Universidade do Minho
Mestrado Integrado em Engenharia Informática

Unidade Curricular de
Bases de Dados NOSQL

Ano Letivo de 2020/2021

Recursos Humanos

Filipa Alves dos Santos A83631

Hugo André Coelho Cardoso A85006

João da Cunha e Costa A84775

Válter Ferreira Picas Carvalho A84464

Janeiro, 2021

Data de Recepção	
Responsável	
Avaliação	
Observações	

Recursos Humanos

Filipa Alves dos Santos A83631

Hugo André Coelho Cardoso A85006

João da Cunha e Costa A84775

Válter Ferreira Picas Carvalho A84464

Janeiro, 2021

Resumo

Este trabalho prático foi efetuado no âmbito da unidade curricular Bases de Dados NOSQL, no 1º semestre do 4º ano do Mestrado Integrado de Engenharia Informática, no ano letivo de 2020/21.

O projeto consistiu na análise, planeamento e implementação de um Sistema de Base de Dados (SGBD) relacional e dois não relacionais. Para tal, foi fornecido a base de dados relacional HR (Recursos Humanos), disponibilizada no Oracle Database, que serviu de base para o resto do trabalho.

Neste relatório, começa-se por fazer uma introdução do projeto proposto, com uma contextualização e explicação do caso de estudo em questão e, de seguida, é explicado a base de dados dos Recursos Humanos fornecida. Posteriormente, é descrido a abordagem tomada em cada uma das Bases de Dados utilizadas, bem como as *queries* desenvolvidas e testadas de modo a testar e comparar a eficácia delas nos diversos sistemas.

O principal objetivo deste trabalho foi praticar o desenvolvimento de diferentes Bases de Dados, relacionais e NOSQL, e perceber as suas diferenças para facilitar a escolha do SGBD ideal em futuros projetos.

Área de Aplicação: Conceptualização, desenvolvimento e implementação de Sistemas de Bases de Dados relacionais e não relacionais.

Palavras-Chave: Sistemas de Bases de Dados, Bases de Dados Relacionais, Bases de Dados Não Relacionais, SQL, NOSQL, Oracle, MongoDB, Neo4j, CSV, JSON, HR.

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

1.1. Contextualização

Este trabalho foi realizado no contexto da unidade curricular Bases de Dados NOSQL, com o objetivo de desenvolver as competências dos alunos na disciplina e na utilização e implementação de diferentes tipos de bases de dados.

Estas competências são importantes sendo que a migração entre diferentes bases de dados é um acontecimento comum em situações reais devido a mudanças dos softwares que as utilizam. Cada tipo (relacional, orientado a documentos, orientado a grafos) tem as suas vantagens e desvantagens, dependendo do que se pretende alcançar e é importante ter esta intuição de qual será mais apropriado num certo contexto.

Para além da migração e desenvolvimento dos SGBDs, também é útil para praticar a escrita de *queries* de acordos com os diferentes paradigmas, o que é necessário para comparação e avaliação dos tempos e uma posterior conclusão a partir destes.

1.2. Apresentação do Caso de Estudo

Neste trabalho prático, é proposto a análise, planeamento, e implementação de diferentes SGBDs, a partir da base de dados HR disponibilizada pela Oracle. A HR armazena dados de uma aplicação de Recursos Humanos e todos os seus atributos são explicados em maior pormenor na secção 2.

1.3. Estrutura do Relatório

O relatório encontra-se dividido em sete capítulos:

1) Capítulo 1 - **Introdução**: contextualização e descrição do problema proposto no trabalho.
2) Capítulo 2 - **HR**: descrição das diversas tabelas da base de dados HR.

3) Capítulo 3 - **Escolhas das Bases de Dados Não-Relacionais**: explicação das escolhas do MongoDB e Neo4j como as bases de dados não relacionais utilizadas.
4) Capítulo 4 - **Base de Dados Relacional - Oracle**: explicação do modelo de dados, da migração e *queries* feitas usando uma base de dados desenvolvida para Oracle.
5) Capítulo 5 - **Base de Dados Orientada a Documentos - MongoDB**: explicação do modelo de dados, da migração e *queries* feitas usando uma base de dados orientada a documentos, desenvolvida com MongoDB.
6) Capítulo 6 - **Base de Dados Orientada a Grafos - Neo4j**: explicação do modelo de dados, da migração e *queries* feitas usando uma base de dados orientada a grafos (Neo4j).
7) Capítulo 7 **- Comparação do Desempenho das Queries Transversais**: comparação do desempenho das *queries* feitas em todas as bases de dados criadas.
8) Capítulo 8 **- Conclusão**: breve sumário e análise do projeto realizado.

2. HR

O HR é uma base de dados de uma aplicação de Recursos Humanos, criado pela Oracle, que tem como objetivo principal armazenar os dados dos empregados de uma organização, sendo bastante útil para as empresas gerirem informação acerca dos seus empregados.

2.1. Esquema lógico

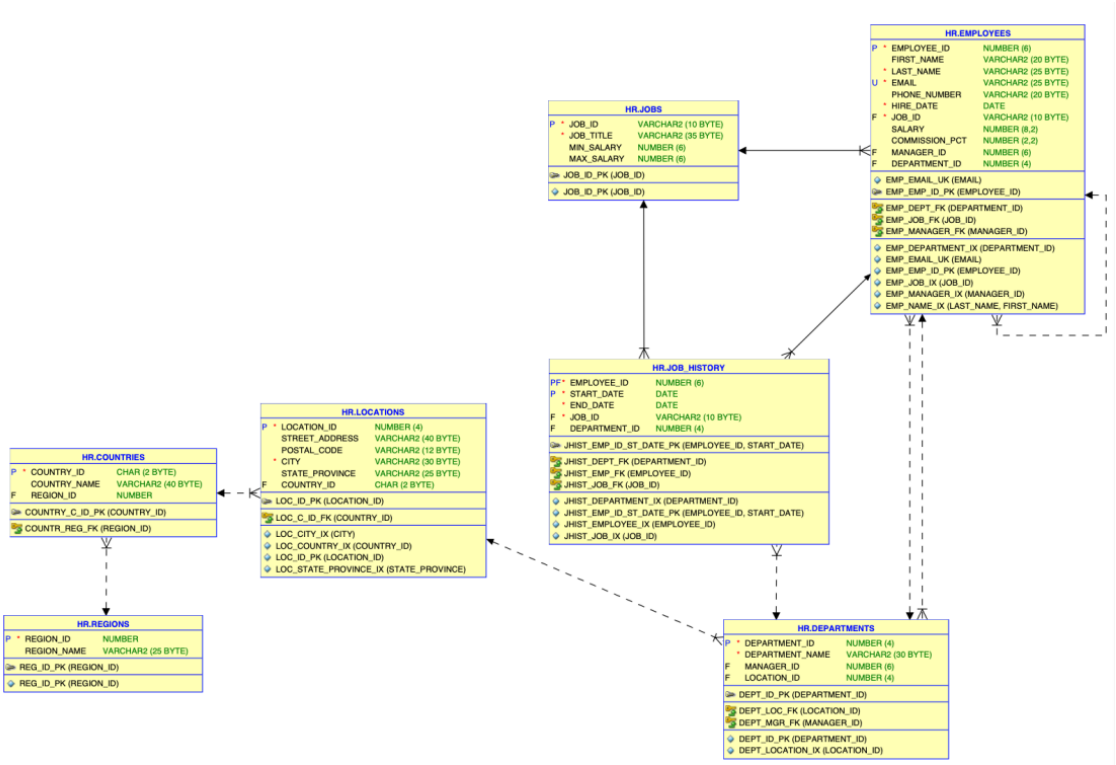

Figura 1 - Esquema Lógico da base de dados HR

O esquema lógico do HR é constituído por 7 tabelas distintas: **EMPLOYEES, JOBS, JOB_HISTORY, DEPARTMENTS, LOCATIONS, COUNTRIES, REGIONS.**

A tabela **EMPLOYEES** contém a lista de todos os empregados, incluindo informações referentes ao primeiro e último nome do empregado, o seu e-mail, número de telemóvel, data em que foi contratado, o seu salário e a sua percentagem de comissão. Esta tabela possui também, como chaves estrangeiras, informação relativa ao seu supervisor, departamento onde trabalha e também do trabalho que cada empregado tem.

