Migração de uma Base de Dados Relacional para três diferentes tipos de Base de Dados

A82200 - Henrique Faria and A81139 - Paulo Bento

Universidade do Minho, Mestrado Integrado em Engenharia Informática

Resumo A Base de Dados que se pretende migrar é a Sakila do MySQL. Esta migrar-se-á para outras três base de dados. O processo de migração será explicitado e será feita uma breve sumula das vantagens de cada uma delas. As bases de dados a usar no trabalho são: uma base de dados documental (MongoDB) , uma base de dados grafica (Neo4j) e para uma Relacional (OracleDB).

Palavras-chave: Python · MySQL · Oracle · Mongo · Neo4J

1 Introdução

A migração entre bases de dados é uma tarefa relativamente típica quando se pretende obter maior eficiência em certas consultas ou melhorar o desempenho de forma geral. As diferentes base de dados sejam estas SQL ou NoSQL têm o seu propósito. Normalmente os serviços maiores, tem uma base de dados relacional para "suportar"todo o serviço e outras NoSQL para "suportar"consultas mais especificas com maior rapidez.

O objetivo do trabalho passou por realizar então uma migração de uma base de dados MySQL para OracleDB, uma parte desta para MongoDB e para Neo4j.

As seções sobre cada base de dados seguem o mesmo princípio, isto é, a motivação. Note-se que algumas modificações feitas(Oracle - Contém uma seção sobre as ficuldades com a migração).

Primeiramente, discute-se sobre a base de dados relacional, seguindo com a documental e, finalmente, a base de dados gráfica.

Por fim terminamos o relatório com algumas conclusões sobre as base de dados desenvolvidas.

2 OracleDB

2.1 Criação da base de dados

2.1.1 Motivação

A criação da base de dados Oracle pretende responder á necessidade de escalar horizontalmente a nossa base de dados bem com garantir maior flexibilidade e custos reduzidos na manutenção.

2.1.2 Criação - modificações

A criação destas bases de dados passa por um script muito identico ao do MySQL.

Ao criar esta base de dados os tipos TINYINT e SMALLINT foram convertidos para NUMBER(10) visto que estes tipos não têm representação nos DataTypes Oracle. Para além disso o tipo BOOLEAN foi modificado visto que as bases de dados Oracle também não suportam este tipo de dados sugerindo optar por um VARCHAR2(Y,N) ou, como foi impementado, NUMBER(1,0).

O tipo DATETIME do MySQL foi transformado no tipo correspondente na base de dados Oracle DATE e o tipo VARCHAR foi convertido em VARCHAR2. Por fim o tipo DECIMAL(5,2) presente na base de dados foi convertido para o seu correspondente Oracle FLOAT(2).

2.2 Criação - Dificuldades na implementação

Ao implementar a mudança da base de dados em python surgiram alguns imprevistos.

Em primeiro lugar a ligação á base de dados foi difícil de conseguir visto que após instalada a base de dados tivemos de proceder á instalação do instant-client, isto suscitou problemas pois este está na versão 19.3.0.0.0 e os torturiais para a instalação deste estão desatualizados estando atualmente na versão 12. Isto levou-nos a procurar soluções em foruns, no git e no StackOverflow até que finalmente as variáveis com o caminho foram criadas e os ficheiros do Instant Client foram colocados na diretoria correta.

Posteriormente surgiram problemas ao correr os scripts para a criação das tabelas pois estes continham pontuação como ';' no fim dos comandos coisa que não pode ser usada nestes scripts no python, removendo os mesmos foi o suficiente para os scripts de criação de tabelas começarem a funcionar.

Outro problema que surgiu estava ligado á criação de duas tabelas que continham chaves estrangeiras uma da outra. Por estarmos a executar o script no python como uma das tabelas não tinha sido criada ao declarar uma chave estrangeira dessa tabela o comando não funcionava. Para resolver este problema, foram criadas primeiro as duas tabelas e a segunda contendo a chave estrangeira referenciando a primeira. Posteriormente foram ambas preenchidas, primeiro a tabela sem chaves estrangeiras seguindo-se a restante. Após serem preenchidas procedeu-se á alteração da primeira tabela criando a restrição de chave estrangeira correspondente.

Por fim surgiu outro problema ao guardar os dados. Na tabela Film os dados referentes a special_features são devolvidos como tuplos em vez de uma string. Para isso foi necessário executar um ciclo for sobre os dados retirados da tabela MySQL e transformar esse tuplos em strings para poderem ser guardados na nossa base de dados.

2.3 Preenchimento da Base de Dados

Para preencher a base de dados foram usados os seguintes comandos (para exemplificar vão ser listados os comandos usados na tabela language):

- **language_sql =** ***"SELECT * FROM language"***
  Este primeiro comando serve para criar a query que servirá para ler a tabela language.

- **mycursor.execute(language_sql)**
  Este comando executa no *MySQL* a leitura da tabela language.

- **languageRows = mycursor.fetchall()**
  Este comando vai buscar todos os registos lidos previamente com a query *language_sql*.

