar os episódios do paciente e a relação `HAS_PRESCRIPTION` para obter as prescrições desses episódios.

Listar os pacientes alocados a um específico quarto: Retorna todos os pacientes que estão alocados a um quarto específico. Utiliza a relação `IN_ROOM` para encontrar as hospitalizações no quarto e a relação `HAS_HOSPITALIZATION` para obter os episódios associados aos pacientes.

Listar todos os internamentos de um determinado paciente: Retorna todas as hospitalizações associadas a um paciente específico. Utiliza a relação `HAS_EPISODE` para encontrar os episódios do paciente e a relação `HAS_HOSPITALIZATION` para obter as hospitalizações desses episódios.

Listar hospitalizações por enfermeira responsável: Retorna todas as hospitalizações pelas quais uma enfermeira específica é responsável. Utiliza a relação `RESPONSIBLE_NURSE` para encontrar as hospitalizações associadas ao enfermeiro.

Listar todos os episódios médicos de um paciente específico: Retorna todos os episódios médicos associados a um paciente específico. Utiliza a relação `HAS_EPISODE` para encontrar os

episódios do paciente.

Listar episódios médicos por tipo de condição: Retorna todos os episódios médicos associados a pacientes com uma condição médica específica. Utiliza a relação `HAS_MEDICAL_HISTORY` para encontrar os pacientes com a condição e a relação `HAS_EPISODE` para obter os episódios desses pacientes.

Listar todos os episódios médicos tratados por um médico específico: Retorna todos os episódios médicos tratados por um médico específico. Utiliza a relação `CONDUCTED_BY` para encontrar as consultas conduzidas pelo médico e a relação `HAS_APPOINTMENT` para obter os episódios associados a essas consultas.

Listar todos os exames laboratoriais para um paciente específico: Retorna todos os exames laboratoriais associados a um paciente específico. Utiliza a relação `HAS_EPISODE` para encontrar os episódios do paciente e a relação `HAS_LAB_SCREENING` para obter os exames desses episódios.

Listar exames baseados no técnico responsável: Retorna todos os exames laboratoriais realizados por um técnico específico. Utiliza a relação `PERFORMED_BY` para encontrar os exames associados ao técnico.

Listar todas as faturas para um paciente específico: Retorna todas as faturas associadas a um paciente específico. Utiliza a relação `HAS_EPISODE` para encontrar os episódios do paciente e a relação `HAS_BILL` para obter as faturas desses episódios.

Listar todas as consultas agendadas para um paciente específico: Retorna todas as consultas agendadas associadas a um paciente específico. Utiliza a relação `HAS_EPISODE` para encontrar os episódios do paciente e a relação `HAS_APPOINTMENT` para obter as consultas desses episódios.

Listar consultas baseadas no médico responsável: Retorna todas as consultas conduzidas por um médico específico. Utiliza a relação `CONDUCTED_BY` para encontrar as consultas associadas ao médico.

Listar os Appointment para um dado Medico por dia: Retorna todas as consultas agendadas para um médico específico em uma data específica. Utiliza a relação `CONDUCTED_BY` e filtra pela data da consulta.

Buscar Appointment por data: Retorna todas as consultas agendadas em uma data específica. Utiliza a propriedade `appointment_date` para filtrar as consultas.

Buscar Appointment por data e depois por hora: Retorna todas as consultas agendadas em uma data e hora específicas. Utiliza as propriedades `appointment_date` e `appointment_time` para filtrar as consultas.

Lista todos os episódios e o respetivo paciente: Retorna todos os episódios médicos e os pacientes associados. Utiliza a relação `HAS_EPISODE` para obter os episódios e os pacientes

correspondentes.

Lista os médicos com mais consultas marcadas, com informação detalhada do paciente: Retorna os médicos com mais consultas marcadas, incluindo informações detalhadas dos pacientes atendidos. Utiliza a relação `CONDUCTED_BY` para encontrar as consultas, agrupa os médicos pelo número de consultas e coleta informações dos pacientes atendidos.

Capítulo 6

Análise Crítica

Ao longo deste projeto, realizámos a migração de dados de um sistema de gestão hospitalar baseado em Oracle SQL para sistemas NoSQL, nomeadamente MongoDB e Neo4j. Esta migração revelou-se essencial para responder às crescentes exigências e complexidades dos dados modernos, oferecendo maior flexibilidade e escalabilidade.

A exploração das bases de dados foi conduzida de forma meticulosa. Para cada uma das bases de dados — relacional, orientada a documentos ou orientada a grafos — desenvolvemos consultas, funções e outras operações de modo a serem aplicáveis no contexto real de um hospital. Em MongoDB, implementámos várias funções e expressões para obter informações detalhadas e gerais. Em Neo4j, as consultas foram otimizadas para obter dados complexos de forma eficiente.

Procurámos garantir que estas ferramentas pudessem ser usadas de forma eficaz tanto pelos pacientes, através de perfis personalizados, como pelo pessoal hospitalar para a gestão de informações e operações diárias. Adicionalmente, desenvolvemos funcionalidades específicas para a gestão de episódios médicos, tanto os que estão a decorrer como os já finalizados, assegurando uma administração eficiente e integrada dos dados hospitalares.

Este nível de detalhe na exploração permitiu-nos criar um sistema robusto e versátil, capaz de atender às necessidades de diversos utilizadores num ambiente hospitalar real.

Durante o desenvolvimento, enfrentámos vários desafios, como a manutenção da integridade dos dados e a adaptação de *triggers*. Os *triggers* desempenharam um papel crucial na manutenção da integridade dos dados e na automatização de processos. A criação de *triggers* em Neo4j, com o auxílio da biblioteca APOC, e em MongoDB, recorrendo à ferramenta Atlas, foi fundamental para a geração automática de identificadores únicos e a preservação da consistência dos dados.