Já na tabela **JOBS** armazena-se a lista de todos os empregos que a organização disponibiliza ou já disponibilizou, incluindo informações sobre o nome do emprego e do teto salarial que possui.

De seguida, temos a tabela **JOB_HISTORY**, que contém informação referente ao histórico de empregos que existiram na organização, incluindo informação do início e fim da execução desse emprego. Inclui também chaves estrangeiras com informação acerca do empregado associado a este emprego e do departamento em que este se insere.

No caso da tabela **DEPARTMENT*S***, esta contém dados relativos aos departamentos, contendo o nome do departamento e chaves estrangeiras com informação sobre o supervisor de cada departamento e localização dos mesmos.

Na tabela **LOCATIONS** temos informação sobre a localização de cada departamento, que contêm informação como a rua, código postal, cidade, estado/província em que cada departamento se situa, incluindo também uma chave estrangeira com informação do país.

Já a tabela **COUNTRIES** refere-se á lista dos países em que cada departamento se situa, contendo informação acerca do nome do país e uma chave estrangeira com a região em que se localiza certo país

Por fim, temos a tabela **REGIONS** que armazena informação das regiões de cada país.

3. Escolha das Bases de Dados Não-Relacionais

Um dos requisitos iniciais do projeto era a escolha de duas bases de dados não-relacionais para realizar a exportação dos dados presentes numa base de dados relacional, em que uma delas teria de ser orientada a documentos e a outra orientada a grafos.

As bases de dados relacionais têm várias desvantagens quando comparadas às bases de dados não-relacionais, entre elas a escalabilidade horizontal, que é praticamente não existente, a necessidade de modelos rígidos para a inserção de informação cuja eventual mudança obriga a uma reestruturação total da base de dados. Esta reestruturação muitas vezes leva à migração dos dados para uma nova base de dados com o modelo lógico atualizado. Para além disto, têm também baixa performance em operações de agregação que envolvem várias tabelas em simultâneo, pelo facto de terem de aceder a vários blocos no disco sempre que são invocadas. Por outro lado, as bases de dados não-relacionais permitem esquemas lógicos arbitrários, o que leva a uma maior flexibilidade no armazenamento da informação. Apresentam, também, uma enorme escalabilidade horizontal porque não guardam os dados em tabelas como nas bases de dados relacionais. Neste paradigma não-relacional, cada elemento do sistema (possivelmente) distribuído pode manter informação relativa a cada unidade (por exemplo documentos numa base de dados documental).

Assim, para a base de dados documental, o grupo escolheu o **MongoDB** uma vez que, para além de oferecer as vantagens descritas acima, aceita a importação de dados em formatos **JSON**, que pode ser manipulado para ter a estrutura inicial que seja necessária. Assim, é bastante flexível na organização dos dados. Como nesta base de dados **HR,** em particular, é possível agrupar os dados das restantes tabelas por departamento, é bastante vantajoso utilizar este paradigma por motivos de compatibilidade direta e facilidade de exportação. Para além desta vantagem, o **MongoDB** permite uma performance bastante elevada no que toca a operações que seriam de agregação numa base de dados relacional, já que cada documento mantém a informação necessária para as realizar.

Para a base de dados orientada a grafos, o grupo decidiu utilizar **Neo4j,** uma vez que este é focado em pesquisas relacionais, permitindo um melhor desempenho em interrogações que necessitem de vários relacionamentos em simultâneo, o que contrasta com as bases de dados relacionais na medida em que as últimas exibem um pior desempenho quando se necessita de realizar este tipo de operações, já que têm de realizar múltiplas combinações de chaves. Assim, em vez de utilizarem as tabelas todas para obter os dados pertinentes, utiliza diretamente os relacionamentos entre os nodos, poupando assim tempo de interpretação de resultados não relevantes. Permite, também, uma maior compactação de dados nos casos de relações N para N, uma vez que não necessita de um nodo intermediário para a ligação entre os nodos.

O grupo desenvolveu 3 *queries* de base que seriam implementadas em todas as bases de dados usadas, de maneira a poder comparar a sua performance no fim e observar o efeito das características de

cada BD sobre o desempenho das *queries*, debatendo a adequação dos sistemas de gestão de bases de dados escolhidos para as consultas em questão.

Além disso, foram criadas ainda mais 2 *queries* extra para cada base de dados usada, projetadas especialmente para a BD em questão, de maneira a capitalizar nas suas vantagens.

Desta forma, é estabelecida uma base de comparação entre as várias bases de dados usadas, bem como uma plataforma para exibir o seu potencial, quando são adequadas para as *queries* a realizar.

4. Base de Dados Relacional - OracleDB

4.1. Importação dos Dados

A fase inicial do projeto é importar todos os dados disponibilizados da base de dados **HR** para **Oracle**, que atuará como SGBD relacional para as várias tabelas já vistas anteriormente.

Para este efeito, foram utilizados os ficheiros de inicialização disponibilizados no ficheiro "*hr.zip*" que continham todos os *scripts* de criação da base de dados assim como a sua população de modo a obter uma base de trabalho para efeitos de interrogações.

Foi utilizado o **SQLDeveloper** para realizar a conexão à base de dados e realizar as interrogações necessárias sobre as tabelas disponíveis, no entanto, foi necessário realizar algumas alterações à formatação de visualização de alguns dados (valores decimais e datas) devido a erros na exportação para CSV, que será visto com mais detalhe em secções mais à frente.

4.2. Queries

De modo a estabelecer uma base de comparação, será utilizado um conjunto de três *queries* que será repetido em secções seguintes nas várias linguagens de interrogação disponibilizadas pelos diferentes paradigmas. Para além destas *queries* base, foram pensadas duas extra que visam tirar partido das capacidades do **Oracle SQL**.

4.2.1 Queries Transversais

Query 1

Média de salários por departamento:

```
select d.department_name as Departamento, round(avg(e.salary),2) as Media_Salario from employees e
inner join departments d on e.department_id=d.department_id
group by d.department_name
order by Media_Salario desc;
```

Figura 1 - *Query* 1 Transversal (SQL)

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

	DEPARTAMENTO	MEDIA_SALARIO
1	Executive	19333,33
2	Accounting	10154
3	Public Relations	10000
4	Marketing	9500
5	Sales	8955,88
6	Finance	8601,33
7	Human Resources	6500
8	IT	5760
9	Administration	4400
10	Purchasing	4150
11	Shipping	3475,56

Figura 2 - Resultado da *query* 1 transversal (SQL)

Query 2

Top 5 empregados com o contrato ativo mais duradouro em cada região:

```
select Regiao, Nome, Data_Inicio
from (select r.region_name as Regiao, concat(concat(e.first_name,' '), e.last_name) as Nome, e.hire_date as Data_Inicio,
      rank() over (partition by r.region_name order by e.hire_date asc, concat(concat(e.first_name,' '), e.last_name) asc) rank
      from employees e
        inner join departments d on e.department_id=d.department_id
        inner join locations l on l.location_id=d.location_id
        inner join countries c on c.country_id=l.country_id
        inner join regions r on r.region_id=c.region_id
     )
where rank <= 5;
```

Figura 3 - *Query* 2 Transversal (SQL)

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

	REGIAO	NOME	DATA_INICIO
1	Americas	Lex De Haan	2001-01-13 00:00:00
2	Americas	Shelley Higgins	2002-06-07 00:00:00
3	Americas	William Gietz	2002-06-07 00:00:00
4	Americas	Daniel Faviet	2002-08-16 00:00:00
5	Americas	Nancy Greenberg	2002-08-17 00:00:00
6	Europe	Hermann Baer	2002-06-07 00:00:00
7	Europe	Susan Mavris	2002-06-07 00:00:00
8	Europe	Janette King	2004-01-30 00:00:00
9	Europe	Patrick Sully	2004-03-04 00:00:00
10	Europe	Ellen Abel	2004-05-11 00:00:00

Figura 4 - Resultado da *query* 2 transversal (SQL)