- **oracleCursor.bindarraysize = len(languageRows)**
  Este comando serve apenas para que o python saiba quantas vezes terá de executar a query que se segue.

- **oracleCursor.executemany('insert into LANGUAGE(LANGUAGE_ID, NAME,LAST_UPDATE) values(:1,:2,:3)', languageRows)**
  Esta query será executada sobre cada linha da talela languageRows.

- **con.commit()**
  Este comando permite finalizar os processos realizados adicionando as modificações realizadas por estes permanentemente á nossa base de dados Oracle.

Nota: Estes comandos foram usados em cada tabela copiada de MySQL para OracleDB.

2.4 Querys á Base de Dados

Em seguida listam-se três das querys feitas á base de dados Oracle e as respetivas respostas.

Listing 1.1. Query á base de dados Oracle para devolver todos os Filmes com id inferior ou igual a 3

```
oracleCursor.execute("SELECT * FROM FILM WHERE FILM_ID <= 3").fetchall()

oracleCursor.execute("SELECT FILM.TITLE, LANGUAGE.NAME FROM FILM,LANGUAGE
 WHERE FILM.LANGUAGE\_ID = LANGUAGE.LANGUAGE\_ID").fetchall()
```

Listing 1.2. Query á base de dados Oracle para devolver o total faturado por utilizador

```
oracleCursor.execute("SELECT CUSTOMER.FIRST_NAME,CUSTOMER.LAST_NAME,
SUM( AMOUNT ) FROM PAYMENT,CUSTOMER
WHERE PAYMENT.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID
GROUP BY CUSTOMER.FIRST_NAME,CUSTOMER.LAST_NAME ").fetchall()
```

3 MongoDB

3.1 Criação da base de dados

3.1.1 Motivação

Por esta ser uma base de dados orientada a documentos, tem várias vantagens face ao MySQL. Para além de ser escalável horizontalmente, é mais facil de manter, atualizar e é ótima para coletar informações referentes a um objeto, evitando os **joins** de tabelas usados pelo MySQL visto que este guarda toda a informação de um objeto num mesmo documento.

3.1.2 Criação - modificações

Para a criação desta base de dados supusemos ser uma aplicação de uma loja na qual só se pretende obter informações sobre que utilizador consome filmes em que loja, quem o atendeu, em que data e qual o preço. Para além disso é importante saber as carateristicas dos filmes presentes nas lojas, as linguas em que estão, quais os atores que participaram e as categorias em que se insere, isto é, é necessário um sistema de suporte de consulta online.

3.2 Criação - Estrutura da Base de Dados

Em seguida apresentam-se os documentos gerados com base nos requisitos para a base de dados MongoDB. A coleção dos Filmes(films) contem o identificador do Filme(id), o titulo(title), o ano que saiu(release_year), uma descrição do conteúdo do Filme(descrição), o Idioma original(original_language), o Idioma estrangeiro(foreign_language),um Array com as categorias as qual pertence(categorys) e , por último, uma lista de atores que participaram no filme(actors).

Listing 1.3. Esquema da coleção films

```
Filme:
{
        "id": ,
        "title": ,
        "release_year": ,
        "descricao": ,
        "original_language": ,
        "foreign_language": ,
        "categorys": ,
        "actors":
}
```

A segunda coleção para os Pagamentos(payments) contem um objeto por loja que é identificado por um identificador(store_id), contendo também o identificador do manager(manager_id), um Array onde esta organizado as faturas

por customer(paymentsByCostumer), cada elemento desse Array tem um identificador do cliente(customer_id), o nome do mesmo (name), o email (email) e um Array de todas as faturas desse Cliente.

Cada fatura é identificado pelo seu identificador próprio(payment_id), a quantia paga(amount), a data em qual foi efetuado(date), o identificador do staff que atendeu esse pagamento(staff_id), o nome do staff(name) e o email do mesmo(email).

Listing 1.4. Esquema da coleção payments

```
Payments
{
        "store_id" : ,
        "manager_id" : ,
        "paymentsByCostumer" :
        [
                {
                        "customer_id" : ,
                        "name" : ,
                        "email" : ,
                        "payments" :
                        [
                                {
                                        "staff_id" : ,
                                        "name" : ,
                                        "email" : ,
                                        "payment_id" : ,
                                        "amount" : ,
                                        "date" :
                                }
                        ]
                }
        ]
}
```

3.3 Preenchimento da Base de Dados

A query usada para retirar os dados necessários para povoar o primeiro esquema mencionado na seção anterior é a seguinte.

