



UNIVERSIDADE DO MINHO

MESTRADO EM ENGENHARIA INFORMÁTICA

Bases de Dados NoSQL  
Trabalho Prático

Rodrigo Pires Rodrigues (PG50726)  
Rui Guilherme Monteiro (PG50739)  
José João Gonçalves (PG50519)

Ano Letivo 2022/2023



# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Base de Dados Relacional - Oracle</b>	<b>3</b>
<b>3</b>	<b>Base de Dados Documental - MongoDB</b>	<b>4</b>
3.1	Estrutura do modelo . . . . .	4
3.2	Migração . . . . .	5
3.3	Pré-Processamento dos Dados . . . . .	5
<b>4</b>	<b>Base de Dados por Grafos - Neo4J</b>	<b>10</b>
4.1	Migração . . . . .	10
4.2	Pré-Processamento dos Dados . . . . .	10
<b>5</b>	<b>Queries</b>	<b>12</b>
5.1	Queries relativas a <i>Users</i> . . . . .	12
5.1.1	Listar todos os pedidos feitos por um utilizador . . . . .	12
5.1.2	Listar o utilizador que fez a compra mais cara . . . . .	12
5.1.3	Listar os utilizadores mais antigos da plataforma . . . . .	13
5.1.4	Listar a média de valor das encomendas de cada utilizador . . . . .	14
5.2	Queries relativas a <i>Orders</i> . . . . .	15
5.2.1	Listar o total de vendas por categoria de produto . . . . .	15
5.2.2	Listar o método de entrega preferido pelos clientes . . . . .	15
5.3	Queries relativas a <i>Employees</i> . . . . .	16
5.3.1	Listar o Funcionário mais antigo . . . . .	16
5.3.2	Listar o Funcionário com maior salário . . . . .	16
5.3.3	Listar o Funcionário que é <i>Manager</i> e tem mais subordinados . . . . .	17
5.4	Queries relativas a <i>Products</i> . . . . .	18
5.4.1	Encontrar os produtos mais populares com base no número de pedidos . . .	18
5.4.2	Listar os produtos que nunca foram vendidos . . . . .	19
<b>6</b>	<b>Análise crítica e comparação de modelos</b>	<b>21</b>
<b>7</b>	<b>Triggers</b>	<b>22</b>
7.1	Triggers MongoDB . . . . .	22
7.1.1	<i>Update</i> - Atualizar <i>department</i> ID do <i>employee</i> se o ID do departamento atualizar . . . . .	22
7.1.2	<i>Delete</i> - Eliminar todas as <i>Orders</i> de um <i>User</i> que foi apagado . . . . .	22
7.1.3	<i>Insert</i> - Se uma <i>order</i> for inserida com um <i>user</i> ID inexistente, inserir também o <i>user</i> . . . . .	22
7.2	Triggers Neo4J . . . . .	23
7.2.1	<i>Update</i> - updateEmployeeDepartmentId . . . . .	23
7.2.2	<i>Delete</i> - DeleteUserCascade . . . . .	24
7.2.3	<i>Insert</i> - insertStoreUser . . . . .	24
<b>8</b>	<b>Conclusão</b>	<b>26</b>

# 1 Introdução

O presente relatório tem como objetivo descrever o projeto de análise, planeamento e implementação de um sistema de base de dados relacional e dois sistemas de base de dados não relacionais, orientados a documentos e grafos, respetivamente. Este trabalho prático surge no âmbito da unidade curricular de Bases de Dados NoSQL e tem como objetivo desenvolver competências na utilização de diferentes paradigmas de bases de dados e na sua aplicação na conceção e implementação de sistemas. Para tal, será utilizada a base de dados relacional *Online Electronics Store*, que representa uma loja fictícia de eletrónicos online, e que inclui diversos objetos de base de dados, como tabelas, *views*, sequências, índices, *triggers*, funções e procedimentos.

As secções que se seguem darão lugar a uma análise detalhada, passo a passo, da base de dados relacional original bem como as medidas a tomar para ser possível trabalhar com os dados fornecidos e desenvolver os sistemas de bases de dados pretendidos. Serão também definidas e implementadas *queries* para demonstrar a operacionalidade dos sistemas implementados, bem como uma análise crítica comparando os modelos e funcionalidades agora implementadas com as disponibilizadas no sistema relacional fornecido.

## 2 Base de Dados Relacional - Oracle

Este trabalho prático teve por base um modelo lógico SQL de uma base de dados *Oracle*, representativo de uma loja fictícia de produtos eletrónicos online, e inclui diversos objetos de base de dados, como tabelas, *triggers*, funções, procedimentos, entre outros. Este modelo explicita a lógica de negócio do programa, os atributos de cada entidade e como se relacionam entre si. Através da sua análise foi possível observar como os dados estão organizados e como seria a melhor política de transição dos mesmos para modelos não relacionais.

Por análise do modelo de base de dados fornecido, é possível averiguar:

- A loja tem um sistema de registo e *login* para os seus clientes, o que implica que a informação pessoal do utilizador é armazenada na base de dados. Isso inclui dados como nome, endereço, contacto e informações de pagamento.
- A loja tem um sistema de categorias de produtos para ajudar os utilizadores a navegar pelo site e encontrar os produtos que desejam. Cada produto é atribuído a uma ou mais categorias.
- A loja tem um sistema de carrinho de compras que permite aos clientes adicionar produtos, bem como um sistema de pedidos que inclui informações sobre os produtos comprados, detalhes de pagamento e informações de envio.
- A loja possui diversos departamentos e tem funcionários atribuídos aos mesmos que, possivelmente podem necessitar de aceder à base de dados para acederem a informações relevantes para a execução das suas funções.
- A loja tem um sistema de gestão de *stock* que permite aos gerentes controlarem os níveis de quantidade dos produtos.

Com base nestas informações obtidas pela visão geral do modelo da base de dados, perspectiva-se uma solução de SGBD (Sistema de Base de Dados) que deve ser capaz de gerir dados relacionados aos utilizadores, produtos, promoções, pedidos, pagamentos, funcionários e gestão de *stock*. Havendo uma correta modelação das relações entre as tabelas, garante-se a integridade referencial dos dados e criam-se consultas eficientes para recuperar informações relevantes para várias operações comerciais. Além disso, a loja pode beneficiar de soluções de SGBD não relacionais como bases de dados documentais (*MongoDB*) ou de grafos (*Neo4J*) para armazenar informações sobre produtos, clientes e histórico de compras, tal como será analisado nas secções seguintes deste relatório.