Query 3

Emprego antigo em que o utilizador com ID 176 faturou mais:

```
select j.JOB_TITLE as Emprego, jh.start_date as DataInicial, jh.END_DATE  as DataFinal,
Round((MONTHS_BETWEEN (jh.end_date,jh.start_date))*j.MAX_SALARY,2) as Faturado from EMPLOYEES e
inner join JOB_HISTORY jh on e.EMPLOYEE_ID = jh.EMPLOYEE_ID
inner join JOBS j on jh.JOB_ID = j.JOB_ID
inner join Departments d on d.department_id = jh.department_id
where e.EMPLOYEE_ID = 176
order by (MONTHS_BETWEEN (jh.end_date,jh.start_date))*j.MAX_SALARY desc
fetch first 1 rows only;
```

Figura 5 - *Query* 3 Transversal (SQL)

Esta *query* resultou no seguinte valor:

	EMPREGO	DATAINICIAL	DATAFINAL	FATURADO
1	Sales Manager	2007-01-01 00:00:00	2007-12-31 00:00:00	240312,26

Figura 6 - Resultado da *query* 3 transversal (SQL)

4.2.2 Queries extra

Query 1

Total de empregados que trabalharam em cada departamento:

```
with employees_by_dep as (
    select d.department_id, jh.employee_id from Departments d, Job_history jh where jh.department_id = d.department_id
    union all
    select d.department_id, e.employee_id from Departments d, Employees e where e.department_id = d.department_id)
select department_id as Departamento, count(distinct(employee_id)) as Total
from employees_by_dep
group by department_id
order by Total desc, department_id;
```

Figura 7 - *Query* 1 Extra (SQL)

Com esta *query* pretende-se explicitar as capacidades de *union* das bases de dados relacionais. Assim, apesar do seu elevado custo no desempenho, permite fazer de forma adequada a ligação entre as várias tabelas relevantes. Assim, evita iterar **mais que uma vez** em cada coleção, o que não é verdade em todos os sistemas de bases de dados.

MongoDB, por exemplo, com a implementação utilizada (analisada numa secção mais à frente) levaria a que os vários registos na coleção fossem iterados no mínimo duas vezes, uma para extrair o ID dos departamentos atuais e agrupar com esse ID os empregados atuais e outra para agrupar pelos IDs dos departamentos no histórico de empregos de cada empregado, enquanto que no SQL é utilizada apenas uma iteração (no máximo) por tabela para extrair os registos. Isto deve-se ao aninhamento dos objetos no **MongoDB.**

Obtiveram-se, assim, os seguintes resultados:

	DEPARTAMENTO	TOTAL
1	50	46
2	80	34
3	30	6
4	60	6
5	100	6
6	90	4
7	110	3
8	20	2
9	10	1
10	40	1
11	70	1

Figura 8 - Resultados da *query* 1 extra (SQL)

Query 2

Todos os empregados com quem o empregado com ID=101 já trabalhou:

```
with depAtual AS (SELECT department_id, hire_date from Employees where employee_id=101),
    depsAntigos AS (SELECT department_id, start_date, end_date from Job_history where employee_id=101)
select distinct(e.employee_id), concat(concat(e.first_name,' '), e.last_name) as Nome
from depAtual dAt, depsAntigos dAnt, Employees e, Job_history jh
where e.employee_id != 101 and e.department_id = dAt.department_id
or (jh.department_id = dAt.department_id and jh.end_date > dAt.hire_date and jh.employee_id = e.employee_id)
or (e.department_id = dAnt.department_id and e.hire_date < dAnt.end_date)
or (jh.employee_id = e.employee_id and jh.department_id = dAnt.department_id and (
    (jh.start_date > dAnt.start_date and jh.start_date < dAnt.end_date) or
    (jh.end_date > dAnt.start_date and jh.end_date < dAnt.end_date) or
    (jh.start_date < dAnt.start_date and jh.end_date > dAnt.end_date)))
order by employee_id;
```

Figura 9 - *Query* 2 Extra (SQL)

De forma semelhante à *query* extra anterior, esta tira partido da iteração singular por cada tabela relevante. No **OracleSQL**, é possível atribuir resultados de projeções em variáveis para consulta posterior enquanto que, utilizando o **MongoDB** como comparação novamente, não é possível iterar sobre a mesma coleção mais que uma vez numa única *query* (à exceção de valores ***$facet***, mas não há partilha de variáveis entre os vários estágios logo não é útil para casos destes).

Assim, enquanto que no **MongoDB** obrigaria a iterar sobre a mesma coleção no mínimo **4 vezes,** nomeadamente extrair o departamento onde o empregado com ID 101 trabalha; obter todos os empregados que trabalham com esse empregado atualmente; obter os departamentos onde o empregado com ID 101 já trabalhou e obter todos os empregados nesses departamentos que já trabalharam com ele, no **OracleSQL** cada tabela é acedida apenas **1** vez.

Foram obtidos os seguintes resultados:

	EMPLOYEE_ID	NOME
1	100	Steven King
2	102	Lex De Haan
3	200	Jennifer Whalen
4	205	Shelley Higgins
5	206	William Gietz

Figura 10 - Resultados da *query* 2 extra (SQL)

5. Base de Dados Documental - MongoDB

5.1. Migração dos Dados

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

De modo a atingir este objetivo, o grupo utilizou a ferramenta **Studio3T** que permite a exportação direta de **SQL**, linguagem nativa do SGBD **Oracle,** para objetos **JSON**, realizando a importação direta numa coleção arbitrária.

Os objetos **JSON**, de modo a tirar o máximo aproveitamento deste paradigma documental, foram estruturados de forma a manter em cada um deles toda a informação relativa a um departamento, de modo a serem inseridos em uma **única coleção** chamada *departments* que, de forma resumida, guarda a localização do departamento e mantém uma listagem dos empregados que trabalham nele.

Os objetos criados serão analisados com mais detalhe na secção seguinte.

5.2. Estruturação dos Dados

Tal como referido anteriormente, os dados originais foram primeiro transformados em objetos **JSON** antes de ser procedida à migração, pelo que foram criados utilizando a seguinte estrutura em **Studio3T**:

Field name	Source dataset	Source column
(root)	DEPARTMENTS + 3	
DEPARTMENT_ID	DEPARTMENTS	DEPARTMENT_ID
DEPARTMENT_NAME	DEPARTMENTS	DEPARTMENT_NAME
MANAGER_ID	DEPARTMENTS	MANAGER_ID
LOCATION		
EMPLOYEES	EMPLOYEES + 1	

Figura 11 - Nível 1 de aninhamento (JSON)

A estrutura acima diz respeito ao primeiro nível de aninhamento em **JSON**, que indica a informação base de cada departamento: **ID**, **NAME**, **MANAGER_ID**.

Para além deste primeiro nível, é necessário manter informação da localização do departamento que, como é uma relação **1 para 1**, vários departamentos terão o mesmo valor repetido para este campo. Assim, é necessário criar um objeto aninhado que mantém toda esses dados relativos à localização.

Na imagem seguinte encontra-se a representação dos campos do objeto aninhado **LOCATION:**

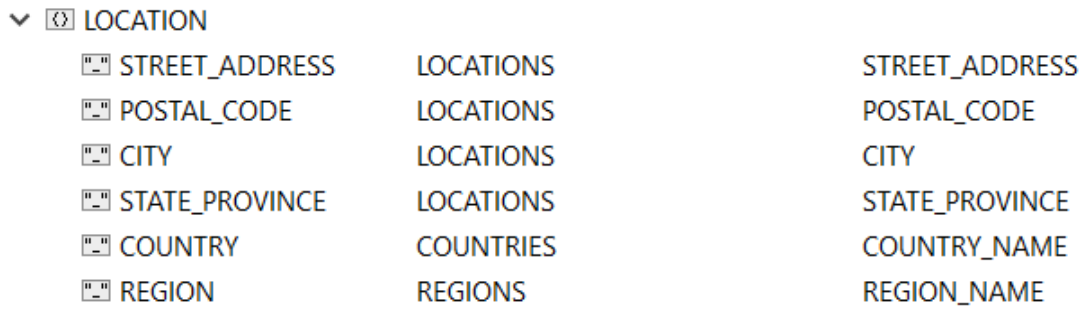

Figura 12 - LOCATION no nível 2 de aninhamento (JSON)

As tabelas originais **COUNTRY** e **REGION** foram adaptadas a atributos singulares (cujos nomes são os nomes das tabelas) visto que os seus **IDs** em **Oracle** tinham como único propósito identificar cada registo na tabela e, em cada um desses registos, era mantido apenas o nome do país, no caso do **COUNTRY**, ou o nome da região, no caso da **REGION**. Assim, em vez de introduzir complexidade extra ao **JSON** introduzindo mais níveis de aninhamento desnecessários, foi diretamente colocado no atributo a informação pertinente.