Listing 1.5. Query para povoar a primeira coleção

```
SELECT f.title,f.release_year,f.description,c.name AS Category,
a.first_name,a.last_name, language.name AS Foreign_language,extra.name
AS Original_language,
f.film_id
 FROM film AS f LEFT JOIN film_category AS fc ON fc.film_id = f.film_id
  LEFT JOIN category AS c ON c.category_id = fc.category_id
```

```
LEFT JOIN film_actor AS fa ON fa.film_id = f.film_id
  LEFT JOIN actor AS a ON a.actor_id = fa.actor_id
    LEFT JOIN language ON language.language_id = f.language_id
      LEFT JOIN ( SELECT f.film_id, l.name FROM film AS f
        LEFT JOIN language AS l ON l.language_id = f.original_language_id
          WHERE f.original_language_id is not null) extra ON f.film_id =
            extra.film_id
```

Em seguida podemos observar o loop usado para criar os documentos film.

```
while (True):
    try:
        i1 = next(myFilmIt)
    except StopIteration:
        print("first collection done")
        break
    if i1aux == None:
        i1aux = i1
    # category and actors
    if i1aux[0] == i1[0]:
        actors.append(i1[4] + " " + i1[5])
        if i1[3] not in categorys:
            categorys.append(i1[3])
    else:
        info = {"id": i1aux[8], "title": i1aux[0], "release_year": i1aux[1], "descrição": i1aux[2],
                "original_language": i1aux[7], "foreign_language": i1aux[6], "categorys": categorys, "actors": actors}
        filmsList.insert_one(info)
        actors.clear()
        categorys.clear()
    i1aux = i1
```

Figura 1. Ciclo que gera cada documento Film guardado na base de dados MongoDB

Cada objeto iterado neste ciclo while corresponde a um filme, podendo haver filmes repetidos em iterações sucessivas cuja única diferença reside sempre no ator e pode ter também diferentes categorias, tendo portanto, cada iteração, um ator diferente e uma ou nenhuma categoria. Assim fazendo uso do ciclo com a verificação do id de cada filme sendo igual permite que identifiquemos o mesmo filme em várias iterações e possamos adicionar os atores todos numa string, bem como as categorias. Quando a verificação da igualdade do filme falha sabemos que podemos adicionar o filme á base de dados pois já coletámos todos os atores e categorias deste.

A segunda query usada para povoar a coleção payments foi a seguinte:

Listing 1.6. Query para povoar a segunda coleção

```
SELECT s.store_id, c.first_name AS Costumer_first_name,
c.last_name AS Costumer_last_name, p.amount, p.payment_date,
st.first_name AS Staff_first_name, st.last_name
AS Staff_last_name,s.manager_staff_id,st.staff_id,st.email,
p.payment_id,c.customer_id,c.email FROM store AS s
```

```
LEFT JOIN customer AS c ON s.store_id = s.store_id
LEFT JOIN payment AS p ON c.customer_id = p.customer_id
LEFT JOIN staff AS st ON p.staff_id = st.staff_id
```

Um ciclo parecido com o anterior coleta as informações de cada linha devolvida pela query anterior. Seguidamente, verifica se já existe um objeto corresponde ao store_id devolvido. Caso não exista, cria esse documento. Posteriormente faz o mesmo com os clientes e com os pagamentos por essa ordem. Atualizando os arrays correspondentes de cada.

```
myStoreIt = iter(storeRecords)
while (True):
    try:
        i2 = next(myStoreIt)
    except StopIteration:
        print("second collection done")
        break

    costumer = i2[1] + " " + i2[2]
    costumer_id = i2[11]
    costumer_email = i2[12]
    manager_staff_id = i2[7]
    staff = i2[5] + " " + i2[6]
    staff_id = i2[8]
    staff_email = i2[9]
    date = i2[4]
    amount = float(i2[3])
    payment_id = i2[10]
    store = i2[0]

    storeInfo = {"store_id" : store, "manager_id" : manager_staff_id}
    paymentStaffInfo = {"staff_id" : staff_id, "name": staff, "email" : staff_email,"payment_id": payment_id,"amount" : amount,"date": date}
    customerInfo = {"customer_id": costumer_id,"name": costumer,"email": costumer_email}
    res = paymentList.find_one({"store_id": store})

    if res == None:
        paymentList.insert_one(storeInfo)
    res = paymentList.find_one({"paymentsByCostumer.customer_id": costumer_id,"store_id": store})
    if res == None:
        paymentList.update({"store_id": store},{ "$push": { "paymentsByCostumer" : customerInfo}})
    res = paymentList.find_one({"store_id": store,"paymentsByCostumer.customer_id": costumer_id,"paymentsByCostumer.payments.payment_id": payment_id,"paymentsByCostumer.payments.staff_id": staff_id})
    if res == None:
        res = paymentList.aggregate([{"$match":{"store_id": store,"paymentsByCostumer.customer_id": costumer_id}},{ "$project" : { "index" : { "$indexOfArray": ["$paymentsByCostumer.customer_id",costumer_id]}}}])
        ress = list(res)
        query = "paymentsByCostumer."+ str(ress[0]["index"]) +".payments"
        paymentList.update({"store_id": store,"paymentsByCostumer.customer_id": costumer_id},{ "$push": {query : paymentStaffInfo }})
```

Figura 2. Ciclo que gera cada documento Store guardado na base de dados MongoDB

3.4 Querys a Base de Dados

Em seguida listam-se algumas das querys feitas á base de dados Mongo e as respetivas respostas.