Introduzimos também um nó auxiliar (*Counter*) e uma coleção auxiliar (*counters*) cuja principal função é armazenar o número máximo dos identificadores utilizados até ao momento. Estas duas abordagens facilitaram a geração de novos identificadores únicos para futuros elementos das nossas bases de dados. A gestão de datas foi abordada com cuidado para garantir a precisão e a consistência dos registos temporais nos diferentes sistemas de bases de dados.

Capítulo 7

Conclusão

O trabalho realizado sobre a migração e exploração de bases de dados NoSQL representou um avanço significativo no conhecimento e aplicação dessas tecnologias em ambientes hospitalares. A migração de um sistema baseado em Oracle SQL para MongoDB e Neo4j não só demonstrou a viabilidade e os benefícios dos sistemas NoSQL, como também ofereceu *insights* práticos sobre as estratégias e desafios enfrentados.

Observou-se que a flexibilidade e escalabilidade dos sistemas NoSQL proporcionaram uma gestão de dados mais eficiente, particularmente em consultas e operações complexas. Em relação ao MongoDB a utilização da ferramenta Atlas foi essencial para a criação de *triggers* essenciais ao nosso modelo de dados. Quanto a Neo4j a utilização da biblioteca APOC e a definição de *constraints* foram cruciais para manter a integridade dos dados durante a migração. Além disso, as consultas implementadas exibiram capacidades operacionais superiores em comparação com o sistema relacional original, facilitando a obtenção de informações detalhadas.

Para futuros trabalhos, recomenda-se a otimização contínua das consultas para aprimorar ainda mais o desempenho do sistema. A expansão das funcionalidades do sistema pode incluir a integração de novos tipos de dados, a implementação de novos *triggers*, e a melhoria das interfaces de consulta.

Esta abordagem não apenas reforça a versatilidade e a capacidade dos sistemas NoSQL, mas também destaca a importância de um contínuo processo de inovação e adaptação às demandas crescentes e complexas dos dados modernos, particularmente em contextos críticos como o hospitalar.

Apêndice A

Nodos desenvolvidos em Neo4J

A.1 Paciente

birthday	"1996-02-14"
blood_type	AB-
email	olivia.perez@example.com
gender	Female
id_patient	20
patient_fname	Olivia
patient_lname	Perez
phone	222-333-4444

Figura A.1: Exemplo de um nodo paciente

A.2 Seguro de Saúde

co_pay	20.0
coverage	Limited Coverage
dental	false
insurance_plan	Basic Plan
maternity	true
optical	false
policy_number	POL003
provider	DEF Insurance

Figura A.2: Exemplo de um nodo seguro de saúde

A.3 Histórico médico

condition	Osteoporosis
id_record	20
record_date	"2024-09-05"

Figura A.3: Exemplo de um nodo histórico médico

A.4 Contactos de Emergência

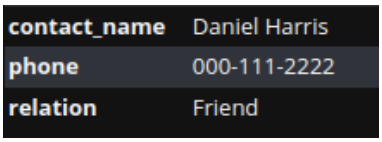


Figura A.4: Exemplo de um nodo contactos de emergência

A.5 Episódios Médicos

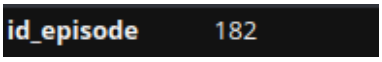


Figura A.5: Exemplo de um nodo episódios médicos

A.6 Faturas

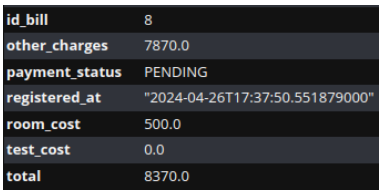


Figura A.6: Exemplo de um nodo faturas

A.7 Prescrição Médica

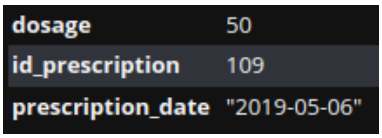


Figura A.7: Exemplo de um nodo prescrição médica

A.8 Medicamento

id_medicine	18
m_cost	20.0
m_name	Fluoxetine
m_quantity	20

Figura A.8: Exemplo de um nodo medicamento

A.9 Departamento

department_head	Ethan Brown
department_name	Gynecology
id_department	17

Figura A.9: Exemplo de um nodo departamento

A.10 Funcionários

address	"41829 Andrew Course Apt. 567
date_joining	"2019-07-10"
date_separation	"2022-01-05"
email	seanlyons@example.org
emp_fname	William
emp_lname	Grant
id_emp	13
is_active_status	false
qualification	PhD
role	DOCTOR
ssn	804408507

Figura A.10: Exemplo de um nodo funcionários

A.11 Consulta

appointment_date	"2020-12-28"
appointment_time	14:17
id_doctor	24
schedule_on	"2020-11-27"

Figura A.11: Exemplo de um nodo consulta

A.12 Testes Médicos

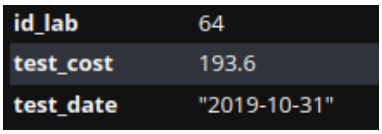


Figura A.12: Exemplo de um nodo testes médicos

A.13 Hospitalização

Figura A.13: Exemplo de um nodo hospitalização

A.14 Quarto

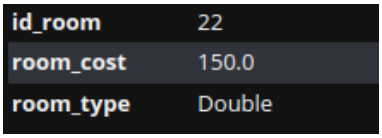


Figura A.14: Exemplo de um nodo quarto

A.15 Contador

Figura A.15: Exemplo de um nodo contador