Para além da localização, cada um dos departamentos têm vários empregados, que dá origem a uma relação **1 para N**. Assim, é necessário manter para cada departamento uma lista (*array)* de empregados – **EMPLOYEES** – que lá trabalham, tal como visto na primeira figura apresentada.

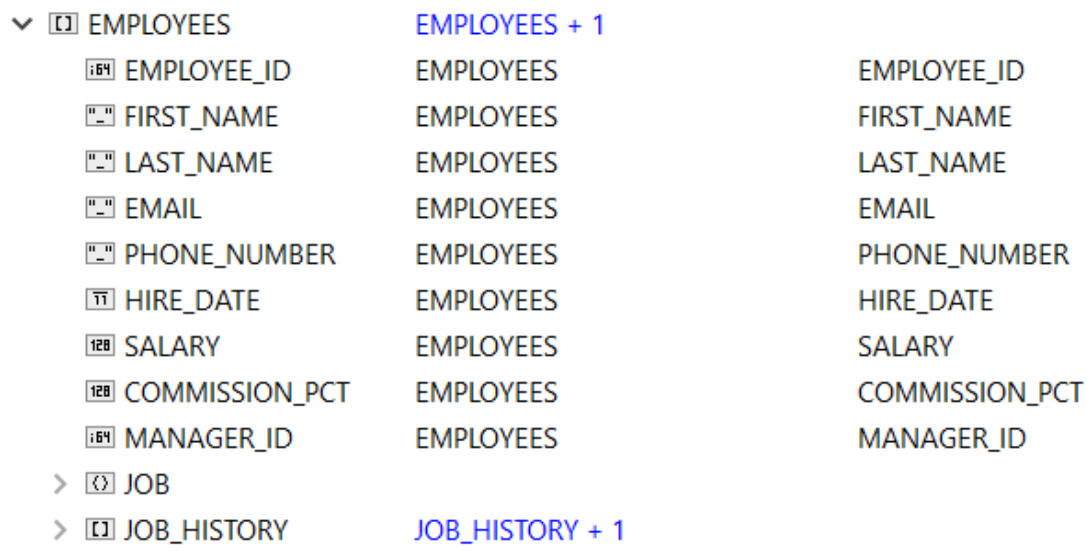

Figura 13 - EMPLOYEES no nível 2 de aninhamento (JSON)

Os empregados para além de toda a informação base como o seu primeiro nome, último nome, salário, etc, têm também um emprego – **JOB** – no seu departamento, que como é uma relação **1 para 1**, gera apenas um atributo com a informação respetiva desse emprego atual. Foi removido o campo de **ID** de departamento dentro do **JOB** pois como é um objeto aninhado no próprio departamento, seria um campo totalmente desnecessário. Para além dessa mudança, o campo **ID** de **JOB** foi também eliminado

por um motivo semelhante ao **COUNTRY** e **REGION** anteriores pois uma vez que apenas servem para identificar cada trabalho na tabela, não é um campo relevante para o objeto, pelo que se obtém a seguinte estrutura:

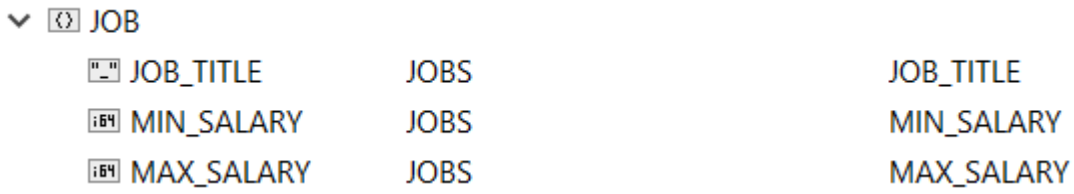

Figura 14 - JOB no nível 3 de aninhamento (JSON)

Para além do trabalho atual do empregado é necessário manter o histórico de trabalhos que ele já realizou nos vários departamentos, que origina o campo **JOB_HISTORY**. Como é uma relação **1 para N**, é necessário manter esta informação utilizando uma lista de empregos semelhante ao campo **JOB** já analisado. No entanto, como o empregado pode ter trabalhado num departamento diferente no passado é necessário manter o seu identificador. O objeto gerado tem, então, a seguinte estrutura:

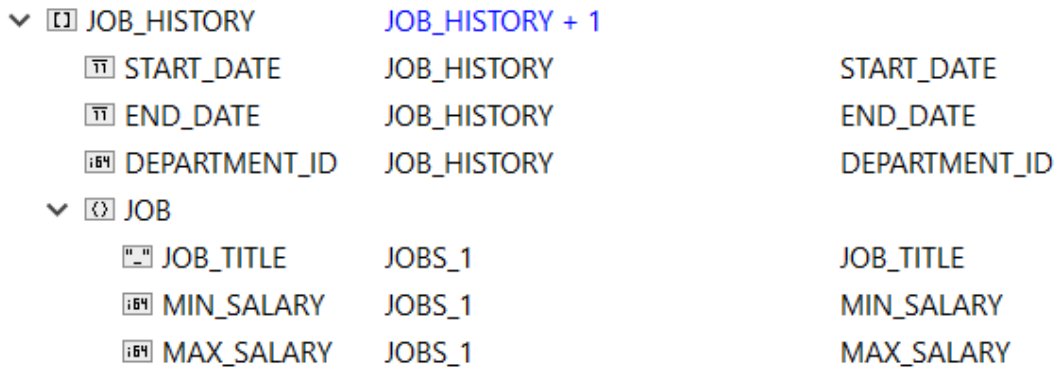

Figura 15 - JOB_HISTORY no nível 3 de aninhamento (JSON)

Como discutido anteriormente, é mantido o ID de departamento anterior para cada entrada no *array* assim como a informação do trabalho em si, após a remoção do seu ID pelas mesmas razões.

Assim, o objeto final tem no total **4** níveis de aninhamento.

Nos anexos está presente um exemplo de um objeto concreto após a sua conversão em JSON.

5.3. Queries

Após a importação dos dados em formato **JSON** para o **MongoDB**, o grupo procedeu à realização das três *queries* iniciais neste novo paradigma. Comparativamente ao **Oracle**, as *queries* ficam muito mais simples uma vez que não é necessário fazer *joins* de tabelas, toda a informação relevante está num único documento. Para além destas queries iniciais, foram também realizadas *queries* extra de modo a explorar ainda mais estas vantagens da implementação utilizada.

```
db.departments.find(
  {"EMPLOYEES": {$elemMatch: {"FIRST_NAME": "Shelley", "LAST_NAME": "Higgins"}}},
  {_id: 0, "DEPARTMENT_NAME": 1, "REGION": "$LOCATION.REGION"}
)
```

Figura 16 - *Query* exemplo com "find"

A título de exemplo inicial, a *query* presente na figura acima para encontrar onde trabalha o empregado “Shelley Higgins” (o nome do departamento e a região que o mesmo se situa), fica com uma sintaxe muito simples, quando a mesma operação seria bastante complexa em **Oracle**, uma vez que implicaria utilizar *joins* entre **EMPLOYEES, DEPARTMENTS, LOCATIONS, COUNTRIES** e **REGIONS** e projeções condicionais sobre esses *joins*.

