Implementações da database Sakila em bases de dados relacionais e noSQL

D. Barbosa[A78679], F. Oliveira[A78416], M. Silva[A79607], M. Leite[A78225], and V. Peixoto[A79175]

Universidade do Minho, Rua da Universidade 4710 - 057 Braga Portugal

Resumo Relatório relativo ao trabalho prático da unidade curricular de Bases de Dados NoSQL onde foi abordada a migração da base de dados *Sakila* para três sistemas distintos: *PL/SQL* da *Oracle*, *MongoDB* e *Neo4j*.

Keywords: Bases de Dados · MongoDB · Neo4j · Sakila · SQL · noSQL · Oracle · Modelo lógico · Migração de dados

1 Introdução

Uma das propostas mais recorrentes nos dias de hoje é a migração entre diferentes tipos de bases de dados. Essa migração pode decorrer de diversos motivos, desde à mudança de paradigma do *software* em questão, até a uma nova necessidade relativamente aos dados.

Assim sendo, neste trabalho pretende-se fazer a migração da base de dados *Sakila*, tipicamente em *MySQL* para uma outra base de dados relacional, em *PL/SQL* da *Oracle* e duas não relacionais (*MongoDB* e *Neo4j*).

Vamos referir as opções tomadas ao longo do trabalho, bem como as alterações efetuadas derivadas da mudança de paradigma (*SQL* vs. *NoSQL*) ou como diferenças entre *SQL* e *PL/SQL* foram ultrapassadas.

2 *Sakila Database*

Sakila Sample Database é uma base de dados disponibilizada pela plataforma *MySQL* no seu *website*, cujo objetivo é servir como esquema *standard* a ser usado como exemplo em livros, tutoriais, artigos científicos, entre outros. [1]

Analisando o modelo lógico que nos foi fornecido, nomeadamente as várias tabelas e os seus atributos (cujo significado atribuído a cada um destes consultamos na página referente à Sakila no *website* do MySQL), foi-nos possível concluir que a base de dados em questão aparenta ser apropriada para uso por parte de um negócio de aluguer de filmes que compreende várias lojas.

Após examinar a sua estrutura, foi-nos possível distinguir três *layers* distintas:

1. ***Layer* do Negócio**:

   A *layer* do Negócio é compostas por 4 tabelas: *staff, store, payment* e *rental*. A tabela *staff* contém as informações relativas a todos os funcionários da empresa, tais como o seu nome, fotografia, dados de *login*, endereço de correio eletrónico e a sua morada e loja onde trabalham (sob a forma de chaves estrangeiras para as tabelas correspondentes).
   A segunda tabela - *store* - lista as informações básicas de cada loja da cadeia, remetendo para as tabelas referentes a staff e morada através de chaves estrangeiras.
   Por fim, as tabelas *rental* e *payment*, estão relacionadas com transações de negócio, guardando dados como o funcionário que processou o pagamento e o aluguer, bem como o cliente que o requereu e o devolveu, o valor pago, o item alugado (cuja informação está guardada na tabela inventário) e a data prevista de entrega do mesmo.

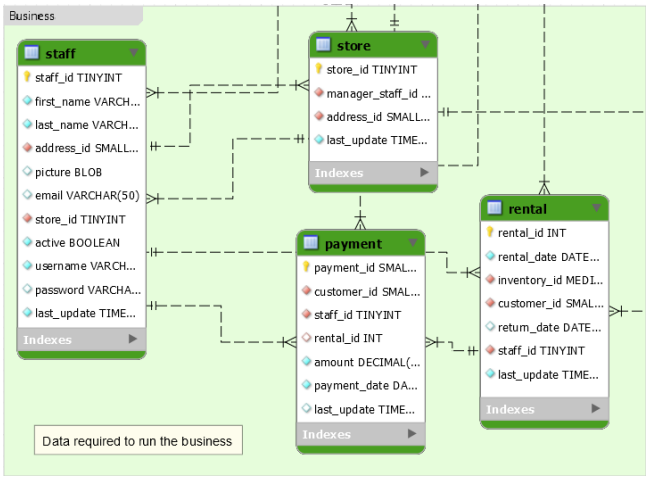

Figura 1. Conjunto das tabelas da *Layer "Business"*

2. ***Layer* dos dados do Cliente**:

   Este estrato do modelo lógico engloba várias tabelas que guardam a informação referente aos clientes. A tabela principal - *customer* - possui atributos relativos aos seus dados pessoais, tais como o nome do cliente (*first_name* e *last_name*), o seu endereço de correio eletrónico (*email*) e a sua morada (*address*).

