



Universidade do Minho

Mestrado em Engenharia Informática

Bases de Dados NoSQL

Base de Dados Online Electronics Store

Ana Murta (PG50184)
Hugo Gomes (PG51242)
Manuel Novais (PG50575)

junho, 2023

Conteúdo

1	Introdução	6
2	Online Electronics Store	7
2.1	Esquema Lógico	7
3	Base de Dados Relacional - Oracle	10
3.1	Importação dos dados	10
3.2	Queries	10
3.2.1	Queries Base	10
3.2.2	Queries Extra	12
4	Base de Dados Documental - MongoDB	14
4.1	Migração dos dados	14
4.2	Estruturação dos dados	16
4.3	Queries	21
4.3.1	Queries base	21
4.3.2	Queries Extra	24
5	Base de Dados Orientada a Grafos – Neo4j	27
5.1	Migração dos dados	27
5.2	Estruturação dos dados	28
5.3	Queries	29
5.3.1	Queries base	29
5.3.2	Queries Extra	31

6	Comparação do desempenho das queries	33
6.1	Query 1	33
6.2	Query 2	34
6.3	Query 3	34
6.4	Query 4	35
7	Conclusão	36

Lista de Figuras

2.1	Esquema lógico da base de dados OES (Oracle)	7
3.1	Resultado da query base 1 (SQL)	11
3.2	Resultado da query base 2 (SQL)	11
3.3	Resultado da query base 3 (SQL)	12
3.4	Resultado da query base 4 (SQL)	12
3.5	Resultado da query extra 1 (SQL)	13
3.6	Resultado da query extra 2 (SQL)	13
4.1	Comando de criação da view	15
4.2	Pipeline da view	15
4.3	Nível 1 de aninhamento (JSON)	16
4.4	EMPLOYEES no nível 2 de aninhamento (JSON)	17
4.5	Nível 1 de aninhamento (JSON)	17
4.6	CART_ITEM no nível 2 de aninhamento (JSON)	17
4.7	Nível 1 de aninhamento (JSON)	17
4.8	DELIVERY_ADRESS no nível 2 de aninhamento (JSON)	18
4.9	PAYMENT no nível 2 de aninhamento (JSON)	18
4.10	ORDER_ITEMS no nível 2 de aninhamento (JSON)	18
4.11	Nível 1 de aninhamento (JSON)	19
4.12	SHOPPING_SESSION e ORDERS no nível 2 de aninhamento (JSON)	19
4.13	Nível 1 de aninhamento (JSON)	19
4.14	CATEGORY, DISCOUNT e STOCK no nível 2 de aninhamento (JSON)	20
4.15	Nível 1 de aninhamento (JSON)	20

4.16	Resultado da query base 1 (MongoDB)	21
4.17	Resultado da query base 2 (MongoDB)	22
4.18	Resultado da query base 3 (MongoDB)	23
4.19	Resultado da query base 4 (MongoDB)	23
4.20	Resultado da query extra 1 (MongoDB)	25
4.21	Resultado da query extra 2 (MongoDB)	26
5.1	Modelo lógico da base de dados OES (Neo4j)	28
5.2	Resultado da query base 1 (Neo4j)	29
5.3	Resultado da query base 2 (Neo4j)	30
5.4	Resultado da query base 3 (Neo4j)	30
5.5	Resultado da query base 4 (Neo4j)	31
5.6	Resultado da query extra 1 (Neo4j)	31
5.7	Resultado da query extra 2 (Neo4j)	32

Capítulo 1

Introdução

O constante avanço da tecnologia e o advento da era digital têm resultado num aumento exponencial no volume de dados gerados diariamente. Empresas, instituições e organizações de todos os setores estão a lidar com uma quantidade cada vez maior de informações, provenientes de diversas fontes, como transações comerciais, registos de clientes, interações em redes sociais, sensores e dispositivos conectados à Internet.

Esse aumento massivo no volume de dados representa um desafio significativo para os sistemas tradicionais de base de dados, baseados em modelos relacionais. Esses sistemas muitas vezes têm dificuldades em lidar com o processamento, armazenamento e análise eficientes de grandes quantidades de informações estruturadas. Além disso, a natureza dos dados modernos, muitas vezes semi ou não estruturados, não se encaixa perfeitamente no modelo relacional, que requer uma estrutura rígida e predefinida.

Diante esse cenário, surgem as bases de dados não relacionais como alternativa viável para lidar com o aumento do volume de dados e a necessidade de flexibilidade na modelagem das informações. Essas bases de dados são projetadas para oferecer escalabilidade horizontal, capacidade de armazenamento e recuperação eficiente de dados não estruturados ou semi-estruturados, além de permitir a exploração de relações complexas entre as entidades.

Neste contexto, o presente projeto tem como objetivo migrar uma base de dados no esquema relacional para dois esquemas não relacionais: MongoDB e Neo4j. Essa migração permitirá explorar as vantagens oferecidas por essas bases de dados específicas, superando os desafios impostos pelo aumento do volume de dados e pela necessidade de uma estrutura mais flexível.

Ao migrar para o MongoDB, uma base de dados orientada a documentos, e para o Neo4j, orientada a grafos, esperamos alcançar uma maior eficiência e capacidade de análise em relação à base de dados relacional. Através deste projeto, poderemos avaliar como essas bases de dados não relacionais podem atender às necessidades específicas do nosso cenário, proporcionando uma solução mais adequada para a gerência e exploração dos dados.

No decorrer deste relatório, serão apresentados os detalhes da migração, incluindo o processo de extração dos dados da base de dados relacional, a transformação necessária para se adequarem aos formatos do MongoDB e do Neo4j, bem como os resultados obtidos e as conclusões decorrentes desse projeto de migração.

Capítulo 2

Online Electronics Store

O *Online Electronics Store* (OES) é uma base de dados Oracle de uma loja fictícia de eletrônicos online. Esta armazena várias informações sobre a mesma, como por exemplo, os utilizadores registados no site, os departamentos entre os funcionários da loja, entre outros.

2.1 Esquema Lógico

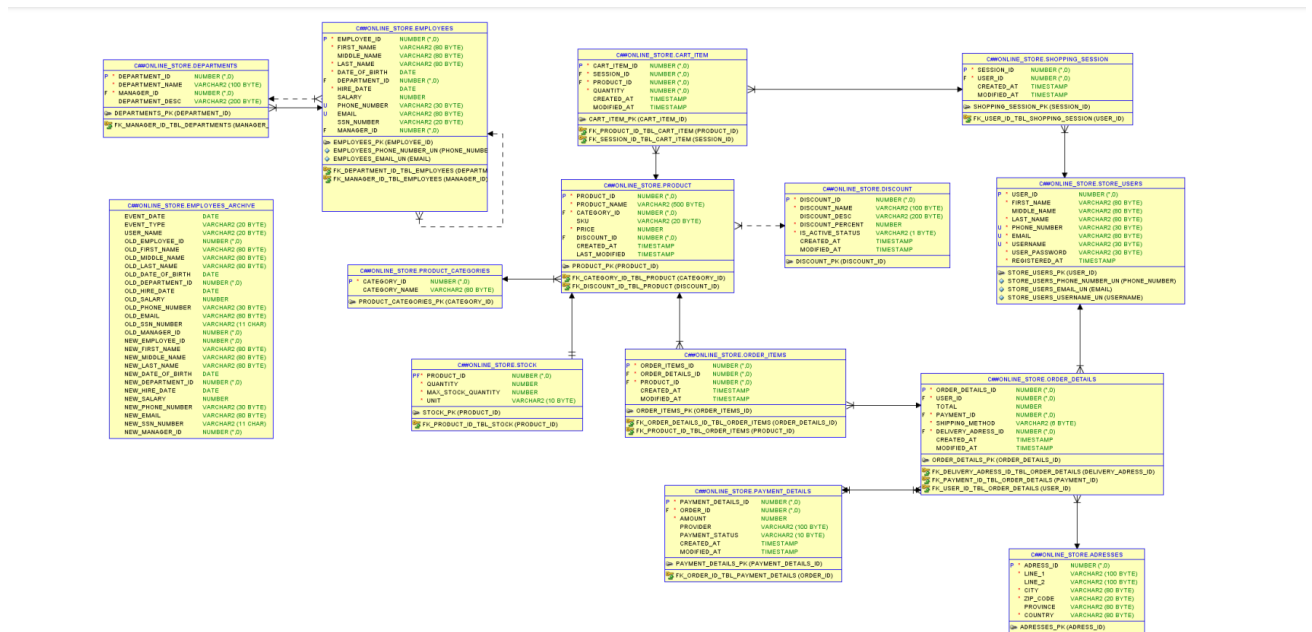


Figura 2.1: Esquema lógico da base de dados OES (Oracle)

O esquema lógico do OES é constituído por 14 tabelas distintas: **STORE_USERS**, **PRODUCT_CATEGORIES**, **PRODUCT**, **DISCOUNT**, **CART_ITEM**, **SHOPPING_SESSION**, **ORDER_DETAILS**, **ORDER_ITEMS**, **PAYMENT_DETAILS**, **EMPLOYEES**, **DEPARTMENTS**, **ADDRESSES**, **EMPLOYEES_ARCHIVE**, **STOCK**.