### 3 Base de Dados Documental - MongoDB

Com base no resultado do passo anterior, o presente passo consiste em iniciar a migração para uma Base de Dados NoSQL de formato Documental.

Para tal, o modelo documental envolve a aplicação de técnicas que permitam escolher e representar a melhor organização em documentos dos dados referentes à loja de produtos eletrónicos.

#### 3.1 Estrutura do modelo

A partir do modelo da base de dados relacional chegou-se a um modelo documental que maximiza este paradigma.

De modo a que os dados possam ser representados em formato documental, foram aplicadas algumas técnicas de modo a juntar os dados das tabelas em coleções de dados que se relacionam e que podem tirar partido da proximidade de acesso.

Partindo deste novo formato e organização da informação, associa-se agora os dados da seguinte forma:

- Grupo 1: **USER**  
STORE\_USERS: Contém os utilizadores registados no site da loja.  
SHOPPING\_SESSION: Contém as sessões criadas pelos utilizadores.
  - CART\_ITEM: Contém os produtos adicionados ao carrinho pelo cliente numa determinada sessão.
- Grupo 2: **EMPLOYEES**  
EMPLOYEES: Contém os funcionários da loja.  
DEPARTMENTS: Departamentos dos funcionários internos da loja.  
EMPLOYEES\_ARCHIVE: Contém os dados de funcionários arquivados.
- Grupo 3: **ORDER**  
ORDER\_DETAILS: Detalhes do pedido do utilizador.  
ORDER\_ITEMS: Contém os produtos do pedido feito pelo utilizador.  
PAYMENT\_DETAILS: Contém os detalhes de pagamento do pedido.  
ADDRESSES: Contém as moradas de envio da encomenda.
- Grupo 4: **PRODUCT**  
PRODUCT\_CATEGORIES: Contém as categorias dos produtos.  
PRODUCT: Contém os produtos que a loja vende.  
DISCOUNT: Promoções ativas e expiradas na loja.  
STOCK: Stock de produtos.

Essencialmente existem quatro coleções principais que agregam os dados. O pensamento por de trás destas coleções passa por juntar na mesma coleção dados que se apresentam fortes relações entre si, tal como é possível observar no modelo de base de dados elaborado no enunciado deste trabalho prático para a base de dados relacional.

Assim, é natural e imediato notar que certas tabelas pertencem ao mesmo "meio" que outras tabelas, isto é, complementam-se. Analisando os grupos definidos para se transformarem em coleções, e começando por ordem, podemos destacar:

- No grupo/coleção 1 (USER) encontram-se as informações relativas aos utilizadores da loja de eletrónicos. Ao registo inicial dos *store\_users* faz-se um *append* de informações relativas às suas sessões na loja e, naturalmente essas sessões passarão a ter informação do carrinho de compras (*cart\_items*).

- No grupo/coleção 2 (EMPLOYEES) residem dados relativos aos trabalhadores da loja. Um ponto importante a destacar nesta coleção é o facto de permitir eliminar informação redundante na medida em que uma das transformações/tratamentos aplicados passou por, em vez de ter registos de *employees* separados dos registos de arquivos de *employees*, passa-se a ter apenas um registo de *employees* que serve ambos os propósitos e tem um campo adicional *state* que indica (*old*, *active*) para simbolizar se este se trata de um registo atualizado ou antigo. Adicionalmente, estes registos possuem também um campo indicativo do departamento onde o trabalhador colabora.
- No grupo/coleção 3 (ORDER) foram agrupados dados relativos a encomendas/pedidos, nomeadamente detalhes do pedido, respetivos itens, detalhes de pagamento e morada de entrega. Todas estas informações constam do mesmo documento, facilitando a obtenção de dados essenciais de um pedido todos em proximidade.
- No grupo/coleção 4 (PRODUCT) são armazenados dados relativos aos produtos com informação das respetivas categorias, descontos, stock e detalhes do produto.

Agrupar tabelas em coleções permite combinar entidades relacionadas num único documento. Isto simplifica a estrutura de dados, tornando-a mais fácil de entender e trabalhar e elimina a necessidade de operações de junção complexas que poderiam necessárias numa base de dados relacional. Outra das razões para a forma escolhida de organização de dados é **eficiência de leitura**, isto é, ao armazenar dados relacionados juntos, pode-se otimizar as operações de leitura. A recuperação de dados de uma única coleção é mais rápida em comparação com a obtenção de dados de várias tabelas e a sua junção. Isto é particularmente vantajoso ao lidar com grandes conjuntos de dados ou consultas complexas. Adicionalmente, a **desnormalização dos dados** permite realizar atualizações atómicas, o que significa que se podem atualizar vários campos relacionados num único documento, garantindo a consistência dos dados e evitando problemas que poderiam surgir da atualização de várias tabelas numa base de dados relacional.

Ainda assim é importante notar que a desnormalização vem com *trade-offs*, uma vez que pode aumentar a redundância dos dados e exigir esforço adicional para garantir a consistência dos mesmos. No entanto, considerou-se cuidadosamente os padrões de acesso e consultas possíveis de forma a projetar as coleções de acordo.

## 3.2 Migração

Desenvolveu-se um programa *migration\_mongodb.py* para efetuar a migração dos dados para a base de dados documental *MongoDB*. O programa efetua a conexão com a base de dados Oracle a partir da qual obterá os dados para enviar para o MongoDB em formato JSON.

Este programa foi criado para facilitar a implementação dos dados em MongoDB de forma a que seja possível uma transferência eficiente e que se adaptasse à alteração dos dados em Oracle.

## 3.3 Pré-Processamento dos Dados

Reunidos os requisitos estruturais, o próximo passo centra-se na modelação e processamento dos dados.

Para obter cada uma das tabelas provenientes do SQL Developer, recorreu-se à biblioteca *Cx-Oracle* para efetuar a conexão e posteriormente efetuaram-se queries para obter cada uma das tabelas em um *Dataframe pandas*.