5.3.1 Queries Transversais

Query 1

Média de salários por departamentos:

```
db.departments.aggregate([
  {$unwind: "$EMPLOYEES"},
  {$group: {_id: "$DEPARTMENT_NAME", Media_Salario: {$avg: "$EMPLOYEES.SALARY"}}},
  {$sort: {"Media_Salario": -1}},
  {$project: {_id: 0, Departamento: "$_id", Media_Salario: {$round: ["$Media_Salario",2]}}}
]).pretty()
```

Figura 17 - *Query* 1 Transversal (MongoDB)

Query 2

Top 5 empregados com o contrato ativo mais duradouro em cada região:

```
db.departments.aggregate([
  {$match: {"LOCATION.REGION": {"$exists": true, "$ne": null}} },
  {$unwind: "$EMPLOYEES"},
  {$addFields: {Nome: {$concat: ["$EMPLOYEES.FIRST_NAME", " ", "$EMPLOYEES.LAST_NAME"]}} },
  {$sort: {"EMPLOYEES.HIRE_DATE": 1,"Nome": 1} },
  {$group: { _id: "$LOCATION.REGION", Top5: {$push: {Nome:"$Nome", Data_Inicio:"$EMPLOYEES.HIRE_DATE"}}} },
  {$project: {_id: 0, Regiao: "$_id", Top5:{$slice:["$Top5", 5]}} }
]).pretty()
```

Figura 18 - *Query* 2 Transversal (MongoDB)

Query 3

Emprego antigo em que o empregado com ID 176 faturou mais:

```
db.departments.aggregate([
  {$unwind: "$EMPLOYEES"},
  {$match: {"EMPLOYEES.EMPLOYEE_ID": 176} },
  {$unwind: "$EMPLOYEES.JOB_HISTORY"},
  {$project: {_id: 0, Emprego:"$EMPLOYEES.JOB_HISTORY.JOB.JOB_TITLE", DataInicial: "$EMPLOYEES.JOB_HISTORY.START_DATE", DataFinal: "$EMPLOYEES.JOB_HISTORY.END_DATE",Faturado:
  {$sort: {"Faturado": -1}},
  {$limit: 1}
]).pretty()
```

Figura 19 - *Query* 3 Transversal (MongoDB)

5.3.2 Queries Extra

Query 1

Número total de empregados por localização:

```
db.departments.aggregate([
  {$unwind: "$EMPLOYEES"},
  {$match: {"LOCATION": {"$ne": {}}} },
  {$group: { _id: "$LOCATION", Total: {$sum: 1}}},
  {$sort: {"Total": -1}},
  {$project: {_id: 0, Localizacao: "$_id", Total: "$Total"}}
]).pretty()
```

Figura 20 - *Query* 1 Extra (MongoDB)

Com esta *query,* o objetivo principal foi mostrar o quão mais simples é utilizar informação que no **Oracle** estaria espalhada em imensas tabelas. Assim, em vez de dar *joins* por quase todas as tabelas do modelo relacional, é simplesmente gerado uma lista com informação de todos os empregados (“*unwind*”) e é utilizado uma função de agregação de imediato nesse conjunto, cortando muito a complexidade da operação e, consequentemente, o tempo de execução.

Obteve-se o seguinte resultado (de notar que o resultado era bastante mais extenso, portanto a imagem seguinte é um excerto do mesmo):

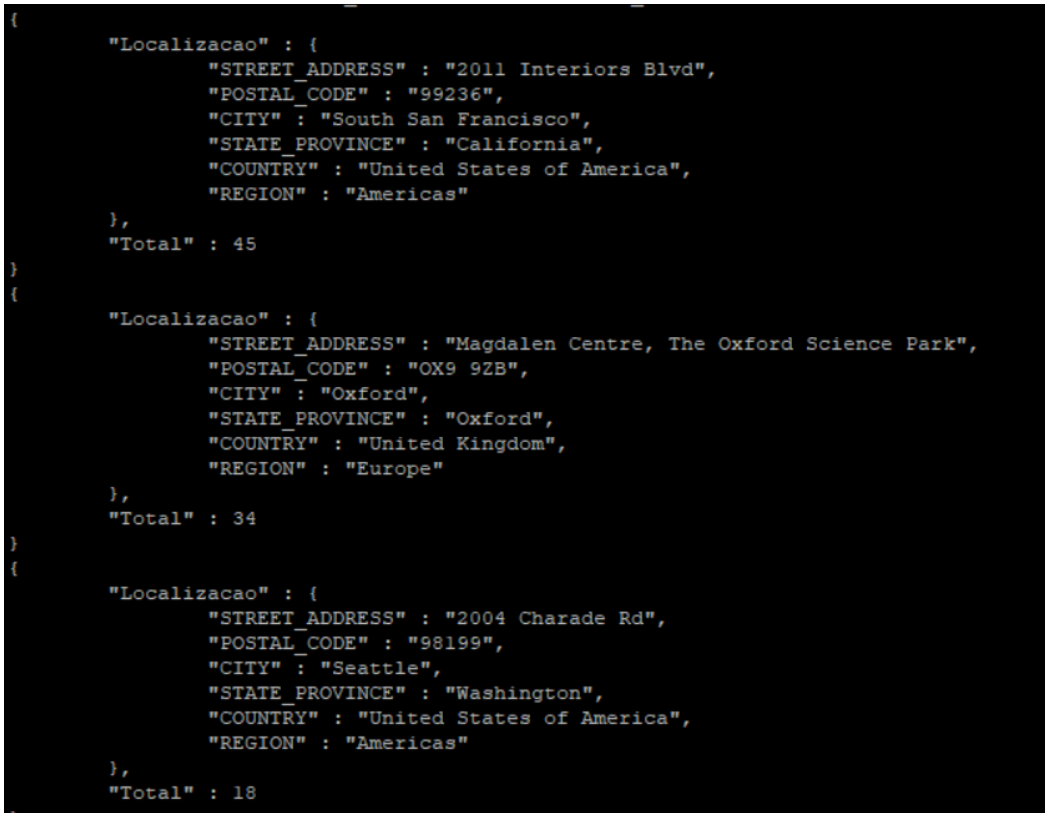

Figura 21 - Resultado parcial da *query* extra 1 (MongoDB)

Query 2

Maior salário de empregado por região:

```
db.departments.aggregate([
  {$unwind: "$EMPLOYEES"},
  {$match: {"LOCATION.REGION": {"$exists": true, "$ne": null}} },
  {$group: { _id: "$LOCATION.REGION", Maximo: {$max: "$EMPLOYEES.SALARY"}}},
  {$sort: {"Maximo": -1}},
  {$project: {_id: 0, Regiao: "$_id", Maximo: "$Maximo"}}
]).pretty()
```

Figura 22 - *Query* 2 Extra (MongoDB)

Semelhante à *query* extra anterior, o objetivo principal foi utilizar informação que no **Oracle** estaria acessível após vários *joins*. Neste caso em concreto, caso utilizássemos o modelo relacional anterior, teríamos de dar *join* entre **EMPLOYEES, DEPARTMENTS, LOCATIONS, COUNTRIES** e, por fim, **REGIONS** para obter as regiões em que cada empregado trabalha**.** No **MongoDB**, essa informação está

diretamente acessível num dos atributos do departamento, que é o objeto base, logo fica também muito mais simples de gerir estes dados e, consequentemente, é também mais rápida a sua execução.

A figura seguinte mostra o resultado obtido:

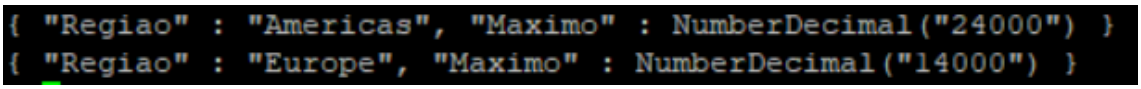

Figura 23 - Resultado da *query* extra 2 (MongoDB)

6. Base de Dados Orientada a Grafos – Neo4j

6.1. Modelo de Dados

De forma a migrar a base de dados relacional para uma base de dados orientada a grafos, foi necessário adaptar o esquema lógico da informação representada ao paradigma não relacional e às características do Neo4j, de maneira a tirar proveito das suas vantagens.

As principais alterações realizadas foram as seguintes:

- a tabela **JOB_HISTORY** era usada apenas para ligar cada empregado aos seus empregos/departamentos antigos, pelo que foi substituída por dois relacionamentos novos **WORKED_IN** (EMPLOYEE -> DEPARTMENT) e **PRACTICED** (EMPLOYEE -> JOB), que possuem como atributos as datas entre as quais o empregado exerceu dito cargo;
- foram removidas as chaves estrangeiras das tabelas – cada linha das tabelas passa a ser representada por um nodo e ligam-se entre si através de relacionamentos estabelecidos entre as suas chaves primárias, com um nome mais concreto para indicar a natureza do relacionamento como, por exemplo, **SUPERVISED_BY**. Desta forma, torna-se desnecessário guardar a informação das chaves estrangeiras nos nodos.