   No que toca à morada, a tabela *customers* possui uma chave estrangeira para a tabela *address*, que por sua vez possui uma para a tabela *city* e, por fim, esta possui uma para a *country*. Esta situação (tabelas separadas para cidades e país) deve-se ao facto de entre os vários clientes haver muitos cujo

valor destes atributos é igual, existindo portanto um número considerável de valores repetidos. Assim, criando duas tabelas separadas e guardando apenas o seu *id*, consegue-se poupar memória, sendo a poupança mais significativa para um número elevado de registos.

É também importante referir que a *table address* contém as moradas não só dos clientes, mas também dos membros do *staff* e das lojas.

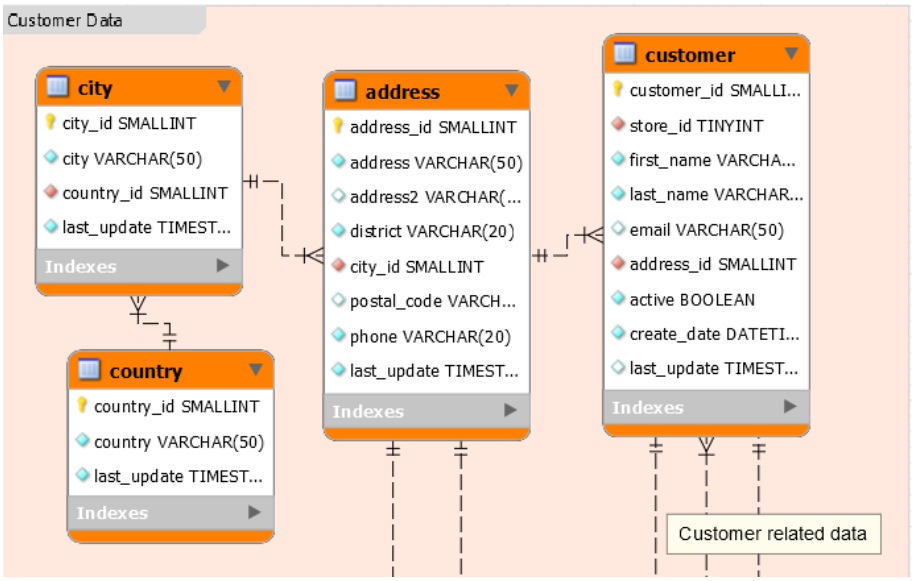

Figura 2. Conjunto das tabelas da *Layer "Customer Data"*

3. ***Layer* do Inventário**

Por fim, temos a *layer* do Inventário. Esta é composta por 8 tabelas, sendo que todas elas listam informações respeitantes aos filmes. As tabelas em questão são a `film`, `film_category`, `category`, `language`, `actor`, `film_actor`, `film_text` e a `inventory`.