Em seguida, de modo a tratar os dados e agrupar de acordo com o referido na Subsecção 3.1 foram efetuados os seguintes passos:

- **Grupo User:** Com vista a obter o formato/organização referido acima, foi realizada uma série de operações relacionadas à estruturação e armazenamento de dados. Primeiro, itera-se sobre cada linha do dataframe *store\_users*, que contém informações dos utilizadores da loja e cria-se um dicionário com as informações relevantes do utilizador para adicionar na coleção.

De seguida, é criado um dicionário chamado `sessions_por_user` para armazenar arrays de sessões de compra para cada utilizador com informação dos itens do carrinho de compras.

```
_id: ObjectId('646359a700ddf5bc502fa566')
user_ID: 1
user_firstName: "Jody"
user_middleName: ""
user_lastName: "Cabell"
user_phoneNumber: "256 375 7831"
user_email: "ccabell0@cbslocal.com"
user_username: "ccabell0"
user_password: "WEbtznzfbUDDW"
user_registeredAt: "2019-08-15 17:25:56"
▼ shopping_sessions: Array
  ▼ 0: Object
    session_ID: 1
    session_userID: 1
    session_createdAt: "2022-05-18 22:11:16"
    session_modifiedAt: "2022-05-19 04:18:28"
  ▼ cart_items: Array
    ▼ 0: Object
      cartItem_ID: 1
      cartItem_sessionID: 1
      cartItem_productID: 3
      cartItem_quantity: 1
      cartItem_createdAt: "2022-05-19 11:29:16"
      cartItem_modifiedAt: "2022-07-01 09:57:48"
    ► 1: Object
```

Figura 1: Documento JSON da coleção User

- **Grupo Employees:** De modo obter um documento no formato referido, começou-se por definir um dicionário chamado *employee\_doc* que armazena informações relacionadas a um funcionário, juntamente com um novo campo *employee\_state*, que ditará o estado *active* ou *old*. De modo a juntar as informações que estão presentes na outra tabela referida, iterou-se sobre o *Dataframe employeesArchive* e, iterar pelo `EMPLOYEES_ARCHIVE` e desconstruir em 2 registos (um com o estado "old", e o outro com o estado "active"). Caso o registo tenha presente informação no campo `OLD_EMPLOYEE_ID` dos arquivos, os dados antigos do funcionário são escritos no novo campo criado neste documento final. De igual modo é feito para o `NEW_EMPLOYEE_ID`.

Em seguida, para adicionar o campo referente ao departamento do funcionário, é percorrido o conjunto dos funcionários ativos e atualizado o seu campo *dept\_info* com a informação presente na tabela dos *DEPARTMENTS*.

Deste modo, apresenta-se um exemplo da representação final do novo formato documental:

```

    _id: ObjectId('6466722cace66cc7fc22f282')
    emp_ID: 2
    emp_firstName: "Brittney"
    emp_MiddleName: "Leonidas"
    emp_LastName: "Dimitriou"
    emp_dateOfBirth: "1976-07-21 00:00:00"
    emp_departmentID: 2
    emp_hireDate: "2018-01-08 00:00:00"
    emp_salary: 11800
    emp_phoneNumber: "604 235 4231"
    emp_Email: "ldimitriou1@icio.us"
    emp_ssnNumber: "825-48-6752"
    emp_managerID: 1
    employee_state: "active"
  ▼ department_info: Object
      department_id: 2
      department_name: "Development"
      manager_id: 2
      department_description: "Developing of the store's website"

```

Figura 2: Documento JSON da coleção Employees

- **Grupo Order:** É realizada uma série de operações relacionadas à estruturação e armazenamento de dados de pedidos, detalhes do pedido, itens do pedido, detalhes de pagamento e endereços de entrega. Primeiro, prepara-se os documentos dos pedido, construindo os objetos JSON dos pedidos a partir da extração de dados do dataframe `order_details`. De seguida é criado um dicionário chamado `items_por_order` para armazenar arrays de itens para cada pedido. Em terceiro, itera-se sobre cada linha do dataframe `payment_details`, que contém informações dos detalhes de pagamento para construir objetos JSON para posterior atualização dos detalhes de pagamento nos pedidos. Finalmente, procede-se à atualização dos endereços de entrega nos pedidos iterando em cada linha do dataframe `addresses`, que contém informações dos endereços de entrega.



Figura 3: Documento JSON da coleção Order

- **Grupo Product:** De modo a obter um formato mono-documental para as informações do produto, juntamente com a sua categoria, stock e desconto, foram efetuadas algumas alterações, à semelhança do anterior.

Inicialmente construiu-se o dicionário com os dados dos produtos e enviou-se para o MongoDB, passando depois a efetuar um *update* com a informação em novos campos denominados *product.category*, *product.discount* e *product.stock*. Deste modo, para cada registo dos produtos, é feito (à semelhança do anterior) um *update* do documento com os valores correspondentes ao ID que ambas as tabelas partilham. Isto é, uma vez que, por exemplo, as tabelas Product e Stock partilham o campo *PRODUCT\_ID*, é feita uma pesquisa por esse campo e, caso coincida, é atualizado o documento desse produto com os dados correspondentes ao stock, caso contrário fica vazio.

Segue um exemplo do documento final para o grupo Product:



```
_id: ObjectId('646359a700ddf5bc502fa583')
product_ID: 1
product_name: "ASUS X515-BQ26W 8GB RAM 256GB SSD"
product_categoryID: 1
product_sku: "HR278YRE"
product_price: 2399
product_discountId: 1
product_createdAt: "2022-07-01 09:57:48"
product_lastModified: "2022-07-12 23:58:55"
▼ product_category: Object
  category_ID: 1
  category_name: "Computers, Laptops and Consoles"
▼ product_stock: Object
  quantity: 47
  max_stock_quantity: 120
  unit: "1PCS"
▼ product_discount: Object
  discount_id: 1
  discount_name: "Hits of the Week"
  discount_desc: "Weekly discount"
  discount_percent: 15
  discount_is_active_status: "Y"
  discount_createdAt: 2022-03-10T02:27:51.000+00:00
  discount_modifiedAt: "2022-07-01 09:57:48"
```

Figura 4: Documento JSON da coleção *Product*

## 4 Base de Dados por Grafos - Neo4J

Partindo novamente do modelo relacional inicial, pretende-se agora representar os dados numa nova ferramenta NoSQL, desta vez uma Base de Dados orientada por Grafos, o Neo4J.