A tabela **STORE_USERS** contém os utilizadores registados no site da loja incluindo informações sobre o primeiro e último nome do utilizador, o seu número de telemóvel, email, nome de usuário, password e data de registo.

No caso da tabela **PRODUCT**, esta fornece informações sobre os produtos que a loja vende, como o seu nome, o SKU, o preço e, as datas de criação e modificação. Esta tabela possui também, como chaves estrangeiras, informação relativa à categoria e desconto do produto, sendo que este só pode estar contido numa categoria e ter um desconto associado.

Na tabela **PRODUCT_CATEGORIES** é armazenada informação referente às categorias dos produtos, incluindo informação sobre o nome da categoria e o seu id. Uma categoria pode ter mais do que um produto.

A tabela **DISCOUNT** contém informações sobre promoções ativas e expiradas na loja, incluindo o nome, descrição, percentagem de desconto, status ativo/não ativo e, as datas de criação e modificação.

Na tabela **STOCK** é armazenado informações sobre o *stock* dos produtos, incluindo, a quantidade atual e máxima permitida do produto e unidade de medida utilizada para contabilizar a quantidade em *stock* do mesmo. Para além disto, esta possui, como chave estrangeira, informação relativa ao produto, isto é, cada *stock* está associado a um produto.

De seguida, temos a tabela **CART_ITEM** que contém informação referente aos produtos adicionados ao carrinho pelo cliente numa determinada sessão. Ela armazena a quantidade e as datas de criação e modificação. Esta tabela possui também, como chaves estrangeiras, informação relativa à sessão em que criou o carrinho e ao produto adicionado, sendo que cada carrinho só está associado a um produto e sessão.

A tabela **SHOPPING_SESSION** armazena informações sobre as sessões criadas pelos utilizadores, nomeadamente, as datas de criação e modificação. Esta também possui uma chave estrangeira que referencia a tabela **STORE_USERS**. Assim sendo temos que cada sessão está associada a um utilizador.

Na tabela **ORDER_DETAILS** contém informações sobre os detalhes do pedido realizado pelo o utilizador, como o total, o método de envio e, as datas de criação e modificação. Esta tabela possui também, como chaves estrangeira, informações relativas ao usuário que efetuou o pedido, o método de pagamento utilizado e o endereço de entrega da mesma, sendo que cada *order_detail* está apenas associado a um para cada um destes. Para além desta, esta ainda contém, como chave estrangeira, informação referente aos produtos do pedido, sendo que esta pode estar associada a mais do que um *order-item*.

No caso da tabela **ORDER_ITEMS** é armazenada informação referente aos produtos do pedido feito pelo utilizador, incluindo, as datas de criação e modificação. Esta contém também, como chaves estrangeiras, informações relativas a aos produto e detalhes pedido realizado pelo o usuário, sendo que só está associada apenas a um produto e a um detalhe.

A tabela **PAYMENT_DETALIS** contém informações sobre os detalhes de pagamento do pedido, incluindo o valor, o provedor e status do pagamento e, as datas de criação e modificação. Para além disto, possui também, como chave estrangeira, informação referente ao ID do pedido, associando cada pagamento ao mesmo.

De seguida, temos a tabela **ADDRESSES** que contém as moradas dos clientes, incluindo informações sobre a sua província, a cidade, o país, *line_1*, *line_2* e o código postal.

A tabela **EMPLOYEES** contém informações sobre os funcionários da loja, como o seu nome,

data de nascimento, data de contratação, salário, email, número de telemóvel e número da segurança social. Esta também possui, como chaves estrangeiras, informações relativas ao departamento e *manager* de cada funcionário, sendo que este só pode pertencer a um departamento e ter um *manager*.

Na tabela **DEPARTMENTS** são armazenadas dos departamentos dos funcionários internos da loja, incluindo, o nome e descrição do mesmo. Para além disto, esta também possui, como chave estrangeira, informações referente ao *manager* do departamento.

Por fim, a tabela **EMPLOYEES_ARCHIVE** contém os dados de funcionários arquivados - novas linhas adicionadas, linhas atualizadas e linhas excluídas, juntamente com informações sobre a hora da modificação e também sobre o utilizador que fez as alterações.

Capítulo 3

Base de Dados Relacional - Oracle

3.1 Importação dos dados

No início do projeto, foi realizado um processo de importação dos dados disponibilizados da base de dados EOS para o Oracle, que será usado como sistema de gerência do banco de dados relacional para as diversas tabelas mencionadas anteriormente. Esta importação foi realizada utilizando os ficheiros fornecidos no arquivo "oes.zip", que continham os *scripts* necessários para a criação do banco de dados e população das tabelas. Isto permitiu obter, assim, uma base de trabalho para a realização das várias *queries*. O grupo utilizou o SQL Developer para estabelecer a conexão com o banco de dados e executar as consultas sobre as tabelas disponíveis.

3.2 Queries

Uma das tarefas pedidas no trabalho prático é a realização de uma análise crítica do mesmo, comparando, entre os modelos relacional e não relacionais das bases de dados, as funcionalidades implementadas. Deste modo, o grupo decidiu utilizar um conjunto base de quatro *queries* que serão usadas nos distintos modelos das bases de dados. Para além destas, também foram criadas mais duas interrogações para cada modelo com o intuito de experimentar diferentes operações aplicadas nas aulas.

3.2.1 Queries Base

Query 1

Obter o nome e a categoria de cada produto com um stock menor do que 10.

```
SELECT p.product_name AS Product_Name, pc.category_name AS Category_Name
FROM PRODUCT p
INNER JOIN PRODUCT_CATEGORIES pc ON p.category_id = pc.category_id
INNER JOIN STOCK s ON p.product_id = s.product_id
WHERE s.quantity < 10;
```

Executando a *query* anterior, obtém-se os seguintes resultados:

	PRODUCT_NAME	CATEGORY_NAME
1	Asus Vivobook Pro 14	Computers, Laptops and Consoles
2	BenQ GW2283	Monitors

Figura 3.1: Resultado da query base 1 (SQL)

Query 2

Listar todas as sessões de compra criadas pelo o usuário 'Evonne Warin' .

```
SELECT ss.*
FROM SHOPPING_SESSION ss
INNER JOIN STORE_USERS su ON ss.user_id = su.user_id
WHERE su.first_name = 'Evonne' AND su.last_name = 'Warin';
```

Os resultados obtidos depois da execução da *query*:

	SESSION_ID	USER_ID	CREATED_AT	MODIFIED_AT
1	12	12	22.01.14 02:12:10,000000000	22.01.14 08:19:22,000000000

Figura 3.2: Resultado da query base 2 (SQL)

Query 3

Obter todos os utilizadores (username,phone_number,email) que fizeram pedidos de produtos na categoria 'Monitors' depois do mês de março de 2022.

```
SELECT su.username AS Username, su.phone_number AS Phone_Number, su.email As Email
FROM STORE_USERS su
INNER JOIN ORDER_DETAILS od ON su.user_id = od.user_id
INNER JOIN ORDER_ITEMS oi ON od.order_details_id = oi.order_details_id
INNER JOIN PRODUCT p ON oi.product_id = p.product_id
INNER JOIN PRODUCT_CATEGORIES pc ON p.category_id = pc.category_id
WHERE pc.category_name = 'Monitors'
AND EXTRACT(YEAR FROM od.created_at) >= 2022
AND EXTRACT(MONTH FROM od.created_at) >= 3;
```