Primeiramente é executada uma query simples que devolve todos os titulos e as linguagens que cada filme tem. Poder-se-ia, facilmente, filtrar a base de dados usando qualquer um dos campos presentes na coleção films.

Listing 1.7. Query ao Mongo para devolver todos os Filmes

```
filmsList.find({},
{
        "title": 1,
        "original_language": 1,
        "foreign_language": 1,
        "_id": 0
})
```

Por exemplo, a seguinte query devolve todos os titúlos dos filmes que tenham estreado depois do ano 2005. Seguindo-se uma query para determinar os titúlos dos filmes da categoria Action.

Listing 1.8. Query ao Mongo para devolver todos os Filmes estreados depois de 2005

```
db.films.find(
{
        "release_year": {$gt : 2005}
},
{
        "title":1
})
```

Listing 1.9. Query ao Mongo para devolver todos os Filmes da Categoria Action

```
db.films.find(
{
        "categorys": "Action"
},
{
        "title":1
})
```

Finalmente, efetuamos uma query realizada sobre coleção payments. Esta query usa o aggregate para fazer "unwind"de um dos arrays presentes na coleção payments. De seguida, filtra só por aqueles que tem o customer_id igual ao desejado. E, por fim, devolve os objetos que contem os identificadores da store e todos pagamentos realizados pelo Cliente.

Listing 1.10. Query para obter as faturas de um dado Cliente organizado por Store

```
db.payments.aggregate(
[{"$unwind": "$paymentsByCostumer"},
 {"$match":{"paymentsByCostumer.customer_id" : 1}},
 {"$project" :{"store_id":"$store_id",
 "payments":"$paymentsByCostumer.payments"}}])
```

Note-se que este tipo de queries já é mais trabalhosa para o Mongo, logo esta não é a base de dados mais indicada para estas.

4 Neo4J

4.1 Criação da base de dados

4.1.1 Motivação

A base de dados gráfica é ideal para as queries que relacionam muitas entidades entre si. Assim, poder-se-ia fazer o paralelismo com as queries que necessitam de muitos Joins no modelo relacional. Esta base de dados assenta numa estrutura de grafos.

4.1.2 Criação - modificações

Para a criaçao desta base de dados supôs-se a mesma aplicação que para o Mongo, ou seja, uma aplicação de procura de filmes por atores, categoria, etc e também para consultar as faturas dos clientes por loja, quem emitiu e onde. Dado a natureza do tipo de base de dados ainda se conectaram estas duas partes.

4.2 Criação - Estrutura da Base de Dados

A estrutura da base dados pode ser dividida em dois uma parte que relaciona os Filmes com os Atores, Categorias e Idiomas e outra que relaciona os Pagamentos com os Cliente,Lojas e os Staff.Assim pode-se identificar os seguintes nodos para a primeira parte:

- Filme: Representa um filme.Contem o identificador(id), o título(title), o ano em que o filme estreou(release_year) e a descriçao do mesmo(description).
- Ator: Representa um ator que pode ter participado num Filme. Contem o identificador(id) do mesmo e o seu nome(name).
- Categoria: Representa uma categoria/género onde o Filme pode ser enquadrado. Contem o identificador e o nome da categoria(name).
- Idioma: Representa um dos idiomas que pode ser ouvidos os Filmes. Contem também o identificador (id) e a sua designação (name).

Os relacionamentos para esta primeira "parte"são:

- Atuou_Em: Relaciona os nodos Ator com os nodos Filme. Designa que atores participaram em que Filmes.
- Tem_Categoria: Relaciona os nodos Filme com os nodos Categoria. Está relação permite estabelecer quais as Categorias a que o Filme pertence.

- Idioma_Estrangeiro: Relaciona os nodos Filme com os nodos Idioma. Designa o Idioma estrangeiro em qual se pode ouvir o Filme.

- Idioma_Original: Relaciona os nodos Filme com os nodos Idioma. Designa o Idioma original do Filme. No entanto, a relação não é usada visto que os dados que povoam a base dados MySQL não contem nenhuma entrada deste tipo apesar de estar no modelo lógico.

A segunda parte, como foi mencionado, diz respeito aos Pagamentos. Os nodos criados para representar estes dados foram os seguintes:

- Loja: Representa uma Loja onde se poderiam fazer aluguer de Filmes. Contem um identificador(id) e um identificador de manager(manager_id). Para descobrir o nome do manager, por exemplo, cruza-se este identificador com os identificadores dos trabalhadores que estão nesta loja.

- Staff: Representa os trabalhadores das Lojas. Contem um identificador(id), o nome completo do mesmo (name) e o email (email).

- Cliente: Representa um cliente que já tenha feito algúm tipo de pagamento em umas das lojas. Contem um identificador do Cliente(id), o nome(name) e o email dele (email).

- Pagamento: Representa um pagamento realizado por um Cliente. Contem um identificador do Pagamento(id), a quantia que foi paga (amount) e a data em que foi realizado o mesmo (date).