### 4.1 Migração

Para desenvolver este processo, foi desenvolvido um *script migration\_neo4j.ipynb* que converte tabelas em nodos além de criar as relações entre os nodos. O programa efetua a conexão com a base de dados Oracle, a partir da qual obtém os dados para enviar ao Neo4J através de *queries* Cypher armazenadas num ficheiro de texto.

### 4.2 Pré-Processamento dos Dados

Tendo em vista a fase de migração de dados, começou-se por definir os nodos que seriam utilizados. Deste modo, passa-se a expor e explicar cada um dos nodos definidos:

- **Store.Users** Os utilizadores da Loja de Produtos eletrónicos.
- **Product** O conjunto dos produtos disponíveis na loja, aos quais foi adicionada a informação relativa ao stock e a categoria correspondente, presentes nas tabelas SQL denominadas *STOCK* e *PRODUCT.CATEGORIES*
- **Discount** Os descontos disponíveis e expirados na loja. Uma vez que nem todos os produtos contêm desconto, optou-se por desenvolver um nodo à parte para este efeito, sendo assim utilizada uma relação entre o produto e o desconto em caso afirmativo, tal como se explica posteriormente.
- **Cart\_Item** Conjunto de produtos adicionados ao carrinho de compras
- **Shopping\_Session** Informação correspondente a uma sessão de um utilizador.
- **Order\_Details** Detalhes de uma encomenda realizada por um determinado utilizador
- **Order\_Items** Items que constituem uma encomenda
- **Payment\_Details** Detalhes de pagamento associados a uma compra
- **Employees** Informação correspondente a um funcionário, tanto a nível atual como a nível histórico, sendo que um nodo contém a informação da tabela SQL denominada *EMPLOYEES\_ARCHIVE*
- **Departments** Conjunto dos vários departamentos que a loja contém
- **Addresses** Informação relativa à morada dos vários clientes

Decidiu-se considerar a tabela *STOCK* como uma propriedade do nodo *Product*, assim através da relação entre os seus *ProductIDs*, foi possível acrescentada essa informação.

Também foi acrescentada a tabela *Product\_Categories* de modo a simplificar o modelo de dados, reduzindo o número de nodos e relações, o que faz com que a utilização de *queries* seja mais direta, visto o número de categorias ser pequeno. Esta decisão também melhora a eficiência das *queries* que envolvam informação relativa ao produto e à categoria, visto não ser necessário atravessar relações. Por último, armazenar a informação das categorias diretamente no nodo dos produtos reduz a redundância de dados e potencialmente simplifica a manutenção dos dados.

Relativamente à tabela *Employees\_Archive* que se encontra presente na base de dados Oracle, esta armazena dados históricos dos funcionários, de modo a simplificar a base de dados de grafos, reduzindo o número de nodos, foi acrescentada a informação histórica ao nodo dos *Employees*.

De igual modo, foram desenvolvidos os relacionamentos, que unem pares de nodos, permitindo estabelecer uma ligação entre eles. Em seguida, demonstra-se a estrutura utilizada:

- **WORKS\_IN** Relaciona um *Employee* e um *Department*, indicando para cada funcionário qual o departamento em que ele desempenha as suas funções;
- **BELONGS\_TO** Relaciona uma *Shopping\_Session* e um *User*, sendo que cada user tem uma sessão associada;
- **PLACED\_BY** Relaciona uma *Order\_Details* com um *User*, indicando para cada uma das encomendas qual o utilizador que a colocou;
- **RELATES\_TO** Relaciona os *Payment\_Details* e a *Order\_Details*, indicando a que encomenda se referem determinados pagamentos;

## 5 Queries

Nesta secção será apresentado um conjunto de *queries* estratégicas, escolhidas de forma a tirar partido da implementação e funcionalidades da base de dados Oracle, da base de dados MongoDB e, por fim, da bases de dados Neo4J permitindo demonstrar a operacionalidade do sistema de bases de dados e a comparação da performance entre cada uma delas.

### 5.1 Queries relativas a Users

#### 5.1.1 Listar todos os pedidos feitos por um utilizador

##### Oracle

```
SELECT USER_ID, COUNT(*) AS NUM_ORDERS
FROM ORDER_DETAILS
GROUP BY USER_ID;
```

##### MongoDB

```
db.user.aggregate([
  {
    "$lookup": {
      "from": "order",
      "localField": "user_ID",
      "foreignField": "order_userID",
      "as": "user_orders"
    }
  },
  {
    "$project": {
      "user_id": "$user_ID",
      "user_orders": "$user_orders"
    }
  }
])
```

##### Neo4J

```
MATCH (u: Store_User)<-[:PLACED_BY]-(o: Order_Details)
RETURN u.id AS user_id, COLLECT(o.id) AS user_orders
```

#### 5.1.2 Listar o utilizador que fez a compra mais cara

##### Oracle

```
SELECT u.user_id, u.username, MAX(od.total) AS max_total_value
FROM store_users u
JOIN shopping_session ss ON u.user_id = ss.user_id
JOIN order_details od ON ss.session_id = od.user_id
GROUP BY u.user_id, u.username
HAVING MAX(od.total) = (SELECT MAX(total) FROM order_details)
```

##### MongoDB

```
db.order.aggregate([
  {
    "$lookup": {
```

```

        "from": "user",
        "localField": "order_userID",
        "foreignField": "user_ID",
        "as": "user"
    }
},
{
    "$unwind": "$user"
},
{
    "$sort": {
        "order_total": -1
    }
},
{
    "$limit": 1
},
{
    "$project": {
        "_id": "$user._id",
        "username": "$user.user_username",
        "maxOrderTotal": "$order_total"
    }
}
}
])

```

Nesta consulta, usamos a agregação do MongoDB para realizar as operações necessárias. Primeiro, realizamos um lookup para unir a coleção de pedidos (order) com a coleção de utilizadores (user) com base no campo order.userID e user\_ID, respetivamente. Em seguida, desdobramos o documento resultante do lookup. Depois, classificamos os documentos com base no campo order.total em ordem decrescente. Limitamos o resultado a apenas um documento, que será o utilizador com a compra mais cara. Por fim, projetamos os campos desejados, incluindo \_id, username e maxOrderTotal.