Após a execução da *query*, foram obtidos os seguintes resultados:

	PRODUCT_NAME	PRODUCT_PRICE
1	Samsung Galaxy A52	1199
2	Apple Iphone 11	2499
3	Apple Iphone 12 mini	3599

Figura 3.5: Resultado da query extra 1 (SQL)

Query 2

Obter o nome completo, email e nome do departamento dos 4 funcionários com os salários mais altos por ordem decrescente.

```
SELECT e.first_name AS "First Name", e.last_name As "Last Name",
e.email AS "Email",
d.department_name AS "Department",
e.Salary AS "Salary"
FROM Employees e
INNER JOIN DEPARTMENTS d ON e.department_id = d.department_id
ORDER BY e.salary DESC
FETCH FIRST 4 ROWS ONLY;
```

Foram obtidos os seguintes resultados, após a execução da *query*:

	First Name	Last Name	Email	Department	Salary
1	Brittney	Dimitriou	ldimitrioul@icio.us	Development	11800
2	Ansel	Stanborough	gstanborough8@webs.com	Sales department	11200
3	Kenon	Andries	gandries0@google.de	Management	9300
4	Eileen	Sowerbutts	fsowerbutts9@illinois.edu	Sales department	8300

Figura 3.6: Resultado da query extra 2 (SQL)

Capítulo 4

Base de Dados Documental - MongoDB

4.1 Migração dos dados

Após a análise do esquema relacional fornecido, o grupo procedeu à migração dos dados da base de dados Oracle para o formato JSON, uma vez que o MongoDB permite a importação de registos neste formato.

Desta forma, a estratégia aplicada pelo o grupo foi criar um *script* em python (`db_mongo.py`) que exporta os dados para objetos JSON. Isto é conseguido estabelecendo uma conexão à base de dados Oracle no SQL Developer e de consultas SQL. Depois, os dados obtidos são transformados e importados para as coleções correspondentes. Posteriormente, no ficheiro `db_insert2mongo.py` os ficheiros json são lidos e os seus dados inseridos na base de dados, em conjunto com a criação das views e indexes. O código do trigger implementado está no ficheiro `trigger_archiving_employees.js`.

De modo a aproveitar ao máximo o paradigma documental do MongoDB, as coleções foram pensadas de modo a manter todas as informações coerentes sobre a mesma no mesmo documento, ou seja, os objetos JSON foram estruturados de forma a agrupar o máximo número de dados das tabelas da base de dados relacional. O conteúdo de cada objeto JSON criado para cada coleção será analisado com mais detalhe na secção seguinte.

Para realizar a migração da base de dados relacional para uma base de dados orientada a documentos, foi essencial ajustar o esquema lógico para facilitar este processo e problemas de conflito. Isto é, no processo junção das tabelas, que será explicado na próxima secção, havia problemas de conflitos uma vez que era necessário combinar tabelas que continham colunas com o mesmo nome, pelo que, nestes casos, foi preciso alterar o nome de certas colunas em determinadas tabelas. Para além disto, para as datas de criação e modificação foi alterado o seu tipo de dados timestamp para o formato "%Y-%m-%d %H:%M:%S".

A migração dos procedures e triggers foi feito com recurso ao MongoDB Atlas. O trigger "trigger_archiving_employees" foi configurado para todas as operações (insert, update, delete e replace). Também foi necessário ativar as opções de "Full Document" e "Document Pre-image" para ser possível obter os dados do documento após e antes este ser alterado, respetivamente. A coleção alvo deste trigger é a departments sendo que esta é a que contém os EMPLOYEES. A função a ser executada no trigger foi escrita em javascript. Devido à estrutura da base de dados adotada,

a função do trigger tem de verificar manualmente se um EMPLOYEE foi adicionado, atualizado ou removido através da comparação dos dados antes e depois de uma operação na coleção departments ser efetuada. Quando um department é adicionado ou removido, então a função sabe que vai ter de adicionar ou remover EMPLOYEES da coleção EMPLOYEES_ARCHIVE. O único problema encontrado aparece quando existe uma mudança de departamento, que irá dar trigger a uma remoção do EMPLOYEE e de seguida a uma inserção do mesmo, em vez de dar trigger a uma atualização dos dados deste EMPLOYEE. Este problema foi encontrado já na fase final do trabalho, já não sendo possível para o grupo atualizar o código, queries e relatório de modo a implementar estas mudanças.

A migração da view existente foi feita automaticamente após criar as coleções e inserir os respetivos dados. Esta foi criada através do comando da figura 4.1 e utilizando a pipeline da figura 4.2. Esta vai buscar os dados das SHOPPING_SESSIONS de todos os utilizadores, incluindo os dos produtos no carrinho, para serem apresentados na view.

```
db.create_collection('vw_user_cart', viewOn='store_users', pipeline=pipeline)
```

Figura 4.1: Comando de criação da view

```
pipeline = [
  {
    '$unwind': {
      'path': '$SHOPPING_SESSIONS'
    }
  }, {
    '$project': {
      'SHOPPING_SESSIONS': 1
    }
  }, {
    '$lookup': {
      'from': 'products',
      'localField': 'SHOPPING_SESSIONS.CART_ITEMS.PRODUCT_ID',
      'foreignField': '_id',
      'as': 'CART_ITEMS'
    }
  }, {
    '$addFields': {
      'USER_ID': '$SHOPPING_SESSIONS.USER_ID',
      'SESSION_ID': '$SHOPPING_SESSIONS.SESSION_ID',
      'CREATED_AT': '$SHOPPING_SESSIONS.CREATED_AT',
      'MODIFIED_AT': '$SHOPPING_SESSIONS.MODIFIED_AT'
    }
  }, {
    '$project': {
      'USER_ID': 1,
      'SESSION_ID': 1,
      'CREATED_AT': 1,
      'MODIFIED_AT': 1,
      'CART_ITEMS': 1
    }
  }
]
```

Figura 4.2: Pipeline da view

Relativamente aos indexes, estes são criados automaticamente após a inserção dos dados na base de dados. Apenas foi criado um índice para o PRODUCT.ID na coleção store_users e um index para os ids dos departamentos, managers e employee novos e antigos na coleção employees_archive, devido a estes serem os possíveis campos que poderão ser usados para fazer lookup para outras

coleções.

4.2 Estruturação dos dados

Para a base de dados não relacional orientada a documentos foram implementadas quatro coleções: `departments`, `products`, `store_users` e `employees_archive`.

A coleção **`departments`** armazena as informações referente aos departamentos e seus respectivos funcionários da loja, agrupando as tabelas `DEPARTMENTS` e `EMPLOYEES`.

A coleção **`products`** agrega as informações das tabelas `PRODUCT`, `PRODUCT_CATEGORIES`, `DISCOUNT` e `STOCK`, contendo, assim todas as informações detalhadas sobre os produtos que a loja vende.

Já a coleção **`store_users`** contém todas as informações referentes aos utilizadores registados no site da loja, nomeadamente, as informações sobre as suas sessões criadas e, por conseguinte, os produtos adicionados ao carrinho durante essas, agrupando, assim, as tabelas `SHOPPING_SESSION` e `CART_ITEM`. Para além disto, esta coleção também armazena todas as informações sobre os detalhes dos pedidos realizados pelos utilizadores, agregando as tabelas `ORDER_DETAILS`, `ORDER_ITEMS`, `PAYMENT_DETAILS` e `ADDRESSES`. No final, ambas combinações de tabelas são agrupadas à tabela `STORE_USERS`.

Por último, a coleção **`employees_archive`** diz respeito aos dados de funcionários arquivados, tendo sido apenas convertido os dados originais da tabela `EMPLOYEES_ARCHIVE` para o formato JSON.

Tal como referido anteriormente, antes de se proceder à importação dos dados no MongoDB, estes foram formatados para objetos JSON.