Estes nodos são interligados com os seguintes relacionamentos:

- Trabalha_Em : Relaciona os nodos Staff com os nodos Loja. Designa que trabalhadores estão a trabalhar em que Loja.

- Pertence : Relaciona os nodos Pagamento com os nodos Clientes. Designa os Pagamentos que foram efetuados pelo /Pertencem ao Cliente.

- Emitida_Em : Relaciona os nodos Pagamento com os nodos Loja. Vinculando os Pagamentos as Lojas onde foram efetuadas.

- Emitida_Por : Relaciona os nodos Pagamentos com os nodos Staff. Vincula os pagamentos com que os emitiu.

Nas seguintes imagem encontra-se uma representação gráfica dos nodos e relacionamentos mencionados acima. Sendo a primeira imagem da primeira parte e segunda da segunda parte.

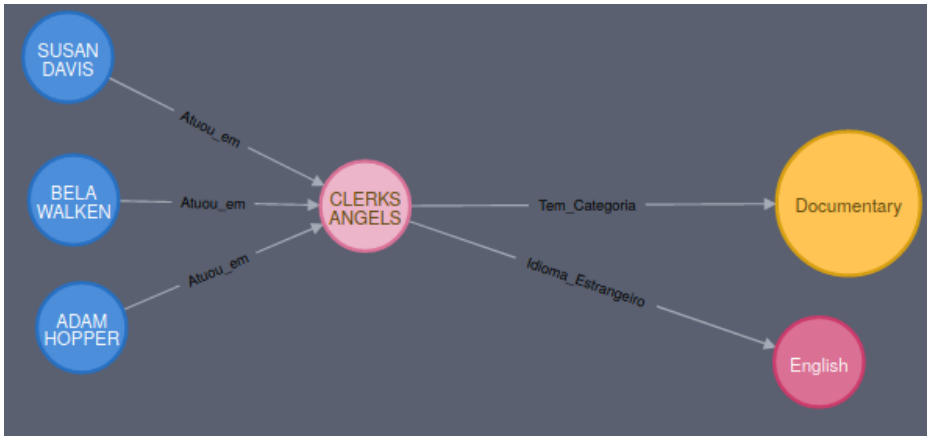


Figura 3. Representação gráfica dos nodos e Relacionamentos dos Filmes

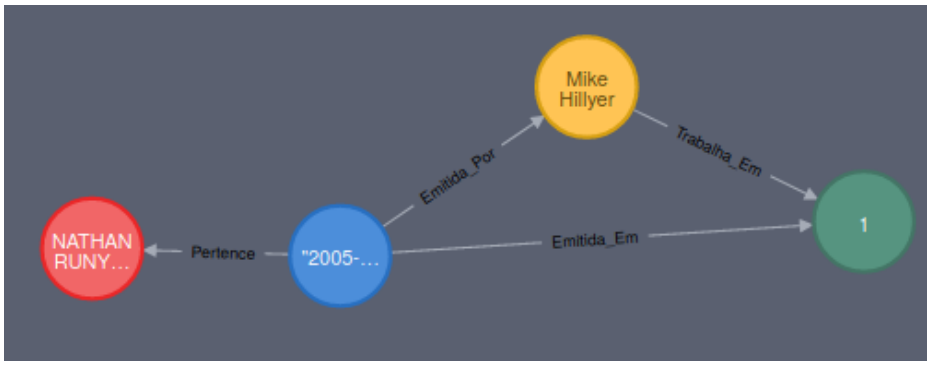


Figura 4. Representação gráfica dos nodos e Relacionamentos dos Pagamentos

De forma a conetar as duas partes e permitir queries mais interessantes, e mais possibilidades.Adicionou-se a relação Corresponde que conecta os nodos Pagamento e os nodos Filme, isto é, quais os pagamentos que correspondem a dado filme.