#### Neo4J

```

MATCH (o:Order_Details)-[:PLACED_BY]->(u:Store_User)
RETURN u.id AS userID, u.Username AS username, o.total AS maxOrderTotal
ORDER BY o.total DESC
LIMIT 1

```

### 5.1.3 Listar os utilizadores mais antigos da plataforma

#### Oracle

```

SELECT * FROM store_users
ORDER BY registered_at ASC;

```

#### MongoDB

```

db.users.find().sort({ user_registered_at: 1 });

```

#### Neo4J

```

MATCH (u:Store_User)
RETURN u.First_name, u.Registered_at
ORDER BY u.Registered_at ASC

```

#### 5.1.4 Listar a média de valor das encomendas de cada utilizador

##### Oracle

```
SELECT u.user_id, u.username, AVG(o.total) AS average_order_total
FROM store_users u
JOIN order_details o ON u.user_id = o.user_id
GROUP BY u.user_id, u.username;
```

##### MongoDB

```
db.order.aggregate([
  {
    $lookup: {
      from: "user",
      localField: "order_userID",
      foreignField: "user_ID",
      as: "user"
    }
  },
  {
    $unwind: "$user"
  },
  {
    $group: {
      _id: "$user.user_ID",
      average_order_value: { $avg: "$order_total" }
    }
  }
])
```

Esta consulta realiza um "lookup" entre as coleções "order" e "user" com base no campo "order\_userID" e "user\_ID", respectivamente. Em seguida, realiza um agrupamento por "user\_ID" e calcula a média do valor das encomendas usando o operador de agregação "\$avg" no campo "order\_total". O resultado será uma lista de utilizadores com a média do valor das encomendas.

##### Neo4J

```
MATCH (o:Order_Details)-[:PLACED_BY]->(u:Store_User)
RETURN u.Username, AVG(o.total) AS Average_Order_Total
```

## 5.2 *Queries* relativas a *Orders*

### 5.2.1 Listar o total de vendas por categoria de produto

#### Oracle

```
SELECT pc.category_name, COUNT(DISTINCT od.order_details_id) AS total_sales
FROM product_categories pc
JOIN product p ON pc.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN order_details od ON oi.order_details_id = od.order_details_id
GROUP BY pc.category_name;
```

#### MongoDB

```
db.product.aggregate([
  {
    "$lookup": {
      "from": "order",
      "localField": "product_ID",
      "foreignField": "order_items.item_productID",
      "as": "orders"
    }
  },
  {
    "$group": {
      "_id": "$product_category.category_name",
      "total_orders": { "$sum": { "$size": "$orders" } }
    }
  }
])
```

#### Neo4J

```
MATCH (o:Order_Details)-[:CONTAINS_ITEM]->(oi:Order_Items)-[:IS_A]->(p:Product)
WITH p.Category_name AS category, COUNT(oi) AS num_items
RETURN category, SUM(num_items) AS total_sales
```

### 5.2.2 Listar o método de entrega preferido pelos clientes

#### Oracle

```
SELECT shipping_method, COUNT(*) AS preference_count
FROM order_details
GROUP BY shipping_method
ORDER BY preference_count DESC;
```

#### MongoDB

```
db.order.aggregate([
  {
    $group: {
      _id: "$order_shippingMethod",
      count: { $sum: 1 }
    }
  },
  {
```

```

    $sort: { count: -1 }
  },
  {
    $limit: 1
  }
]
])

```

#### Neo4J

```

PROFILE MATCH (o:Order_Details)-[:PLACED_BY]->(u:Store_User)
WITH o.shipping_method AS shipping_method, COUNT(*) AS preference_count
RETURN shipping_method, preference_count
ORDER BY preference_count DESC;

```

### 5.3 Queries relativas a *Employees*

#### 5.3.1 Listar o Funcionário mais antigo

##### Oracle

```
SELECT * FROM Employees ORDER BY hire_date ASC FETCH FIRST 1 ROWS ONLY;
```

##### MongoDB

```
db.Employees.find().sort({ emp_hireDate: 1 }).limit(1);
```

##### Neo4J

```

MATCH (e:Employees)
RETURN e
ORDER BY e.hire_date ASC
LIMIT 1

```

#### 5.3.2 Listar o Funcionário com maior salário

##### Oracle

```

SELECT * FROM Employees
WHERE salary = (SELECT MAX(salary) FROM Employees);

```

##### MongoDB

```

db.employees.find(
  { emp_salary: { $eq: db.employees.aggregate([ { $group: { _id: null, maxSalary: { $max:
    "$emp_salary" } } } ]).next().maxSalary } }
);

```

##### Neo4J

```

MATCH (e:Employees)
RETURN e
ORDER BY e.salary DESC
LIMIT 1

```



### 5.3.3 Listar o Funcionário que é *Manager* e tem mais subordinados

#### Oracle

```
SELECT m.EMPLOYEE_ID, m.FIRST_NAME, m.LAST_NAME, COUNT(*) AS num_subordinados
FROM employees m
JOIN employees s ON m.EMPLOYEE_ID = s.MANAGER_ID
GROUP BY m.EMPLOYEE_ID, m.FIRST_NAME, m.LAST_NAME
HAVING COUNT(*) > 0
ORDER BY COUNT(*) DESC
```

#### MongoDB

```
db.employees.aggregate([
  {
    $match: {
      emp_managerID: { $ne: 0 }
    }
  },
  {
    $lookup: {
      from: "employees",
      localField: "emp_ID",
      foreignField: "emp_managerID",
      as: "subordinates"
    }
  },
  {
    $addFields: {
      numSubordinates: { $size: "$subordinates" }
    }
  },
  {
    $sort: {
      numSubordinates: -1
    }
  },
  {
    $limit: 1
  },
  {
    $project: {
      emp_ID: 1,
      emp_firstName: 1,
      emp_LastName: 1,
      numSubordinates: 1
    }
  }
])
```

Essa *query* realiza as seguintes etapas:

- Filtra os funcionários que possuem um ID de gerente diferente de 0 (ID de gerente atribuído em casos com valores nulos)
- Realiza uma junção com a mesma coleção "employees" para buscar os subordinados de cada gerente.
- Adiciona um campo "numSubordinates" que representa o número de subordinados de cada gerente.