Na figura 4.3 podemos observar a estrutura aplicada à coleção `departments`. Num primeiro nível de aninhamento em JSON, a informação base armazenada é a seguinte:

```
_id:  
DEPARTMENT_NAME:  
DEPARTMENT_DESC:  
MANAGER_ID:  
▶ EMPLOYEES: Array
```

Figura 4.3: Nível 1 de aninhamento (JSON)

Para além desta, também é necessário guardar a informação relativa aos funcionários do departamento, que é uma relação de 1 para N, isto é, cada departamento têm vários funcionários. Deste modo, é necessário manter, para cada um, uma lista (*array*) de funcionários da loja (`EMPLOYEES`).

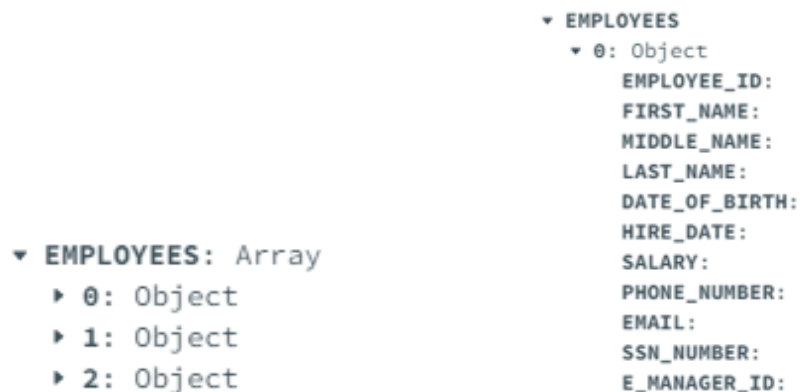


Figura 4.4: EMPLOYEES no nível 2 de aninhamento (JSON)

Para facilitar o processo da criação da coleção `store_users`, o grupo começou por criar dois objetos JSON diferentes, um referente as sessões criadas pelo o utilizador e outro que armazena as informações sobre os detalhes dos pedidos deste. Na figura 4.5 está representada a estrutura aplicada ao objeto JSON das sessões do utilizador (*shopping-session.json*). Como mencionado anteriormente, de modo a obter esta informação foram agregadas as tabelas `SHOPPING_SESSION` e `CART_ITEM` que mantém uma relação de 1 para N, ou seja, a cada carrinho está associado a uma sessão do utilizador e, cada sessão pode ter associada mais do que um carrinho. Assim sendo, foi necessário criar uma lista (*array*) de *cart_items* para cada *shopping-session*.



Figura 4.5: Nível 1 de aninhamento (JSON)

Figura 4.6: CART_ITEM no nível 2 de aninhamento (JSON)

Na figura 4.7 podemos observar a estrutura decidida para os objetos JSON que contém a informação referente aos detalhes do pedido do utilizador (*orders.json*). Num primeiro nível de aninhamento em JSON, a informação base armazenada é relativas à informação fornecida pela tabela `ORDER_DETAILS`.

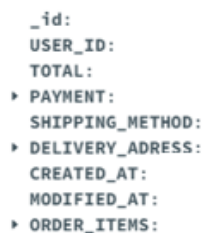


Figura 4.7: Nível 1 de aninhamento (JSON)

Para além desta, como referido anteriormente também era necessário adicionar as informações das outras tabelas de modo a completar as mesmas e maximizar a *performance* do MongoDB. Como a relação entre a tabela ORDER_DETAILS e ADRESSES, é de 1 para 1, foi necessário criar um objeto aninhado que mantém todos os dados relativos às moradas dos clientes, uma vez que um pedido está associado apenas um endereço. Já as tabelas ORDER_DETAILS e ORDER_ITEMS mantém uma relação de 1 para N, isto é, cada pedido do utilizador pode ter associado mais do que produto. Deste modo, foi criada uma lista (*array*) de *order_items* para cada *order_detail*. Por fim, ainda foi criado um objeto aninhado que armazena os dados relativos aos detalhes de pagamento, uma vez que a relação entre as tabelas ORDER_DETAILS e PAYMENT_DETAILS é de 1 para 1. Nas imagens seguintes encontra-se a representação dos objetos aninhados PAYMENT, DELIVERY_ADRESS e da lista ORDER_ITEMS.

```
▼ DELIVERY_ADRESS: Object
  ADRESS_ID:
  LINE_1:
  LINE_2:
  CITY:
  ZIP_CODE:
  PROVINCE:
  COUNTRY:
```

Figura 4.8: DELIVERY_ADRESS no nível 2 de aninhamento (JSON)

```
▼ PAYMENT: Object
  PAYMENT_ID:
  AMOUNT:
  PROVIDER:
  PAYMENT_STATUS:
  PD_CREATED_AT:
  PD_MODIFIED_AT:
```

Figura 4.9: PAYMENT no nível 2 de aninhamento (JSON)

```
▼ ORDER_ITEMS: Array
  ▼ 0: Object
    ORDER_ITEMS_ID:
    PRODUCT_ID:
    OI_CREATED_AT:
    OI_MODIFIED_AT:
  ▶ 1: Object
  ▶ 2: Object
```

Figura 4.10: ORDER_ITEMS no nível 2 de aninhamento (JSON)

Assim sendo, na figura 4.11 é possível observar a estrutura final do objeto JSON da coleção *store.users*, que num primeiro nível de aninhamento contém as informações dos utilizadores registados no site da loja. A relação entre as tabelas STORE_USERS e ORDER_DETAILS e, STORE_USERS e SHOPPING_SESSION é de 1 para N, isto é, cada utilizador pode ter associado um ou mais pedidos e sessões. Desta forma, foram adicionadas para casa objeto da coleção, as listas (*arrays*) ORDERS, que corresponde aos objetos JSON do ficheiro *orders.json*, e SHOPPING_SESSIONS, que corresponde aos objetos JSON do ficheiro *shopping.sessions*.

```

    _id:
    FIRST_NAME:
    MIDDLE_NAME:
    LAST_NAME:
    PHONE_NUMBER:
    EMAIL:
    USERNAME:
    USER_PASSWORD:
    REGISTERED_AT:
  ▾ SHOPPING_SESSIONS:
    ▸ 0: Object
  ▾ ORDERS: Array
    ▸ 0: Object

```

Figura 4.11: Nível 1 de aninhamento (JSON)

```

  ▾ SHOPPING_SESSIONS: Array
    ▾ 0: Object
      SESSION_ID:
      USER_ID:
      CREATED_AT:
      MODIFIED_AT:
      ▸ CART_ITEMS: Array
  ▾ ORDERS: Array
    ▾ 0: Object
      ORDER_DETAILS_ID:
      USER_ID:
      TOTAL:
      ▸ PAYMENT: Object
      SHIPPING_METHOD:
      ▸ DELIVERY_ADDRESS: Object
      CREATED_AT:
      MODIFIED_AT:
      ▸ ORDER_ITEMS: Array

```

Figura 4.12: SHOPPING_SESSION e ORDERS no nível 2 de aninhamento (JSON)

Na figura 4.13 é possível observar a estrutura aplicada à coleção products que armazena as informações sobre os produtos que a loja vende, incluindo, a categoria a que pertence, o desconto e *stock* do mesmo. Num primeiro nível de aninhamento em JSON, a informação base armazenada é relativa às informações da tabela PRODUCT.

```

    _id:
    PRODUCT_NAME:
    SKU:
    PRICE:
    CREATED_AT:
    LAST_MODIFIED:
  ▸ CATEGORY: Object
  ▸ DISCOUNT: Object
  ▸ STOCK: Object

```

Figura 4.13: Nível 1 de aninhamento (JSON)