De resto, a conversão do modelo relacional para Neo4j foi direta. No fim, ficamos com o seguinte modelo lógico:

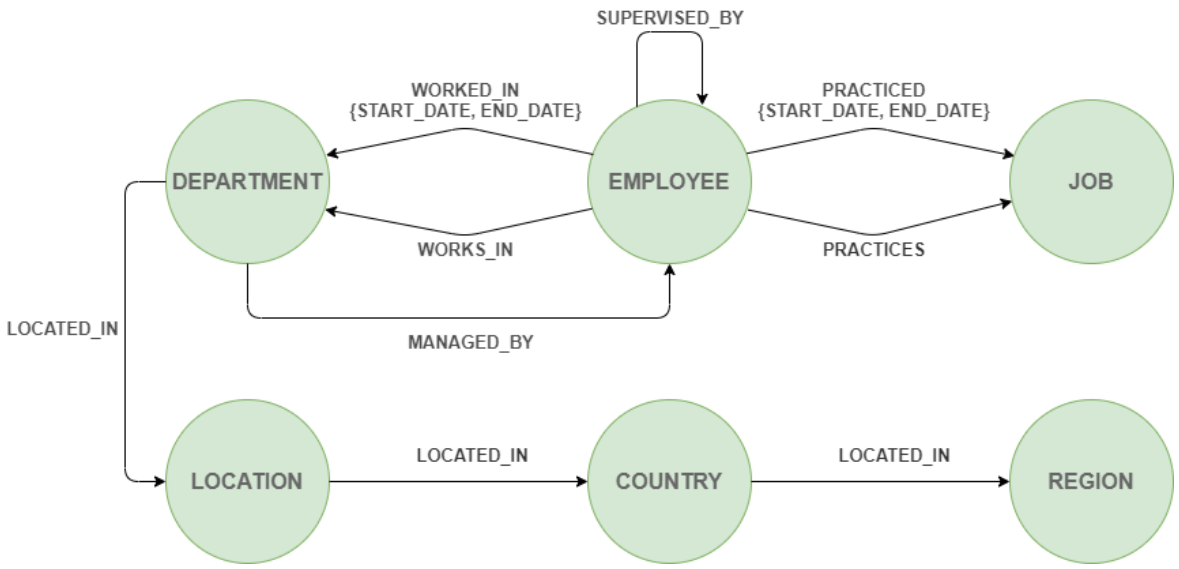

Figura 24 - Modelo lógico para Neo4j

6.2. Migração de Dados

Para realizar a migração, o grupo começou por exportar os dados da base de dados relacional para ficheiros **CSV**, através da funcionalidade de exportação do **SQLDeveloper**. Durante este passo, foram criados sete ficheiros diferentes, cada um com a informação de uma dada tabela do modelo relacional.

Uma precaução importante que se teve de tomar consistiu em mudar a forma de representação de números decimais em **SQLDeveloper**, passando de usar vírgulas para pontos, de forma a garantir que a informação ficava legível no ficheiro CSV. Caso contrário, a vírgula do número decimal seria interpretada como um delimitador do ficheiro pelo Neo4j, que acabaria por considerar o decimal como dois atributos diferentes em vez de um só, alterando a informação carregada para a base de dados não relacional.

Além disso, foi também mudada a forma de representação de datas para usar 4 algarismos para o ano, de forma a possibilitar a conversão para data em Neo4j, na query 3.

De seguida, procedeu-se à criação dos nodos na base de dados, a partir das tabelas. Para este processo, o grupo usou programas escritos em **Cypher** (língua nativa do Neo4j), para importar os dados dos ficheiros CSV e transformar cada linha das tabelas num nodo com as respetivas propriedades. O exemplo abaixo corresponde ao código usado para gerar os nodos **EMPLOYEE**:

```
LOAD CSV WITH HEADERS FROM "file:///EMPLOYEES.csv" AS row
CREATE (employee:EMPLOYEE {EMPLOYEE_ID: row.EMPLOYEE_ID})
SET employee.FIRST_NAME = row.FIRST_NAME,
    employee.LAST_NAME = row.LAST_NAME,
    employee.EMAIL = row.EMAIL,
    employee.PHONE_NUMBER = row.PHONE_NUMBER,
    employee.HIRE_DATE = row.HIRE_DATE,
    employee.SALARY = row.SALARY,
    employee.COMMISSION_PCT = row.COMMISSION_PCT,
    employee.MANAGER_ID = row.MANAGER_ID
RETURN employee;
```

Figura 25 - Código utilizado para gerar os nodos EMPLOYEE

Por fim, restou apenas criar os relacionamentos entre os nodos. A lógica usada foi análoga à dos nodos, carregando os dados dos ficheiros CSV, fazendo corresponder a chave primária e estrangeira aos nodos em questão (através da diretiva **MATCH**) e estabelecendo o relacionamento entre eles. Abaixo é possível observar um exemplo, neste caso do relacionamento **WORKS_IN**, de empregado para departamento:

```
LOAD CSV WITH HEADERS FROM 'file:///EMPLOYEES.csv' AS row
MATCH (employee:EMPLOYEE {EMPLOYEE_ID: row.EMPLOYEE_ID})
MATCH (department:DEPARTMENT {DEPARTMENT_ID: row.DEPARTMENT_ID})
CREATE (employee)-[:WORKS_IN]→(department)
RETURN employee,department
```

Figura 26 - Relacionamento WORKS_IN

No caso especial da tabela **JOB_HISTORY**, que foi substituída por dois relacionamentos, foi necessário atribuir propriedades aos relacionamentos com a informação presente na tabela, ou seja, as datas de início e fim da exerção do emprego em questão, como por exemplo:

```
LOAD CSV WITH HEADERS FROM 'file:///JOB_HISTORY.csv' AS row
MATCH (employee:EMPLOYEE {EMPLOYEE_ID: row.EMPLOYEE_ID})
MATCH (job:JOB {JOB_ID: row.JOB_ID})
CREATE (employee)-[:PRACTICED {START_DATE: row.START_DATE, END_DATE: row.END_DATE}]→(job)
RETURN employee,job
```

Figura 27 - JOB_HISTORY em Neo4j

6.3. Queries

Uma vez criada e populada a base de dados, faltava apenas a implementação das *queries*. Em primeiro lugar, adaptaram-se as 3 *queries* transversais criadas pelo grupo à sintaxe de **Cypher**, de maneira a torná-las compatíveis com o Neo4j. O propósito destas *queries* era compará-las nas 3 bases de dados implementadas, de maneira a observar o impacto dos vários paradigmas no seu desempenho e entender que paradigma era mais favorável a cada uma e porquê.

Depois, foram desenvolvidas mais duas *queries* extra especialmente para Neo4j, de maneira a focar os seus pontos fortes em relação aos outros SGBD e explorar as suas potencialidades.

6.3.1 Queries transversais

Query 1

Média de salários por departamentos:

```
MATCH (e:EMPLOYEE)-[w:WORKS_IN]→(d:DEPARTMENT)
RETURN d.DEPARTMENT_NAME AS Departmento,
    round(avg(toInteger(e.SALARY)),2) AS Média
ORDER BY Média DESC
```

Figura 28 - *Query* 1 Transversal (Neo4j)

Query 2

Top 5 empregados com o contrato ativo mais duradouro em cada região:

```
MATCH (e:EMPLOYEE)-[:WORKS_IN]→(d:DEPARTMENT)-[:LOCATED_IN]→(l:LOCATION)
MATCH (l:LOCATION)-[:LOCATED_IN]→(c:COUNTRY)-[:LOCATED_IN]→(r:REGION)
WITH e.FIRST_NAME+' '+e.LAST_NAME AS Nome, e.HIRE_DATE AS Datas, r
ORDER BY e.HIRE_DATE, Nome
RETURN r.REGION_NAME AS Região,
    collect(Nome)[..5] As Nomes,
    collect(Datas)[..5] AS Datas_Início
```

Figura 29 - *Query* 2 Transversal (Neo4j)

Query 3

Emprego antigo em que o empregado com id 176 faturou mais:

```
MATCH (e:EMPLOYEE)-[r:PRACTICED]→(j:JOB)
WHERE e.EMPLOYEE_ID = '176'
RETURN j.JOB_TITLE as Emprego,
    r.START_DATE as Data_Inicial,
    r.END_DATE as Data_Final,
    duration.inMonths(date(replace(r.START_DATE,".","-")),
    date(REPLACE(r.END_DATE,".","-"))).months*(toInteger(j.MAX_SALARY)) AS Faturado
ORDER BY Faturado DESC
LIMIT 1
```