- Ordena os funcionários pelo número de subordinados em ordem decrescente.
- Limita o resultado a apenas 1 funcionário (o gerente com mais subordinados).
- Projeta os campos desejados no resultado final.

#### Neo4J

```
MATCH (m:Employees)-[:MANAGES]->(s:Employees)
WITH m, count(s) as num_subordinados
WHERE num_subordinados > 0
RETURN m.id AS EMPLOYEE_ID, m.first_name AS FIRST_NAME, m.last_name AS LAST_NAME,
       num_subordinados
ORDER BY num_subordinados DESC
LIMIT 1
```

## 5.4 Queries relativas a *Products*

### 5.4.1 Encontrar os produtos mais populares com base no número de pedidos

#### Oracle

```
SELECT p.product_name, COUNT(DISTINCT p.product_id) AS order_count
FROM product p
JOIN order_items oi ON oi.product_id = p.product_id
JOIN order_details od ON od.order_details_id = oi.order_details_id
GROUP BY p.product_name
ORDER BY order_count DESC;
```

#### MongoDB

```
db.order.aggregate([
  {
    $unwind: "$order_items"
  },
  {
    $group: {
      _id: "$order_items.item_productID",
      order_count: { $sum: 1 }
    }
  },
  {
    $lookup: {
      from: "product",
      localField: "_id",
      foreignField: "product_ID",
      as: "product"
    }
  },
  {
    $unwind: "$product"
  },
  {
    $project: {
      _id: 0,
      product_name: "$product.product_name",
      order_count: 1
    }
  }
])
```

```

},
{
  $sort: { order_count: -1 }
}
]);

```

Esta *query* para o *MongoDB* passa por vários estágios do pipeline de agregação:

- **\$unwind**: Este estágio desenrola o *array* "order\_items", criando um novo documento para cada elemento do *array*. Isto permite-nos trabalhar com itens de pedido individuais em etapas subsequentes.
- **\$group**: Esta etapa agrupa os documentos pelo campo "item\_productID" dos "order\_items" desenrolados. Ele calcula a contagem de pedidos para cada produto exclusivo usando o acumulador \$sum.
- **\$lookup**: Este estágio realiza uma junção à esquerda com a coleção "produto", recupera as informações do produto com base no campo "\_id" do estágio \$group anterior e no campo "product\_ID" da coleção "product". O produto correspondente é adicionado ao documento no campo denominado "produto".
- **\$unwind**: Este estágio faz *unwind* do *array* "product" criado pelo estágio \$lookup anterior. Esta etapa é necessária, pois a operação \$lookup retorna uma matriz, embora esperemos apenas uma correspondência de produto.
- **\$project**: Esta etapa reformula o documento, excluindo o campo "\_id" e selecionando o campo "product.name" do documento "product". Também inclui o campo "order\_count" da fase \$group anterior.
- **\$sort**: Esta etapa classifica os documentos com base no campo "order\_count" em ordem decrescente (-1 indica ordem decrescente).

O resultado final fornecerá os nomes dos produtos e as contagens de pedidos correspondentes, classificados pela contagem de pedidos em ordem decrescente.

## Neo4J

```

MATCH (p:Product)<-[:IS_A]-(:Order_Items)<-[:CONTAINS_ITEM]-(o:Order_Details)
RETURN p.Product_name, COUNT(DISTINCT o) AS Order_Count
ORDER BY Order_Count DESC

```

### 5.4.2 Listar os produtos que nunca foram vendidos

#### Oracle

```

SELECT p.*
FROM PRODUCT p
LEFT JOIN ORDER_ITEMS o ON p.PRODUCT_ID = o.PRODUCT_ID
WHERE o.PRODUCT_ID IS NULL;

```

#### MongoDB

```

db.product.aggregate([
{
  "$lookup": {
    "from": "order",
    "localField": "product_ID",
    "foreignField": "order_items.item_productID",

```

```

        "as": "orders"
    }
},
{
    "$match": {
        "orders": { "$size": 0 }
    }
},
{
    "$project": {
        "_id": 0,
        "product_name": 1
    }
}
])

```

- \$lookup: Esta etapa realiza um *left-join* com a coleção "order\_items", faz referência ao campo "product\_ID" da coleção "product" e ao campo "item\_productID" da coleção "order\_items". Os documentos correspondentes da coleção "order\_items" são adicionados ao *array* "order\_items" no documento.
- \$match: Esta etapa filtra os documentos onde o tamanho do *array* "order\_items" é igual a 0, ou seja, não há pedidos associados ao produto.
- \$project: Esta etapa projeta apenas o campo "product\_name" no resultado, removendo o campo "\_id".

#### Neo4J

```

MATCH (p:Product)
WHERE NOT EXISTS {
    MATCH (:Order_Details)-[:CONTAINS]->(:Order_Items)-[:IS_A]->(p)
}
RETURN p.Product_name AS product_name

```

## 6 Análise crítica e comparação de modelos

De modo a permitir estabelecer uma análise crítica do trabalho realizado, nesta secção será efetuada uma comparação entre os modelos e as funcionalidades implementadas com as funcionalidades que são disponibilizadas no sistema relacional fornecido.

Uma vez que um dos grandes pilares da escolha de uma base de dados é a velocidade de resposta das *queries*, nomeadamente em sistemas com um elevado número de dados como os presentes em lojas, elaborou-se a Tabela abaixo para permitir estabelecer uma comparação entre os vários DBMS. Os valores do tempo de execução de cada *query* foram obtidos num único computador de forma a manter a consistência entre cada execução e entre cada SGBD. Para obter as medições relativas à Oracle e ao Neo4J, tanto o *SQL Developer* como o *Neo4J Desktop* retornam por defeito o tempo de execução de cada *query*.