Como referido anteriormente, para a criação desta coleção foram agrupadas as tabelas PRODUCT, PRODUCT_CATEGORIES, DISCOUNT e STOCK. Todas as relações entre a tabela PRODUCT e cada uma das restantes é de 1 para 1, isto é, a cada produto está apenas associado uma categoria, uma desconto e um *stock*. Assim sendo, foram criados objetos aninhados para cada uma destas relações, nomeadamente, um que armazena as informações relativas à categoria (CATEGORY), , outro que contém as informações sobre o desconto (DISCOUNT) e, por último, outro que guarda as informações sobre o *stock* do produto (STOCK).

```

▼ CATEGORY: Object
  CATEGORY_ID:
  CATEGORY_NAME:
▼ DISCOUNT: Object
  DISCOUNT_ID:
  DISCOUNT_NAME:
  DISCOUNT_DESC:
  DISCOUNT_PERCENT:
  IS_ACTIVE_STATUS:
  D_CREATED_AT:
  MODIFIED_AT:
▼ STOCK: Object
  QUANTITY:
  MAX_STOCK_QUANTITY:
  UNIT:

```

Figura 4.14: CATEGORY, DISCOUNT e STOCK no nível 2 de aninhamento (JSON)

Por último, a imagem 4.15 apresenta a estrutura da coleção `employees_archive` contém os dados de funcionários arquivados. Cada objeto JSON desta diz respeito aos dados originais da tabela `EMPLOYEES_ARCHIVE`, como mencionado anteriormente.

```

_id:
EVENT_DATE:
EVENT_TYPE:
USER_NAME:
OLD_EMPLOYEE_ID:
OLD_FIRST_NAME:
OLD_MIDDLE_NAME:
OLD_LAST_NAME:
OLD_DATE_OF_BIRTH:
OLD_DEPARTMENT_ID:
OLD_HIRE_DATE:
OLD_SALARY:
OLD_PHONE_NUMBER:
OLD_EMAIL:
OLD_SSN_NUMBER:
OLD_MANAGER_ID:
NEW_EMPLOYEE_ID:
NEW_FIRST_NAME:
NEW_MIDDLE_NAME:
NEW_LAST_NAME:
NEW_DATE_OF_BIRTH:
NEW_DEPARTMENT_ID:
NEW_HIRE_DATE:
NEW_SALARY:
NEW_PHONE_NUMBER:
NEW_EMAIL:
NEW_SSN_NUMBER:
NEW_MANAGER_ID:

```

Figura 4.15: Nível 1 de aninhamento (JSON)

4.3 Queries

Após a conclusão da importação dos dados em formato JSON para o MongoDB, o grupo prosseguiu com a execução das quatro interrogações base no novo paradigma de base de dados. Em comparação com o Oracle, as consultas tornaram-se significativamente mais simples, visto que não foi necessário realizar várias junções entre tabelas (operações JOIN), devido ao facto de todas as informações estarem muito mais agrupadas. Para além destas, também foram criadas duas interrogações adicionais com o intuito de explorar diferentes operações aplicadas nas aulas.

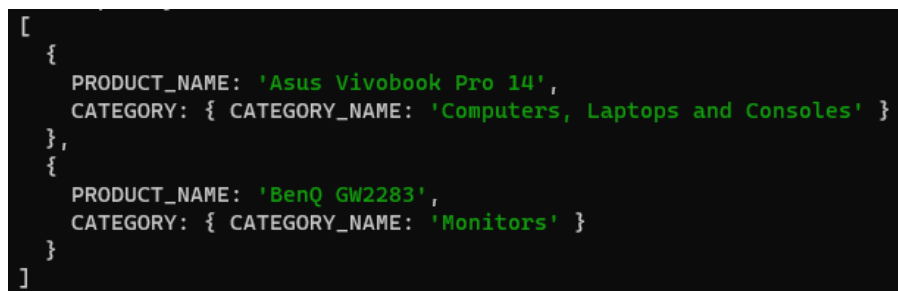
4.3.1 Queries base

Query 1

Obter o nome e a categoria de cada produto com um stock menor do que 10.

```
db.products.find(
  {"STOCK.QUANTITY": {"$lt": 10}},
  { "_id": 0, "PRODUCT_NAME": 1, "CATEGORY.CATEGORY_NAME": 1 }
).pretty()
```

Obtiveram-se, assim, os seguintes resultados:



```
[
  {
    PRODUCT_NAME: 'Asus Vivobook Pro 14',
    CATEGORY: { CATEGORY_NAME: 'Computers, Laptops and Consoles' }
  },
  {
    PRODUCT_NAME: 'BenQ GW2283',
    CATEGORY: { CATEGORY_NAME: 'Monitors' }
  }
]
```

Figura 4.16: Resultado da query base 1 (MongoDB)

Query 2

Listar todas as sessões de compra criadas pelo o usuário 'Evonne Warin' .

```
db.store_users.find(
  { "FIRST_NAME": "Evonne", "LAST_NAME": "Warin"},
  { "_id": 0, "SHOPPING_SESSIONS.SESSION_ID" : 1 , "SHOPPING_SESSIONS.USER_ID" : 1,
    "SHOPPING_SESSIONS.CREATED_AT": 1, "SHOPPING_SESSIONS.MODIFIED_AT": 1 }
).pretty()
```

Foram obtidos os seguintes resultados, após a execução da *query*:

```
[
  {
    SHOPPING_SESSIONS: [
      {
        SESSION_ID: 12,
        USER_ID: 12,
        CREATED_AT: '2022-01-14 02:12:10',
        MODIFIED_AT: '2022-01-14 08:19:22'
      }
    ]
  }
]
```

Figura 4.17: Resultado da query base 2 (MongoDB)

Query 3

Obter todos os utilizadores (username,phone_number,email) que fizeram pedidos de produtos na categoria 'Monitors' depois do mês de março de 2022.

```
db.store_users.aggregate([
  { $match: {
    $expr: {
      $gte: [
        { $dateFromString: { dateString: { $arrayElemAt: ["$ORDERS.CREATED_AT", 0] },
          format: "%Y-%m-%d %H:%M:%S" } },
        ISODate("2022-03-01T00:00:00Z")
      ]
    }
  } },
  { $lookup: {
    from: "products",
    localField: "ORDERS.ORDER_ITEMS.PRODUCT_ID",
    foreignField: "_id",
    as: "PRODUCTS"
  } },
  { $match: { "PRODUCTS.CATEGORY.CATEGORY_NAME": "Monitors" } },
  { $project: {
    _id: 0,
    USERNAME: 1,
    PHONE_NUMBER: 1,
    EMAIL: 1
  } }
]).pretty()
```

Esta *query* resultou nos seguintes valores:

```
[
  {
    PHONE_NUMBER: '565 270 7798',
    EMAIL: 'kwyche1@un.org',
    USERNAME: 'kwyche1'
  },
  {
    PHONE_NUMBER: '354 862 1868',
    EMAIL: 'ckilby7@sitemeter.com',
    USERNAME: 'ckilby7'
  },
  {
    PHONE_NUMBER: '197 771 6322',
    EMAIL: 'sbogeysa@deviantart.com',
    USERNAME: 'sbogeysa'
  }
]
```

Figura 4.18: Resultado da query base 3 (MongoDB)

Query 4

Média dos salários por departamento.

```
db.departments.aggregate([
  { $unwind: "$EMPLOYEES" },
  { $group: {
    _id: "$DEPARTMENT_NAME",
    averageSalary: { $avg: "$EMPLOYEES.SALARY" }
  } },
  { $project: {
    DEPARTMENT_NAME: "$_id",
    averageSalary: 1,
    _id: 0
  } }
]).pretty()
```

Após a execução da *query*, foram obtidos os seguintes resultados:

```
[
  {
    averageSalary: 7733.333333333333,
    DEPARTMENT_NAME: 'Sales department'
  },
  { averageSalary: 9300, DEPARTMENT_NAME: 'Management' },
  { averageSalary: 7866.666666666667, DEPARTMENT_NAME: 'Development' },
  {
    averageSalary: 5433.333333333333,
    DEPARTMENT_NAME: 'Purchase department'
  }
]
```

Figura 4.19: Resultado da query base 4 (MongoDB)