Figura 30 - *Query* 3 Transversal (Neo4j)

6.3.2 Queries extra

Query 1

Empregados no 4º nível da cadeia de supervisão:

```
MATCH (e:EMPLOYEE)-[:SUPERVISED_BY]→(:EMPLOYEE)
    -[:SUPERVISED_BY]→(:EMPLOYEE)-[:SUPERVISED_BY]→(:EMPLOYEE)
RETURN e.FIRST_NAME+' '+e.LAST_NAME AS Nome
ORDER BY Nome
```

Figura 31 - *Query* 1 Extra (Neo4j)

Nesta *query*, tira-se proveito da sintaxe de Cypher para encadear vários relacionamentos **SUPERVISED_BY** de forma a descobrir os empregados que se pretende. Esta sintaxe proporciona uma forma muito simples e intuitiva de usar recursividade em pesquisas, que seria muito mais complicada de implementada numa base de dados relacionais, por exemplo. Além disso, é também muito fácil de comprovar os resultados da *query* em Neo4j graças à representação em grafo da informação:

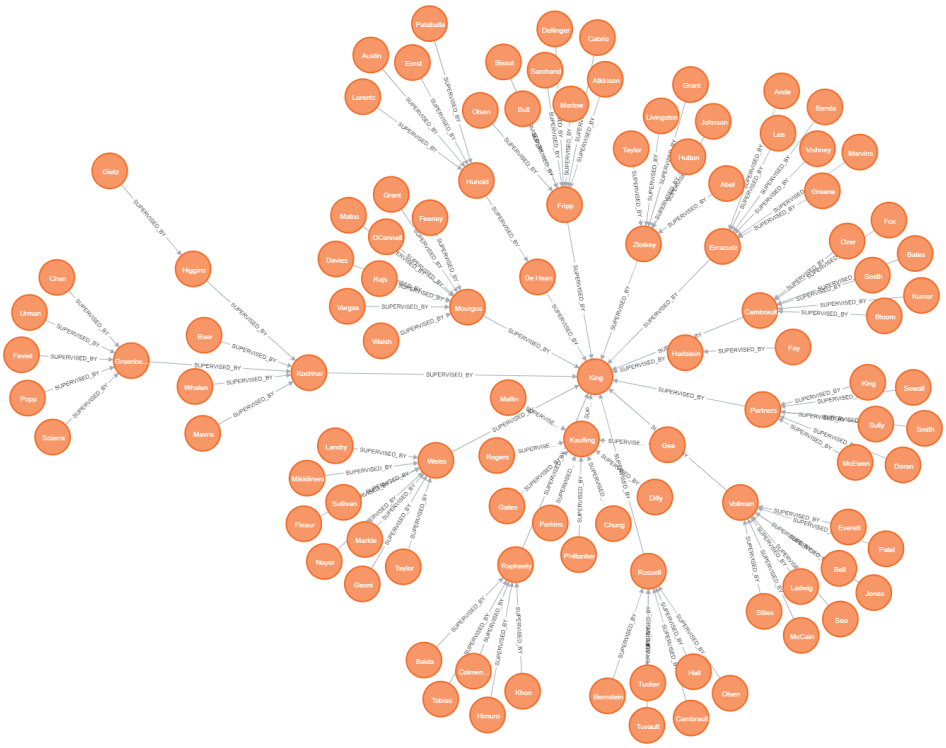

Figura 32 - Resultado em grafo da *query* extra 1 (Neo4j)

Query 2

Criar relacionamentos RESIDES_IN de empregado para país, para todos os empregados que estão no 3º nível da cadeira de supervisão e têm o emprego de 'Shipping Clerks' ou 'Sales Representatives':

```
MATCH (e:EMPLOYEE)-[:WORKS_IN]→(:DEPARTMENT)-[:LOCATED_IN]→(:LOCATION)-[:LOCATED_IN]→(c:COUNTRY)
MATCH (j:JOB)←[:PRACTICES]-(e:EMPLOYEE)-[:SUPERVISED_BY]→(:EMPLOYEE)-[:SUPERVISED_BY]→(:EMPLOYEE)
WHERE j.JOB_ID = 'SH_CLERK' OR j.JOB_ID = 'SA_REP'
MERGE (e)-[:RESIDES_IN]→(c)
RETURN e,c
```

Figura 33 - *Query* 2 Extra (Neo4j)

Nesta *query*, procurou-se tirar proveito da capacidade de criar relacionamentos durante a realização de *queries* no Neo4j e da sua representação visual para criar relacionamentos entre países e os empregados que cumprissem as condições estabelecidas e observar a nova informação de seguida:

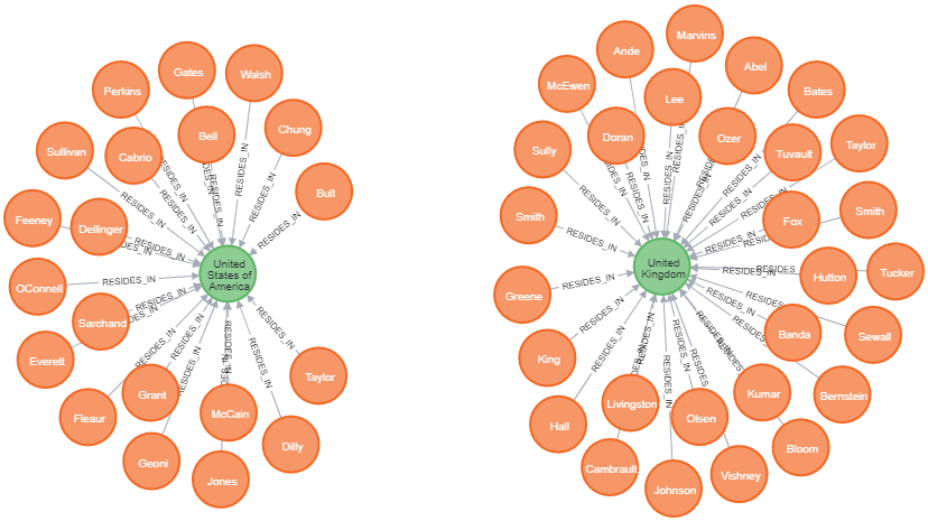

Figura 34 - Resultado em grafo da *query* extra 2 (Neo4j)

7. Comparação do desempenho das queries transversais

O grupo desenvolveu 3 *queries* de base que seriam implementadas em todas as bases de dados usadas, de maneira a poder comparar a sua *performance* no fim e observar o efeito das características de cada BD sobre o desempenho das *queries*, debatendo a adequação dos sistemas de gestão de bases de dados escolhidos para as consultas em questão.

Uma vez implementadas estas *queries* transversais, o grupo cronometrou a sua execução para comparar o desempenho das mesmas nos 3 SGBD diferentes.

No que toca ao **Oracle** e ao **Neo4j**, estes sistemas já cronometram a execução das *queries* automaticamente e apresentam os tempos em conjunto com os resultados. No caso do **MongoDB**, o grupo elaborou funções em **Javascript** que executam as *queries* e as cronometram, como o seguinte exemplo:

```
time(() => db.departments.aggregate([
  {$unwind: "$EMPLOYEES"},
  {$group: {_id: "$DEPARTMENT_NAME", Media_Salario: {$avg: "$EMPLOYEES.SALARY"}}},
  {$sort: {"Media_Salario": -1}},
  {$project: {_id: 0, Departamento: "$_id", Media_Salario: {$round: ["$Media_Salario",2]}}}
]))
```

Figura 35 - Funções em Javascript para cronometrar

Os resultados obtidos foram os seguintes:

	Oracle	MongoDB	Neo4j
Query 1	35 ms	4 ms	57 ms
Query 2	76 ms	4 ms	13 ms
Query 3	12 ms	14 ms	11 ms

Tabela 1 – Resultados obtidos das 3 *queries* transversais

7.1. Query 1

	Oracle	MongoDB	Neo4j
Query 1	35 ms	4 ms	57 ms

Tabela 2 - Resultados obtidos na 1ª *query* transversal

Média de salários por departamentos:

A *query* 1 consistia em determinar a média de salários por departamentos.