Medição (s)	Oracle	MongoDB	Neo4J
Query 5.1.1	0,006	0,02	0,115
Query 5.1.2	0,019	0,031	0,119
Query 5.1.3	0,005	0,0009	0,009
Query 5.1.4	0,019	0,029	0,031
Query 5.2.1	0,074	0,067	0,052
Query 5.2.2	0,005	0,018	0,034
Query 5.3.1	0,038	0,0029	0,07
Query 5.3.2	0,009	0,019	0,002
Query 5.3.3	0,039	0,037	0,02
Query 5.4.1	0,016	0,021	0,04
Query 5.4.2	0,01	0,0077	0,009

Tabela 1: Comparação dos tempos de execução de cada *query* em cada Base de Dados

Observa-se que, em algumas das consultas, o Oracle apresentou tempos de execução mais rápidos em comparação com o MongoDB e o Neo4J, podendo ser atribuído à otimização do Oracle para consultas SQL e ao uso de índices para acelerar as operações.

O MongoDB demonstrou bom desempenho em consultas que envolvem agregação e relacionamentos, como as consultas 5.2.1 e 5.2.2. A estrutura flexível de documentos do MongoDB permite realizar operações de agregação de forma eficiente.

O Neo4J, sendo uma base de dados de grafos, obteve bom desempenho nas consultas relativas a relacionamentos complexos, como a consulta 5.3.3. A estrutura de grafo do Neo4J permite consultas eficientes em cenários onde os relacionamentos são fundamentais.

Para as consultas que envolvem cálculos de média (consulta 5.1.4), o MongoDB teve um desempenho superior ao Oracle, possivelmente devido à sua elevada capacidade de processamento de consultas de agregação.

É interessante notar que, em algumas consultas específicas, como a consulta 5.3.2 no Oracle e a consulta 5.3.1 no MongoDB, foram observados tempos de execução surpreendentemente baixos. Isso pode ser devido às otimizações específicas da base de dados ou à natureza dos dados armazenados.

O ponto principal a tirar desta análise é que as diferenças de desempenho entre as bases de dados podem ser influenciadas pela quantidade de dados, índices, configurações e estrutura dos dados, bem como pelo tipo de consulta realizada, devendo ser avaliado cada caso para a escolha correta do SGBD a utilizar.

## 7 Triggers

Relativamente à utilização de *triggers*, para o MongoDB foi utilizado o MongoDB Atlas, utilizando a aba *Triggers* presente, de modo a facilitar todo o processo. No que toca ao Neo4J, através do APOC, *plug-in* instalado no Neo4J *Desktop*, é possível utilizar *triggers* que permitem o registo de *queries* Cypher que são chamadas quando os dados no Neo4J são alterados (*created*, *updated* ou *deleted*). Podem ser corridos antes ou depois de *commit*.

### 7.1 Triggers MongoDB

#### 7.1.1 Update - Atualizar *department* ID do *employee* se o ID do departamento atualizar

De modo a que, em caso de um ID de departamento ser alterado, mantendo-se o departamento, o ID do departamento associado ao *employee* é atualizado de igual modo.

```
exports = async function(changeEvent) {
  const { fullDocument, updateDescription } = changeEvent;

  if (updateDescription && updateDescription.updatedFields['department_info.department_id']) {

    const newDepartmentId = updateDescription.updatedFields['department_info.department_id'];
    console.log("newDepartmentId= ", newDepartmentId)
    const employees = context.services.get("Cluster0").db("oes").collection("employees");

    await employees.updateMany(
      { "department_info.department_id": newDepartmentId },
      { $set: { emp_departmentID: newDepartmentId } }
    );
  }
};
```

#### 7.1.2 Delete - Eliminar todas as *Orders* de um *User* que foi apagado

Quando um utilizador do sistema é apagado da BD, então as *Orders* que lhe pertencem são apagadas também. Este *trigger* é ativado aquando um *Delete* na *collection User*, sendo que obriga a ativar a opção de Document Preimage, permitindo assim obter o documento que foi *deleted* antes das alterações serem aplicadas.

```
exports = async function(changeEvent) {
  var deletedDocument = changeEvent.fullDocumentBeforeChange;

  if (deletedDocument){
    const userId = deletedDocument.user_ID;
    const collection = context.services.get("Cluster0").db("oes").collection("order");
    await collection.deleteMany({ order_userID: userId });
  }
};
```

#### 7.1.3 Insert - Se uma *order* for inserida com um *user* ID inexistente, inserir também o *user*

Um utilizador não registado poderá efetuar uma *order*. Deste modo, caso a *order* seja aprovada e o utilizador não exista, é então criado um utilizador junto com a sua sessão.

```
exports = async function(changeEvent) {
```

```

const { fullDocument } = changeEvent;

if (fullDocument) {
  const userId = fullDocument.order_userID;

  const usersCollection = context.services.get("Cluster0").db("oes").collection("user");
  const ordersCollection = context.services.get("Cluster0").db("oes").collection("order");

  const existingUser = await usersCollection.findOne({ user_ID: userId });

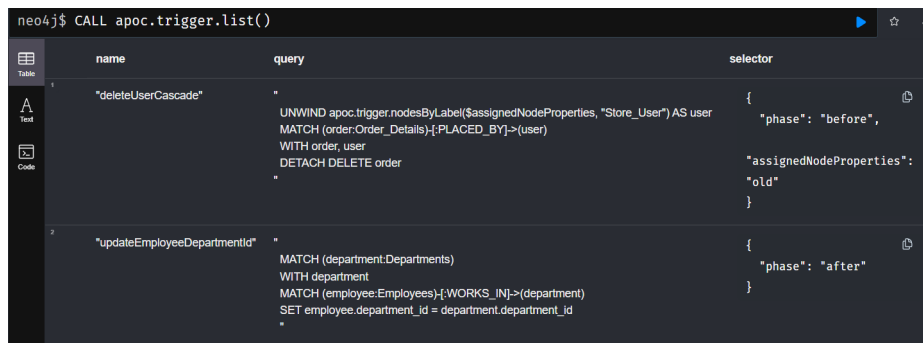
  if (!existingUser) {
    const newUser = {
      user_ID: userId,
      shopping_sessions: [
        {
          session_ID: 99,
        }
      ]
    };
    await usersCollection.insertOne(newUser);
  }
}
};

```

## 7.2 Triggers Neo4J

Após configurar corretamente o APOC, particularmente o ficheiro *apoc.conf* foi possível adicionar *triggers* para o Neo4J.