4.3.2 Queries Extra

Query 1

Obter o nome, a categoria e quantidade vendida dos 4 produtos mais comprados desde 2021.

```
db.store_users.aggregate([
  { $match: {
    $expr: {
      $gte: [
        { $dateFromString: { dateString: { $arrayElemAt: ["$ORDERS.CREATED_AT", 0] },
          format: "%Y-%m-%d %H:%M:%S" } } },
        ISODate("2021-01-01T00:00:00Z")
      ]
    }
  },
  { $unwind: "$ORDERS" },
  { $unwind: "$ORDERS.ORDER_ITEMS" },
  { $group: {
    _id: "$ORDERS.ORDER_ITEMS.PRODUCT_ID",
    totalQuantity: { $sum: 1 }
  } },
  { $sort: { totalQuantity: -1 } },
  { $limit: 4 },
  { $lookup: {
    from: "products",
    localField: "_id",
    foreignField: "_id",
    as: "product"
  } },
  { $project: {
    _id: 1,
    productName: { $arrayElemAt: ["$product.PRODUCT_NAME", 0] },
    productCategory: { $arrayElemAt: ["$product.CATEGORY.CATEGORY_NAME", 0] },
    totalQuantity: 1
  } }
]).pretty()
```

Executando a *query* anterior, obtém-se os seguintes resultados:


```
[
  {
    _id: 3,
    totalQuantity: 4,
    productName: 'JBL Quantum 600',
    productCategory: 'Headphones and Speakers'
  },
  {
    _id: 9,
    totalQuantity: 3,
    productName: 'Logitech MK220',
    productCategory: 'Keyboard and mouse'
  },
  {
    _id: 6,
    totalQuantity: 3,
    productName: 'Xiaomi Redmi Buds 3 Lite',
    productCategory: 'Headphones and Speakers'
  },
  {
    _id: 2,
    totalQuantity: 3,
    productName: 'TV Samsung UE65A DVB',
    productCategory: 'TV and Video'
  }
]
```

Figura 4.20: Resultado da query extra 1 (MongoDB)

Query 2

Listar todos os produtos (product_name, price, category_name, discount_name, stock) que estão com uma promoção ativa de 15 por cento e ordená-los pelo o seu preço.

```
db.products.aggregate([
  { $match: { "DISCOUNT.IS_ACTIVE_STATUS": "Y", "DISCOUNT.DISCOUNT_PERCENT": 15 } },
  { $project: {
    "_id": 0,
    "PRODUCT_NAME": 1,
    "PRICE": 1,
    "CATEGORY.CATEGORY_NAME": 1,
    "DISCOUNT.DISCOUNT_NAME": 1,
    "STOCK.QUANTITY": 1
  }
},
  { $sort: { "PRICE": 1 } }
]).pretty()
```

Foram obtidos os seguintes resultados:

```
[
  {
    PRODUCT_NAME: 'Samsung Galaxy A52',
    PRICE: 1199,
    CATEGORY: { CATEGORY_NAME: 'Smartphones and Smartwatches' },
    DISCOUNT: { DISCOUNT_NAME: 'Hits of the Week' },
    STOCK: { QUANTITY: 169 }
  },
  {
    PRODUCT_NAME: 'Acer Aspire TX Turbo 2',
    PRICE: 1999,
    CATEGORY: { CATEGORY_NAME: 'Computers, Laptops and Consoles' },
    DISCOUNT: { DISCOUNT_NAME: 'Hits of the Week' },
    STOCK: { QUANTITY: 59 }
  },
  {
    PRODUCT_NAME: 'ASUS X515-BQ26W 8GB RAM 256GB SSD',
    PRICE: 2399,
    CATEGORY: { CATEGORY_NAME: 'Computers, Laptops and Consoles' },
    DISCOUNT: { DISCOUNT_NAME: 'Hits of the Week' },
    STOCK: { QUANTITY: 47 }
  },
  {
    PRODUCT_NAME: 'Xbox Series X + Halo Infinite',
    PRICE: 2697,
    CATEGORY: { CATEGORY_NAME: 'Computers, Laptops and Consoles' },
    DISCOUNT: { DISCOUNT_NAME: 'Hits of the Week' },
    STOCK: { QUANTITY: 499 }
  }
]
```

Figura 4.21: Resultado da query extra 2 (MongoDB)

Capítulo 5

Base de Dados Orientada a Grafos – Neo4j

5.1 Migração dos dados

Para realizar a migração dos dados da base de dados relacional Oracle para a base de dados não relacional orientada a grafos, Neo4j, o grupo procedeu às seguintes etapas. Primeiro, os dados originais foram exportados da base de dados relacional para arquivos CSV através de consultas SQL e da funcionalidade de exportação disponibilizada pelo SQLDeveloper. Durante este processo, foram criados catorze ficheiros CSV diferentes, cada um com a informação de uma determinada tabela do modelo relacional.

De seguida, a estratégia aplicada pelo o grupo foi criar um *script* em python (`db_neo4j.py`) que migra-se os dados dos ficheiro CSV para a base de dados Neo4j. Isto é conseguido estabelecendo uma conexão à base de dados não relacional. E, posteriormente, importando os dados dos ficheiros CSV e exportando-os usando a linguagem nativa do Neo4j, Cypher, criando, assim, os nodos e seus relacionamentos.

Este *script* também remove as chaves estrangeiras dos nodos. Isto é, depois de criar os relacionamentos, o grupo concluiu que estas chaves não eram necessárias ser guardadas no nó, uma vez que é criada uma relação entre estes, o que corresponde à mesma funcionalidade que a da chave estrangeira.

Para além disso, foram criados os mesmos índices existentes no modelo relacional. Apesar destes não serem estritamente necessários para o bom funcionamento da base de dados, o objetivo era migrar o máximo conteúdo possível do modelo relacional.

Na *view* é guardada a query para correr num *script* em python ligado à base de dados. Outra possível solução era utilizar a biblioteca APOC (Awesome Procedures On Cypher) e tratar a query como um *user defined procedure*.

A criação de *procedures* pode ser feita de duas formas: criar uma classe em java com a lógica do *procedure* e compilar para um ficheiro .JAR e registar o ficheiro como um plug-in no neo4j, ou utilizar a biblioteca APOC para guardar o *procedure* com `'apoc.custom.asProcedure'` e chamá-lo com `'custom.*procedure_name*'`. O uso da java API é, em geral, mais adequado para *procedures* mais complexos, e o uso do APOC melhor para métodos mais simples.

Para criar um *trigger* pode ser usada a biblioteca APOC novamente e chamar 'apoc.trigger.add' para a sua criação.

A criação de índices, *views*, *procedures* e do *trigger* está explicado e exemplificado no ficheiro neo4j.txt .

5.2 Estruturação dos dados

Para a base de dados não relacional orientada a grafos, a estrutura escolhida pelo o grupo foi cada linha das tabelas corresponder a um nó com as suas respetivas propriedades, isto é, em cada nó é armazenada a informação da linha da tabela correspondente. Para os relacionamentos entre estes, a estrutura e abordagem adotada foi utilizar a chave primária e estrangeira entre as tabelas para o criar o relacionamento.

A figura 5.1 apresenta o modelo lógico da base de dados Neo4j:

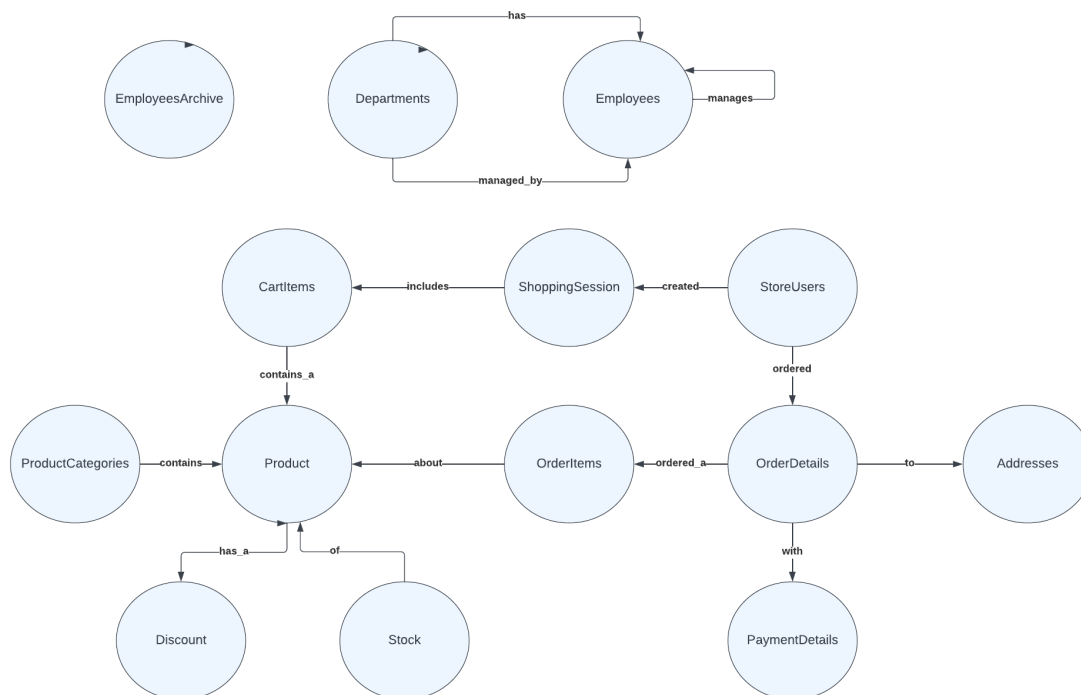


Figura 5.1: Modelo lógico da base de dados OES (Neo4j)