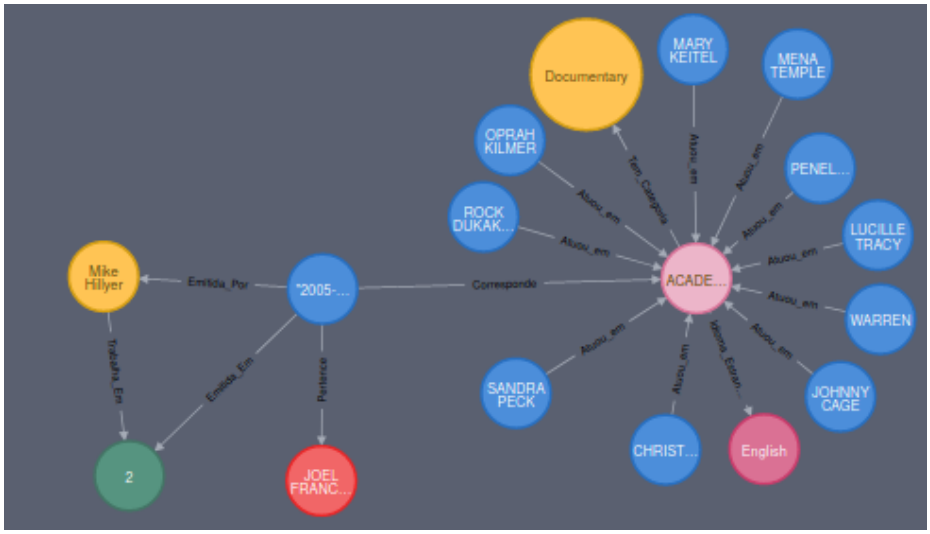


Figura 5. Representação Completo do grafo

4.3 Preenchimento da Base de Dados

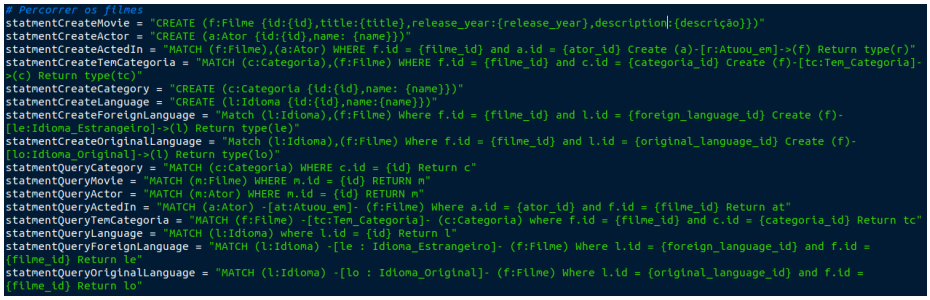
O preenchimento da base de dados foi realizada tendo como base os mesmos ciclos usados para migrar os dados para o MongoDB e também as mesmas queries ao MySQL(adicionando uns atributos das tabelas usadas para os Joins).

Fez também uso do Python e a Biblioteca correspondente de Neo4j, verificando sempre se os nodos ja existem para evitar repetições.

De seguida, apresentam-se as imagens com os Statments usados para criar ligações e para inserir os novos elementos na base de dados Neo4j. A primeira imagem corresponde ao subgrafo relacionado com Filmes e a segunda corresponde ao subgrafo dos Pagamentos, Clientes e etc. Já a terceira imagem corresponde á conexão entre os dois através de uma query ao MySql que junta as tabelas Rental, Payment e Inventory apresentada a seguir.

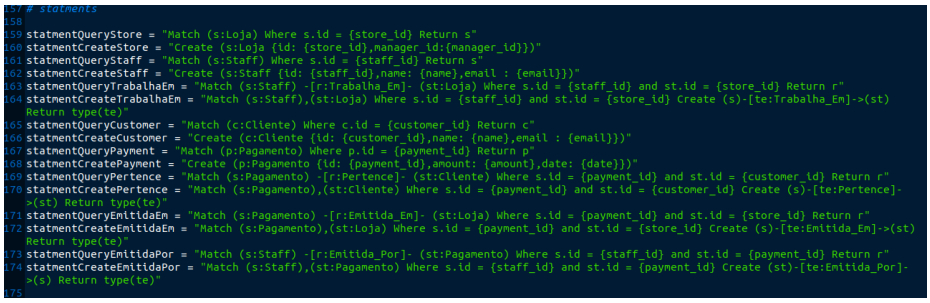
Listing 1.11. Query ao MySQL para fazer a conexão entre os dois subgrafos

```
SELECT payment.payment_id,rental.rental_id,inventory.film_id From
        payment Inner JOIN rental
                ON payment.rental_id = rental.rental_id
                 inner Join inventory
                        ON rental.inventory_id = inventory.inventory_id
```