De modo a verificar que cada *trigger* ficou corretamente criado é possível correr o seguinte comando: "CALL apoc.trigger.list()"



name	query	selector
"deleteUserCascade"	<pre> UNWIND apoc.trigger.nodesByLabel(\$assignedNodeProperties, "Store_User") AS user MATCH (order:Order_Details)-[PLACED_BY]-&gt;(user) WITH order, user DETACH DELETE order </pre>	<pre> {   "phase": "before",   "assignedNodeProperties":     "old" } </pre>
"updateEmployeeDepartmentId"	<pre> MATCH (department:Departments) WITH department MATCH (employee:Employees)-[WORKS_IN]-&gt;(department) SET employee.department_id = department.department_id </pre>	<pre> {   "phase": "after" } </pre>

Figura 5: Lista de Triggers implementados e corretamente a funcionar

### 7.2.1 Update - updateEmployeeDepartmentId

De modo a que, em caso de um ID de departamento ser alterado, mantendo-se o departamento, o ID do departamento associado ao *employee* é atualizado de igual modo.

```

CALL apoc.trigger.add(
  'updateEmployeeDepartmentId',
  MATCH (department:Departments)
  WITH department

```

```

MATCH (employee:Employees)-[:WORKS_IN]->(department)
SET employee.department_id = department.department_id
',
{ phase: 'after' }
)

```

Para testá-lo, basta atualizar o *department\_id* de um Departamento.

```

MATCH (department:Departments)
WHERE department.id = 1
SET department.id = 100

```

Posteriormente, é necessário verificar se o *trigger* teve efeito, para isso, é necessário verificar se os *Employees* que tenham uma relação *WORKS\_IN* com o departamento que teve o seu ID alterado, também sofreram essa alteração:

```

MATCH (employee:Employees)-[:WORKS_IN]->(department:Departments)
RETURN employee, department

```

### 7.2.2 Delete - DeleteUserCascade

Se um utilizador (*Store\_User*) for removido da base de dados, são removidas as suas encomendas (*Order\_Details*) de modo a garantir consistência dos dados.

```

CALL apoc.trigger.add('deleteUserCascade', '
UNWIND apoc.trigger.nodesByLabel($assignedNodeProperties, "Store_User") AS user
MATCH (order:Order_Details)-[:PLACED_BY]->(user)
WITH order, user
DETACH DELETE order
', {phase: 'before', assignedNodeProperties: 'old'}) ;

```

Para o testar, basta apagar um determinado utilizador que possua entradas em *Order\_Details*.

```

MATCH (u:Store_User {id: 1})
DETACH DELETE u

```

Posteriormente, verificar se ainda existe relação entre ambos os nodos.

```

MATCH (o:Order_Details)-[:PLACED_BY]->(u:Store_User {id: 1})
RETURN o, u

```

### 7.2.3 Insert - insertStoreUser

Seguindo a mesma lógica do *trigger* de *insert* feito para o MongoDB, houve a tentativa de criar um *trigger* semelhante para o Neo4J, ou seja, se uma *Order* fosse criada com um ID de utilizador que não existisse, seria criado um *Store\_User* e uma *Shopping\_Session* correspondente.

Infelizmente, não foi possível implementá-lo corretamente, mas ainda assim, apresenta-se a seguir o *trigger*:

```

CALL apoc.trigger.add(
'insertStoreUser',
',
UNWIND $createdNodes AS order
MATCH (order)-[:PLACED_BY]->(user:Store_User)
WHERE NOT EXISTS()-[:BELONGS_TO]->(user))
CREATE (storeUser:Store_User {id: user.id})

```



```
CREATE (session:Shopping_Session {id: $sessionId})
CREATE (session)-[:BELONGS_TO]->(storeUser)
',
{ phase: 'after' }
)
```

Foi possível adicioná-lo à lista de *triggers*, mas na sua testagem não houve sucesso.  
Criar uma encomenda com um *user\_id* que não existe:

```
CREATE (o25:Order_Details {id:1, user_id:1000, total:575, payment_id:1,
shipping_method:'Inpost', delivery_address_ID:12, created_at:'2022-05-18 23:29:16',
modified_at:'2022-07-01 09:57:48'});
```

Verificar se o resultado foi o esperado (Alterar *sessionID* pelo ID desejado):

```
MATCH (order:Order_Details {id: 1})
WITH order
MATCH (user:Store_User {id: order.user_id})
WHERE NOT EXISTS()-[:BELONGS_TO]->(user))
CREATE (storeUser:Store_User {id: user.id})
CREATE (session:Shopping_Session {id: <sessionId>})
CREATE (session)-[:BELONGS_TO]->(storeUser)
```

## 8 Conclusão

O objetivo deste trabalho prático centrou-se no desenvolvimento de competências na utilização de diferentes paradigmas de bases de dados e na sua aplicação na conceção e implementação de sistemas, tendo sido trabalhadas as competências com Bases de Dados Relacionais, Documentais e orientadas a Grafos, assim como a migração entre as mesmas.

Com o presente relatório apresenta-se uma análise detalhada, passo a passo, da base de dados relacional original, juntamente com as medidas necessárias para trabalhar com os dados fornecidos e desenvolver os sistemas de bases de dados desejados. Além disso, foram definidas e implementadas *queries* para demonstrar a operacionalidade dos sistemas implementados, assim como *triggers*, que permitem complementar e enriquecer o sistema.

Por fim, realizou-se uma análise crítica comparativa entre os modelos e funcionalidades agora implementadas com as disponibilizadas no sistema relacional fornecido. Essa comparação permite avaliar os benefícios e desafios de cada paradigma de base de dados, auxiliando na compreensão das vantagens e limitações de cada abordagem.

Em suma, este projeto permitiu adquirir conhecimentos práticos na utilização de diferentes tipos de bases de dados, explorando as potencialidades dos sistemas relacionais, orientados a documentos e grafos. Essas competências são essenciais para a conceção e implementação eficiente de sistemas de bases de dados, atendendo às necessidades específicas de cada contexto e aplicação.