Para algumas das relações entre os nodos, o grupo estava na dúvida de qual direção esta deveria tomar. Isto é, pensado, por exemplo, no caso da relação entre os nós StoreUsers e ShoppingSession. Inicialmente, havia uma incerteza se a melhor relação era "StoreUsers-creates-¿ShoppingSession" ou "ShoppingSession-created.by-¿StoresUsers". Desta forma, o pensamento aplicado para estas situações foi pensar em possíveis operações sobre os dados e, desse modo, analisar qual das relações tornariam estas menos pesadas. Aplicando este pensamento ao caso de exemplo, a conclusão de uma possível operação foi que ver as sessões que o utilizador criou é mais rápido do que ver o utilizador que criou a sessão, devido a, em princípio, haver mais sessões do que utilizadores. Desta forma, criar a primeira opção de relação permitiria uma melhor *performance*

que a segunda opção.

5.3 Queries

Após a conclusão da criação e população da base de dados, o grupo procedeu para a execução das *queries*. Inicialmente, foram adaptadas para a sintaxe do Cypher e implementadas as quatro consultas base. O objetivo destas interrogações base era analisar o impacto dos diferentes paradigmas no seu desempenho, compreender qual seria o mais adequado e porquê. Posteriormente, e como nas outras bases de dados, também foram desenvolvidas mais duas *queries* adicionais com o propósito de analisar os vários tipos de operações aplicadas nas aulas.

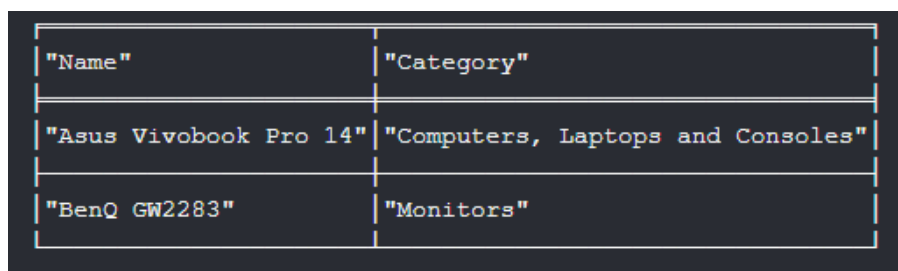
5.3.1 Queries base

Query 1

Obter o nome e a categoria de cada produto com um stock menor do que 10.

```
MATCH (pc:ProductCategories)-[:contains]->(p:Product)<-[:of]-(s:Stock)
WHERE s.quantity < 10
RETURN p.product_name AS Name, pc.category_name AS Category
```

Após a execução da *query*, foram obtidos os seguintes resultados:



"Name"	"Category"
"Asus Vivobook Pro 14"	"Computers, Laptops and Consoles"
"BenQ GW2283"	"Monitors"

Figura 5.2: Resultado da query base 1 (Neo4j)

Query 2

Listar todas as sessões de compra criadas pelo o usuário 'Evonne Warin' .

```
MATCH (su:StoreUsers {first_name: "Evonne", last_name: "Warin"})-[:created]->(ss:
ShoppingSession)
RETURN su.user_id AS User_id, ss.session_id AS Session_id, ss.created_at
AS Created_at, ss.modified_at as Modified_at
```

Foi obtido o seguinte resultado:

"User_id"	"Session_id"	"Created_at"	"Modified_at"
12	12	"22.01.14 02:12:10,000000000"	"22.01.14 08:19:22,000000000"

Figura 5.3: Resultado da query base 2 (Neo4j)

Query 3

Obter todos os utilizadores (username,phone_number,email) que fizeram pedidos de produtos na categoria 'Monitors' depois do mês de março de 2022.

```
MATCH (su:StoreUsers)-[o:ordered]->(od:OrderDetails)-[:ordered_a]->(oi:OrderItems)
-[:about]->(p:Product)<-[:contains]-(pc:ProductCategories)
WHERE pc.category_name = "Monitors" AND
      toInteger(substring(od.created_at, 6, 2)) >= 1 AND
      toInteger(substring(od.created_at, 3, 2)) >= 3 AND
      toInteger(substring(od.created_at, 0, 2)) = 22
RETURN su.username AS Username, su.phone_number AS Phone_Number, su.email AS Email
```

Executando a *query* obtemos o seguinte resultado:

"Username"	"Phone_Number"	"Email"
"sbogeysa"	"197 771 6322"	"sbogeysa@deviantart.com"
"ckilby7"	"354 862 1868"	"ckilby7@sitemeter.com"
"kwyche1"	"565 270 7798"	"kwyche1@un.org"

Figura 5.4: Resultado da query base 3 (Neo4j)

Query 4

Média dos salários por departamento.

```
MATCH (d:Departments)-[:has]->(e:Employees)
RETURN d.department_name AS Department, AVG(e.salary) AS Average_Salary
```

Obtiveram-se, assim, os seguintes resultados:

"Department"	"Average_Salary"
"Management"	9300.0
"Development"	7866.666666666667
"Purchase department"	5433.333333333333
"Sales department"	7733.333333333333

Figura 5.5: Resultado da query base 4 (Neo4j)

5.3.2 Queries Extra

Query 1

Obter as 5 províncias com maior quantidade de vendas.

```
MATCH (a:Addresses)-[t:to]-(od:OrderDetails)
RETURN a.province AS Province, count(od) AS Sales_Quantity
ORDER BY Sales_Quantity DESC
LIMIT 5;
```

Esta *query* resultou no seguinte valor:

"Province"	"Sales_Quantity"
"California"	2
"New York"	2
"Pennsylvania"	2
"Florida"	2
"Oregon"	1

Figura 5.6: Resultado da query extra 1 (Neo4j)

Query 2

Listar todos os utilizadores que escolheram o mesmo *shipping_method*.

```
MATCH (su1:StoreUsers)-[:ordered]->(od1:OrderDetails)
MATCH (su2:StoreUsers)-[:ordered]->(od2:OrderDetails)
WHERE su1 <> su2 AND od1.shipping_method = od2.shipping_method
WITH COLLECT(DISTINCT su1.username) AS Usernames, od1.shipping_method
AS Shipping_Method
RETURN Usernames, Shipping_Method, SIZE(Usernames) AS Number_Of_Users;
```

Após a execução da *query*, foram obtidos os seguintes resultados:

"Usernames"	"Shipping_Method"	"Number_Of_Users"
["jmccconville5", "skelly6", "bbrecon8", "owarinb", "gburb", "oroughf", "ccabell0"]	"Inpost"	6
["ldendle4", "ckilby7", "zmenco9", "ebolesc", "aspreage", "kwyche1"]	"DPD"	6
["hpelham3", "qsaddletonh", "jmonaghan2"]	"UPS"	3
["mkefordd", "lgrinnikovg", "mwildsi", "sbogeysa"]	"DHL"	4

Figura 5.7: Resultado da query extra 2 (Neo4j)

Capítulo 6

Comparação do desempenho das queries

Ao longo do trabalho prático, foram desenvolvidas quatro queries base. Estas quatro consultas foram executadas em todas as bases de dados utilizadas, com o intuito de, no fim, comparar o seu desempenho e analisar o efeito das características de cada sistema sobre o desempenho destas. Depois de implementadas estas interrogações, foi retirado os tempos de execução das mesmas para a comparação entre as diferentes bases de dados.