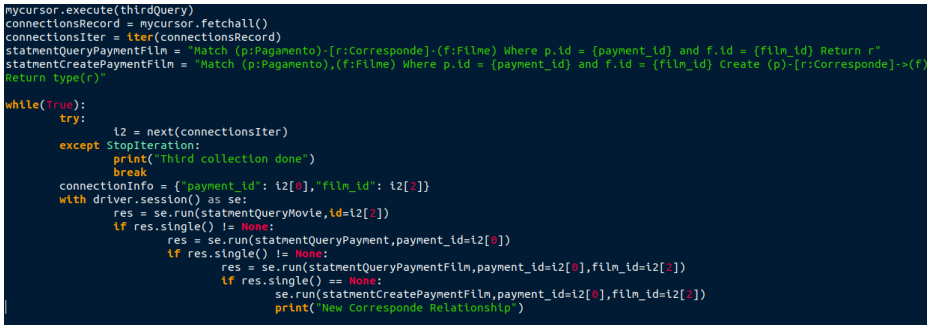
```
# Percorrer os filmes
statmentCreateMovie = "CREATE (f:Filme {id:{id},title:{title},release_year:{release_year},description:{descrição}})"
statmentCreateActor = "CREATE (a:Ator {id:{id},name: {name}})"
statmentCreateActedIn = "MATCH (f:Filme),(a:Ator) WHERE f.id = {filme_id} and a.id = {ator_id} Create (a)-[r:Atuou_em]->(f) Return type(r)"
statmentCreateTemCategoria = "MATCH (c:Categoria),(f:Filme) WHERE f.id = {filme_id} and c.id = {categoria_id} Create (f)-[tc:Tem_Categoria]-
>(c) Return type(tc)"
statmentCreateCategory = "CREATE (c:Categoria {id:{id},name: {name}})"
statmentCreateLanguage = "CREATE (l:Idioma {id:{id},name:{name}})"
statmentCreateForeignLanguage = "Match (l:Idioma),(f:Filme) Where f.id = {filme_id} and l.id = {foreign_language_id} Create (f)-
[le:Idioma_Estrangeiro]->(l) Return type(le)"
statmentCreateOriginalLanguage = "Match (l:Idioma),(f:Filme) Where f.id = {filme_id} and l.id = {original_language_id} Create (f)-
[lo:Idioma_Original]->(l) Return type(lo)"
statmentQueryCategory = "MATCH (c:Categoria) WHERE c.id = {id} Return c"
statmentQueryMovie = "MATCH (m:Filme) WHERE m.id = {id} RETURN m"
statmentQueryActor = "MATCH (m:Ator) WHERE m.id = {id} RETURN m"
statmentQueryActedIn = "MATCH (a:Ator) -[at:Atuou_em]- (f:Filme) Where a.id = {ator_id} and f.id = {filme_id} Return at"
statmentQueryTemCategoria = "MATCH (f:Filme) -[tc:Tem_Categoria]- (c:Categoria) where f.id = {filme_id} and c.id = {categoria_id} Return tc"
statmentQueryLanguage = "MATCH (l:Idioma) where l.id = {id} Return l"
statmentQueryForeignLanguage = "MATCH (l:Idioma) -[le : Idioma_Estrangeiro]- (f:Filme) Where l.id = {foreign_language_id} and f.id =
{filme_id} Return le"
statmentQueryOriginalLanguage = "MATCH (l:Idioma) -[lo : Idioma_Original]- (f:Filme) Where l.id = {original_language_id} and f.id =
{filme_id} Return lo"
```

Figura 6. Os Statments usados no código para migrar os Filmes



```
157 # statments
158
159 statmentQueryStore = "Match (s:Loja) Where s.id = {store_id} Return s"
160 statmentCreateStore = "Create (s:Loja {id: {store_id},manager_id:{manager_id}})"
161 statmentQueryStaff = "Match (s:Staff) Where s.id = {staff_id} Return s"
162 statmentCreateStaff = "Create (s:Staff {id: {staff_id},name: {name},email : {email}})"
163 statmentQueryTrabalhaEm = "Match (s:Staff) -[r:Trabalha_Em]- (st:Loja) Where s.id = {staff_id} and st.id = {store_id} Return r"
164 statmentCreateTrabalhaEm = "Match (s:Staff),(st:Loja) Where s.id = {staff_id} and st.id = {store_id} Create (s)-[te:Trabalha_Em]->(st)
    Return type(te)"
165 statmentQueryCustomer = "Match (c:Cliente) Where c.id = {customer_id} Return c"
166 statmentCreateCustomer = "Create (c:Cliente {id: {customer_id},name: {name},email : {email}})"
167 statmentQueryPayment = "Match (p:Pagamento) Where p.id = {payment_id} Return p"
168 statmentCreatePayment = "Create (p:Pagamento {id: {payment_id},amount: {amount},date: {date}})"
169 statmentQueryPertence = "Match (s:Pagamento) -[r:Pertence]- (st:Cliente) Where s.id = {payment_id} and st.id = {customer_id} Return r"
170 statmentCreatePertence = "Match (s:Pagamento),(st:Cliente) Where s.id = {payment_id} and st.id = {customer_id} Create (s)-[te:Pertence]-
    >(st) Return type(te)"
171 statmentQueryEmitidaEm = "Match (s:Pagamento) -[r:Emitida_Em]- (st:Loja) Where s.id = {payment_id} and st.id = {store_id} Return r"
172 statmentCreateEmitidaEm = "Match (s:Pagamento),(st:Loja) Where s.id = {payment_id} and st.id = {store_id} Create (s)-[te:Emitida_Em]->(st)
    Return type(te)"
173 statmentQueryEmitidaPor = "Match (s:Staff) -[r:Emitida_Por]- (st:Pagamento) Where s.id = {staff_id} and st.id = {payment_id} Return r"
174 statmentCreateEmitidaPor = "Match (s:Staff),(st:Pagamento) Where s.id = {staff_id} and st.id = {payment_id} Create (st)-[te:Emitida_Por]-
    >(s) Return type(te)"
175
```