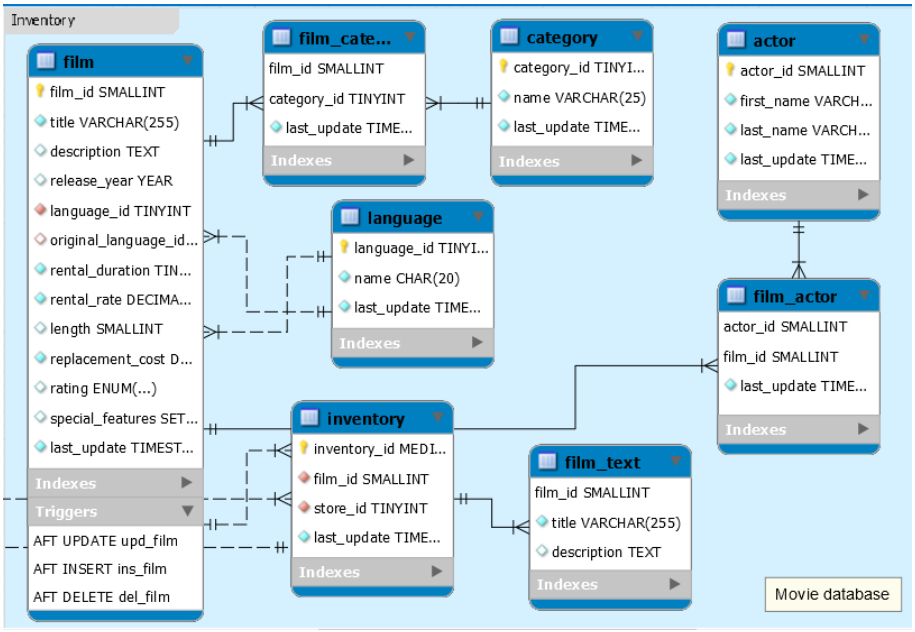

Figura 3. Conjunto das tabelas da *Layer "Inventory"*

3 *Queries*

Como método de teste de correção, relativamente à implementação da base de dados *Sakila* nos três sistemas de bases de dados, propusemos uma série de *queries*:

1. Top 10 dos clientes que mais alugueres efetuaram;
2. Top 10 dos clientes que mais filmes distintos alugaram;
3. Top 10 dos clientes que mais dinheiro gastaram;
4. Top 10 dos filmes mais alugados;
5. Top 10 dos membros do *staff* que processaram o maior volume de pagamentos;
6. Top 10 dos clientes que mais tempo demoram a devolver os filmes em média;
7. Top 10 das lojas com maior número de alugueres efetuados.

Espera-se assim, após a implementação da base de dados e introdução dos respetivos dados de povoamento, que a aplicação destas *queries* a cada um dos sistemas de base de dados apresente resultados iguais aos obtidos na versão original em *MySQL*.

4 Base de dados relacional – Oracle

4.1 Esquema desenvolvido

Tendo em conta que, o sistema de gestão de bases de dados *Oracle* é, tal como o *MySQL* relacional, o esquema de dados utilizado foi o mesmo. De salientar apenas que, alguns dos tipos de dados utilizados em *MySQL* não se encontram disponíveis em *Oracle*.

Em anexo, poderá ser visualizado também o *script* SQL de criação das tabelas do esquema *Sakila*.

4.2 Migração de dados

Relativamente à migração dos dados, utilizou-se a linguagem de programação *Python* para efetuar a interligação entre os dois sistemas de gestão de bases de dados.

Após uma observação de todas as tabelas do esquema de dados, observou-se uma forte relação entre as várias tabelas de dados (mais propriamente, a definição de chaves estrangeiras). Por forma a facilitar a execução da transferência de dados e, uma vez que é garantido que os dados presentes na base de dados *MySQL* se encontram consistentes, as restrições são adicionadas apenas no final de todo o processo. Desta forma, a transferência de dados torna-se bastante menos complexa, mas também mantendo todas as verificações essenciais ao bom funcionamento da base de dados após este processo de migração. Todos os *scripts* utilizados neste processo encontram-se junto deste documento.

Após se proceder então à execução do *script Python*, será necessário adicionar novamente todas as *constraints* necessárias ao bom funcionamento futuro da base de dados. Para esta etapa, todas as *constraints* foram reunidas num *script SQL* em separado, utilizando o *statement* `ALTER TABLE` para se proceder à sua adição. Desta forma, é possível perceber também se os dados inseridos se encontram consistentes.

4.3 Desafios

Uma vez que, nos encontramos perante dos sistemas de gestão de bases de dados relacionais (*MySQL* e *Oracle*), não existem grandes diferenças no que toda ao esquema de dados em si.

Na conversão da criação das tabelas, encontramos alguns problemas no nosso caminho. Abaixo apresentamos quais e qual a abordagem efetuada para a sua resolução:

- **Tipos de dados incompatíveis:** Alguns tipos de dados não são compatíveis entre *MySQL* e *PL/SQL* da *Oracle*, pelo que tivemos de usar um guia de conversão entre tipos de dados.
- **Atualização da *timestamp*:** Ao atualizar um tuplo em qualquer tabela, a sua *timestamp* deve ser atualizada automaticamente para a hora e dia atual. Em *MySQL* essa regra era introduzida na criação das tabelas, mas em *Oracle* foi necessário criar um *trigger* para cada tabela.