No caso do Oracle e do Neo4j, esses sistemas já realizam, automaticamente, a cronometragem da execução das consultas, apresentando os tempos com os resultados. No entanto, para o MongoDB, foi necessário implementar a um *script* em python (*extract_time.py*) que calcula o tempo de cada *query*, como por exemplo:

6.1 Query 1

	Oracle	MongoDB	Neo4j
Query 1	29ms	1ms	2ms

Tabela 6.1: Resultados obtidos na query 1.

A query 1 obtém o nome e a categoria de cada produto com um stock menor do que 10.

No Oracle, esta consulta envolve combinação de três tabelas: `PRODUCT`, `PRODUCT_CATEGORIES` e `STOCK`, sendo que estas duas últimas estão diretamente relacionadas com a primeira. Isto permite não ser preciso realizar muitas combinações de chaves. No entanto, as tabelas de `JOIN` adicionam complexidade, o que aumenta o tempo de execução da consulta.

A base de dados MongoDB apresenta um bom desempenho, uma vez que esta *query* acede a uma única coleção devido à reestruturação aplicada a quando da migração dos dados da base de dados relacional para a não relacional orientada a grafos.

O Neo4j também apresenta uma resultado favorável. Quando as consultas envolvem percorrer várias arestas, este atinge uma melhor *performance* na execução das mesmas. No caso desta

interrogação, foram utilizados os nodos com as informações dos produtos, das categorias e do stock destes. Sendo que foi necessário percorrer todas os relacionamentos criados entre os nodos Product e ProductCategories e, os nodos Product e Stock, o que permitiu obter um bom tempo de execução da interrogação

6.2 Query 2

	Oracle	MongoDB	Neo4j
Query 2	24ms	0.12ms	2ms

Tabela 6.2: Resultados obtidos na query 2.

A *query 2* lista todas as sessões de compra criadas pelo o utilizador 'Evonne Warin'.

No Oracle, esta interrogação apenas utiliza novamente a combinação de apenas duas tabelas, SHOPPING_SESSION e STORE_USERS, que estão diretamente relacionadas. Desta forma, não é necessário realizar muitas combinações de chaves, o que é uma vantagem para o Oracle. Isto permite que a consulta contenha um tempo de execução aceitável.

Para a base de dados não relacional orientada a documentos, o MongoDB, os dados utilizados pela *query* estavam mais centralizados, uma vez que para este modelo os dados foram reestruturados de modo a maximizar a utilização deste paradigma. As informações necessárias à *query* estão organizadas de acordo com os utilizadores, isto é, cada objeto JSON desta coleção corresponde a um utilizador diferente, o que permitiu obter um bom desempenho do tempo de execução.

Por fim, o Neo4j é um sistema de base de dados com foco para consultas relacionais. Este obtém uma boa *performance* ao executar consultas que envolvem a travessia de relacionamentos.

6.3 Query 3

	Oracle	MongoDB	Neo4j
Query 3	314ms	6.8ms	86 ms

Tabela 6.3: Resultados obtidos na query 3.

A *query 3* lista todos os utilizadores (username, phone_number, email) que fizeram pedidos de produtos na categoria 'Monitors' depois do mês de março de 2022. Tal como esperado, esta interrogação teve uma melhor performance nas bases de dados não relacionais do que na base de dado relacional.

No caso do Oracle, a informação é armazenada em várias tabelas relacionadas por chaves estrangeiras. A execução desta query específica envolvia realizar várias combinações de chaves devido à presença de várias tabelas intermediárias entre as tabelas STORE_USERS e PRODUCT_CATEGORIES. Deste modo, como as tabelas de JOIN adicionam complexidade, estas tiveram um impacto negativo no desempenho da consulta.

No MongoDB, os dados foram organizados de acordo com os utilizadores registados no site e os produtos que a loja vende. Como resultado, o tempo de execução da *query* é menor devido à reorganização dos dados, uma vez que reestruturação centralizou significativamente as informações utilizadas pela consulta, o que facilitou o acesso a elas.

O Neo4j é um banco de dados orientado a relacionamentos, projetado para lidar com consultas que envolvem a travessia de múltiplos relacionamentos entre os nós. Na *query* existem o mesmo número de nós intermediários entre os nós de StoreUsers e ProductCategories que existiam de tabelas intermediárias na base de dados Oracle. No entanto, o Neo4j percorre esses relacionamentos de forma eficiente, resultando em um tempo de execução mais rápido. Apesar de existir um tempo adicional necessário para criar a representação visual desses relacionamentos, que, por sua vez, pode afetar o desempenho da consulta.

6.4 Query 4

	Oracle	MongoDB	Neo4j
Query 4	13ms	3.55ms	6 ms

Tabela 6.4: Resultados obtidos na query 4.

A query 4 determina a média de salários por departamentos.

No Oracle, base de dados relacional, essa consulta envolve, novamente, apenas a combinação de duas tabelas, EMPLOYEES e DEPARTMENTS, que estão diretamente relacionadas. Isso significa que não era necessário realizar muitas combinações de chaves, o que é benéfico para o Oracle. No entanto, as operações de agregação afetaram um pouco o desempenho da consulta, pois não são o ponto forte dos bancos de dados relacionais. Apesar disso, o Oracle conseguiu executar a consulta em um tempo razoável.

No caso do MongoDB, os dados utilizados pela *query* estavam mais centralizados, uma vez que, neste modelo, as informações foram organizadas de acordo com os departamentos. Ou seja, cada documento da base de dados corresponde a um departamento diferente. Isto permitiu obter um tempo de execução baixo para a *query*, devido à estrutura adotada para esta coleção.

Por fim, o Neo4j é um sistema de banco de dados orientado a grafos, ou seja, projetado para consultas relacionais. O Neo4j alcança um bom desempenho ao executar consultas que envolvem a travessia de muitas arestas (relacionamentos). No caso desta interrogação, apenas foram necessários os nodos dos funcionários e departamentos.

Capítulo 7

Conclusão

Neste projeto, foi realizado com sucesso a migração de uma base de dados relacional para dois esquemas não relacionais: MongoDB e Neo4j. Assim, foi possível explorar as vantagens oferecidas por essas bases de dados específicas, permitindo uma melhor modelagem e consulta dos dados.

No caso do MongoDB, uma base de dados orientada a documentos, a migração permitiu criar uma estruturação flexível dos dados, com a capacidade de armazenar documentos JSON de forma hierárquica. Isso possibilita consultas eficientes e escalabilidade horizontal, tornando-a ideal para cenários em que a estrutura dos dados é variável ou sujeita a alterações frequentes.

Por outro lado, a migração para o Neo4j, uma base de dados orientada a grafos, permitiu explorar as relações entre os dados de forma mais eficiente. O Neo4j oferece um modelo de dados flexível, que pode representar relacionamentos complexos entre entidades, como conexões sociais, redes de transporte ou, possivelmente para este caso, recomendações de compra personalizadas. Com a migração, foi possível aproveitar a capacidade do Neo4j de armazenar dados em forma de nós, relacionamentos e propriedades, facilitando consultas e análises de dados com foco nas relações.

Ao utilizar diferentes bases de dados não relacionais, foram expandidas as opções de armazenamento e consulta dos dados, adaptando-as de acordo com a natureza das mesmas e os requisitos específicos do projeto. Isso proporcionou uma maior flexibilidade e desempenho em comparação com a abordagem tradicional de base de dados relacional.

É importante ressaltar que a escolha entre uma base de dados relacional e uma base de dados não relacional depende das características dos dados e dos requisitos do projeto. Cada tipo possui vantagens e desvantagens específicas, e é fundamental considerar fatores como a estrutura dos dados, o volume, o tipo de consultas mais frequentes e a escalabilidade necessária para tomar a decisão adequada.

No geral, a migração bem-sucedida da base de dados relacional para os esquemas não relacionais MongoDB e Neo4j permite tirar proveito das características e benefícios oferecidos por cada um deles.