No caso da base de dados relacional, esta *query* usava informação de apenas duas tabelas, **EMPLOYEES** e **DEPARTMENTS**, que estão diretamente ligadas, pelo que não era necessário fazer muitas combinações de chaves, o que é benificial para o **Oracle**. Contudo, as operações de agregação envolvidas

acabaram por piorar um bocado a performance da *query*, visto que não são o ponto forte de bases de dados relacionais. Assim, o Oracle conseguiu executar a *query* num tempo decente, intermediário em relação aos tempos medidos nas outras bases de dados usadas.

Já no caso do **MongoDB**, os dados usados pela *query* encontram-se ainda mais centralizados, uma vez que a informação foi organizada em função dos departamentos neste modelo – isto é, cada documento da base de dados diz respeito a um departamento diferente. Este facto, em conjunto com a maior operacionalidade inata de bases de dados não relacionais para funções de agregação, permitiu obter um tempo de execução bastante baixo para esta *query*.

Por fim, o **Neo4j** teve o pior desempenho uma vez que é um sistema de bases de dados orientado a relacionamentos, ou seja, com foco em **pesquisas relacionais** – é possível tirar proveito máximo do Neo4j quando se executam *queries* que necessitam de percorrer muitas arestas. Além disso, há ainda o tempo extra de criação da parte visual (nodos, setas, etc). No caso desta *query*, o foco eram os nodos dos empregados e dos departamentos, pelo que a *query* acabou por ir contra a maior vantagem do SGBD e, naturalmente, teve um pior desempenho.

7.2. Query 2

	Oracle	MongoDB	Neo4j
Query 2	76 ms	4 ms	13 ms

Tabela 3 - Resultados obtidos na 2ª *query* transversal

A *query* 2 consistia em determinar o top 5 de empregados com o contrato ativo mais duradouro em cada região. Como seria de esperar, esta *query* foi bastante mais rápida nas bases de dados não relacionais do que na base de dados relacional.

Em **Oracle**, a informação é guardada em tabelas, que se relacionam através de chaves estrangeiras e outras tabelas. Esta *query* implicava a realização de muitas combinações de chaves para Oracle, uma vez que há bastantes tabelas intermediárias entre a **EMPLOYEES** e a **REGION**, o que é bastante trabalhoso e acabava por prejudicar a performance da *query* nesta base de dados.

No caso do **MongoDB**, os dados foram organizados em função dos departamentos, sendo que cada departamento passou a conter no seu documento a informação da sua localização e a lista dos seus empregados. Além disso, as tabelas **COUNTRY** e **REGION** foram apagadas e a **LOCATION** passou a guardar o nome dos respetivos país e região, o que centralizou bastante a informação usada por esta *query* e facilitou o seu acesso. Desta forma, torna-se possível executar a *query* em bastante menos tempo, graças à nova organização dos dados.

No que toca ao **Neo4j**, é uma base de dados com foco nos relacionamentos entre os nodos, apresentando uma grande predisposição para **pesquisas relacionais** que necessitem de percorrer muitas destas arestas. A *query* presente é um desses casos, uma vez que há o mesmo número de nodos intermediários entre os nodos de empregados e regiões que havia de tabelas no modelo relacional –

contudo, o Neo4j colmata as falhas do Oracle e está otimizado para percorrer esses relacionamentos eficientemente, resultando num tempo de execução muito inferior. Verifica-se, novamente, que há um tempo extra que custa a criar a parte visual.

7.3. Query 3

	Oracle	MongoDB	Neo4j
Query 3	12 ms	4 ms	11 ms

Tabela 4 - Resultados obtidos na 3ª *query* transversal

A *query* 3 consistia em determinar o emprego antigo do utilizador com id 176 no qual ele ganhou mais dinheiro, através do salário. O grupo escolheu o empregado 176 após a análise minuciosa dos dados, pois era um de apenas dois empregados em toda a base de dados que tinham 2 empregos no histórico.

Novamente, o **MongoDB** voltou a apresentar o melhor desempenho, uma vez que esta *query* acedia apenas a um único documento, e dentro dele a um único empregado, pelo que o SGBD foi bastante eficiente.

No que toca ao **Oracle** e ao **Neo4j**, apresentaram desempenhos bastante semelhantes. A performance em Oracle é degradada pelas várias combinações de chaves que é necessário realizar entre tabelas, enquanto que em Neo4j o motivo é o contrário – não é necessário percorrer quase nenhuma aresta (relembre-se que o grupo eliminou a tabela **JOB_HISTORY** na migração para Neo4j e a substituiu por relacionamentos), sendo a maior parte do tempo de execução usada em cálculos aritméticos e conversões de dados. A performance do Neo4j não foi pior graças ao facto de a *query* aceder apenas a dois nodos – se necessitasse de aceder a vários pares de nodos isolados entre si, seria bastante mais lenta. Tal como nos casos anteriores, há também o tempo extra que demoram a ser criadas as relações entre os nodos visualmente.

7.4. Conclusões

Os testes realizados vieram a comprovar o que foi ensinado nas aulas sobre as várias bases de dados usadas. O **MongoDB** provou ser a melhor BD para todas as *queries* transversais desenvolvidas, devido sobretudo à nova organização dos dados e à centralização da informação, que também facilitava operações de agregação. O **Neo4j** mostrou uma grande operacionalidade para **pesquisas relacionais**, o que se traduziu num melhor desempenho para *queries* que necessitassem de percorrer um grande número de arestas, especialmente a *query* 2. O **Oracle**, como seria de esperar, exibiu o pior desempenho de entre todas as bases de dados usadas, uma vez que não apresenta vantagens notáveis em relação aos SGBD's não relacionais usados, no contexto das *queries* elaboradas. Contudo, apresentou um desempenho decente na *query* 3, devido ao diminuto volume de dados envolvidos na sua realização.

8. Conclusões e Trabalho Futuro

O desenvolvimento deste projeto permitiu a consolidação dos objetivos de estudo abordados nas aulas da unidade curricular, nomeadamente os diferentes tipos e vantagens de bases de dados NOSQL.

Também foram utilizadas novas ferramentas, como Studio 3T, para ajudar na construção dos ficheiros JSON, e aprimorou-se o uso de ferramentas previamente utilizadas, como o SQLDeveloper, útil para estabelecer conexões a base de dados e escrever *queries*.

Concluindo, foi possível obter conhecimentos valiosos através da elaboração deste trabalho prático, uma vez que foram utilizadas ferramentas novas e úteis em futuros trabalhos e adquirido muita experiência relativamente a diferentes tipos de bases de dados.

Lista de Siglas e Acrónimos

BD Base de Dados

SGBD Sistema de Gestão de Bases de Dados

HR *Human Resources* (Recursos Humanos)

Anexos

I. Anexo 1


```
{
    "DEPARTMENT_ID": 20,
    "DEPARTMENT_NAME": "Marketing",
    "MANAGER_ID": 201,
    "LOCATION": {
      "STREET_ADDRESS": "147 Spadina Ave",
      "POSTAL_CODE": "M5V 2L7",
      "CITY": "Toronto",
      "STATE_PROVINCE": "Ontario",
      "COUNTRY": "Canada",
      "REGION": "Americas"
    },
    "EMPLOYEES": [
      {
        "EMPLOYEE_ID": 201,
        "FIRST_NAME": "Michael",
        "LAST_NAME": "Hartstein",
        "EMAIL": "MHARTSTE",
        "PHONE_NUMBER": "515.123.5555",
        "HIRE_DATE": {"$date": "2004-02-17T00:00:00Z"},
        "SALARY": {"$numberDecimal": "13000"},
        "COMMISSION_PCT": null,
        "MANAGER_ID": 100,
        "JOB": {
          "JOB_TITLE": "Marketing Manager",
          "MIN_SALARY": 9000,
          "MAX_SALARY": 15000
        },
        "JOB_HISTORY": [
          {
            "START_DATE": {"$date": "2004-02-17T00:00:00Z"},
            "END_DATE": {"$date": "2007-12-19T00:00:00Z"},
            "DEPARTMENT_ID": 20,
            "JOB": {
              "JOB_TITLE": "Marketing Representative",
              "MIN_SALARY": 4000,
              "MAX_SALARY": 9000
            }
          }
        ]
      },
      (...)
    ]
}
```