Figura 7. Os Statments usados no código para migrar os Pagamentos



```
mycursor.execute(thirdQuery)
connectionsRecord = mycursor.fetchall()
connectionsIter = iter(connectionsRecord)
statmentQueryPaymentFilm = "Match (p:Pagamento)-[r:Corresponde]-(f:Filme) Where p.id = {payment_id} and f.id = {film_id} Return r"
statmentCreatePaymentFilm = "Match (p:Pagamento),(f:Filme) Where p.id = {payment_id} and f.id = {film_id} Create (p)-[r:Corresponde]->(f)
Return type(r)"

while(True):
        try:
                i2 = next(connectionsIter)
        except StopIteration:
                print("Third collection done")
                break
        connectionInfo = {"payment_id": i2[0],"film_id": i2[2]}
        with driver.session() as se:
                res = se.run(statmentQueryMovie,id=i2[2])
                if res.single() != None:
                        res = se.run(statmentQueryPayment,payment_id=i2[0])
                        if res.single() != None:
                                res = se.run(statmentQueryPaymentFilm,payment_id=i2[0],film_id=i2[2])
                                if res.single() == None:
                                        se.run(statmentCreatePaymentFilm,payment_id=i2[0],film_id=i2[2])
                                        print("New Corresponde Relationship")
```

Figura 8. Parte do código que relaciona os Filmes e os Pagamentos

4.4 Querys a Base de Dados

A seguir apresentam-se alguns tipos de queries que podem ser feitas á base de Dados gráfica.

A primeira retorna todos os filmes da Categoria Action, podendo adaptar-se para outro tipo de categorias ou para retornar todas categorias de um filme.

Listing 1.12. Query ao Neo4j para retornar os filmes da categoria ação

```
Match (f:Filme)-[:Tem_Categoria]-(c:Categoria)
        Where c.name = "Action"
                Return f
```

Esta próxima permite devolver todos os atores que participaram no filme de Categoria Action. Podendo-se alterar para retornar todos os Filmes em que um ator participou etc.

Listing 1.13. Query ao Neo4j para retornar todos os atores que participaram em filmes de Action

```
Match (f:Filme)-[r:Tem_Categoria]-(c:Categoria)
        ,(a:Ator)-[:Atuou_em]-(f)
                Where c.name = "Action"
                        Return a
```

Outra Query focada mais no lado dos Pagamentos permite retornar todos os Pagamentos realizados por um Cliente. Note-se que com uma pequena alteração poder-se-ia retornar a lista de pagamentos realizados por um Staff ou pertencentes a uma loja.

Listing 1.14. Query ao Neo4j para retornar todos os Pagamentos de um determindado Cliente

```
Match (c:Cliente)-[:Pertence]-(p:Pagamento)
        Where c.id = 1
                Return p
```

Finalmente, e uma das mais úteis, utilizando a ligação que conecta as duas partes podemos saber quais os filmes sobre os quais um cliente realizou pagamentos permitindo efetuar uma análise sobre os clientes.

Listing 1.15. Query ao Neo4j para retornar todos os filmes que um determinar Cliente já pagou por

```
Match (p:Pagamento)-[:Corresponde]-(f:Filme),
        (c:Cliente)-[:Pertence]-(p)
                Where c.id = 1
                        Return f
```

5 Conclusões

As três base de dados tem perspetivas diferentes de como fazer a persistência dos dados.

A primeira , em relação á original, não possui grande vantagem em termos de desempenho ou queries possíveis visto que os dois modelos são relacionais e representam , basicamente, o mesmo.

A base de dados documental, ao englobar os filmes e os pagamentos, cada uma na sua coleção, permite realizar queries eficientes sobre os elementos de topo. No caso da primeira coleção permite facilmente procurar por filmes, o ano da estreia do filme ou procurar todos os filmes de um determindado género. No entanto, tudo o que se centre nas categorias e atores tornar-se-á mais lento visto que ter-se-ia de inverter a base de dados por completo. Por exemplo, saber quais os filmes em que certo ator atuou, é muito útil para um sistema de pesquisa de filmes e muito custoso para a nossa base de dados Mongo ou organizar todos os filmes por categoria, isto é, todos os filmes que pertencem a categoria Action, todos os filmes que pertencem a categoria Games , etc.

Por outro lado, poder-se-ia ter feito uso da funcionalidade do Mongo parecida as chaves primárias/estrangeira para relacionar as duas coleções, porém, apesar de possível, não seria uma solução muito eficiente comparando com a vantagem de poder ter toda a informação num só documento.

A base de dados gráfica especializa-se nas conexões entre os nodos definidos para mesma, portanto qualquer query que tente relacionar vários nodos será mais eficiente aqui do que na versão documental ou relacional. Por exemplo, ao contrário do que na base de dados de Mongo, "descobrir"todos os filmes em que dado ator participou é bastante eficiente. No entanto, em principio, seria menos eficiente do que a base dados documental em querys sobre um só tipo de nodo, isto é, queries que nao relacionem alguns nodos.

Devido a própria natureza do Neo4j é fácil conectar os dois subgrafos que, são equivalente ás duas coleções do Mongo. Permitindo assim, fazer queries mais "interessantes"em que se relacionem os filmes e os pagamentos, como foi exemplificado numa das queries apresentadas no relatório mais acima.

Em suma, a base de dados relacional continua a ser a mais indicada para fazer transações e as não relacionais para suporte de algumas funcionalidades de um determindado serviço.