– ***Keys* inexistentes:** As *keys* foram convertidas para *indexes* em *Oracle*, uma vez que este não as suporta na criação de tabelas.
– **Autoincrementação nas chaves primárias:** A sintaxe de autoincrementação de uma variável é distinta entre os dois sistemas relacionais.

Apresenta-se apenas de seguida a definição em *MySQL* e em *Oracle* da tabela *actor*, por forma a perceber quais as principais diferenças entre estes dois *SGBD's*.

```
CREATE TABLE actor (
  actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (actor_id),
  KEY idx_actor_last_name (last_name)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Figura 4. Apresentação da definição SQL em *MySQL*.

```
CREATE TABLE actor (
  actor_id SMALLINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 1),
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT actor_PK PRIMARY KEY (actor_id));
CREATE INDEX actor_last_name_IDX ON actor (last_name);
CREATE OR REPLACE TRIGGER actor_timestamp_trigger
    BEFORE UPDATE ON actor
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/
```

Figura 5. Apresentação da definição SQL em *Oracle*.

4.4 *Queries* implementadas

Foram implementadas todas as *queries* propostas inicialmente. Em baixo pode-se comprovar o código *SQL* para cada uma das *queries*:

1. Top 10 dos clientes que mais alugueres efetuaram

```
SELECT
    c.customer_id AS ID,
    (c.first_name || ' ' || c.last_name) AS nome,
    COUNT(r.rental_id) AS num_alugueres
FROM customer c
INNER JOIN rental r ON (r.customer_id=c.customer_id)
WHERE c.active=1
```

```
GROUP BY c.customer_id, (c.first_name || ' ' || c.last_name)
ORDER BY num_alugueres DESC
FETCH FIRST 10 ROWS ONLY;
```

2. Top 10 dos clientes que mais filmes distintos alugaram

```
SELECT
    c.customer_id AS ID,
    (c.first_name || ' ' || c.last_name) AS nome,
    COUNT(i.film_id) AS num_filmes
FROM customer c
INNER JOIN rental r ON (r.customer_id=c.customer_id)
INNER JOIN inventory i ON (i.inventory_id=r.inventory_id)
WHERE c.active=1
GROUP BY c.customer_id, (c.first_name || ' ' || c.last_name)
ORDER BY num_filmes DESC
FETCH FIRST 10 ROWS ONLY;
```

3. Top 10 dos clientes que mais dinheiro gastaram

```
SELECT
    c.customer_id AS ID,
    (c.first_name || ' ' || c.last_name) AS nome,
    SUM(p.amount) AS gastos
FROM customer c
INNER JOIN rental r ON (r.customer_id=c.customer_id)
INNER JOIN payment p ON (p.rental_id=r.rental_id)
WHERE c.active=1
GROUP BY c.customer_id, (c.first_name || ' ' || c.last_name)
ORDER BY gastos DESC
FETCH FIRST 10 ROWS ONLY;
```

4. Top 10 dos filmes mais alugados

```
SELECT
    f.film_id AS ID,
    (f.title || ' (' || f.release_year || ')') AS filme,
    COUNT(r.rental_id) AS num_alugueres
FROM film f
INNER JOIN inventory i ON (i.film_id=f.film_id)
INNER JOIN rental r ON (r.inventory_id=i.inventory_id)
GROUP BY f.film_id, (f.title || ' (' || f.release_year || ')')
ORDER BY num_alugueres DESC
FETCH FIRST 10 ROWS ONLY;
```

5. Top 10 dos membros do staff que processaram o maior volume de pagamentos

```
SELECT
    s.staff_id AS ID,
    (s.first_name || ' ' || s.last_name) AS nome,
    SUM(p.amount) as volume
FROM staff s
INNER JOIN payment p ON (p.staff_id=s.staff_id)
GROUP BY s.staff_id, (s.first_name || ' ' || s.last_name)
ORDER BY volume DESC
FETCH FIRST 10 ROWS ONLY;
```

6. Top 10 dos clientes que mais tempo demoram a devolver os filmes em média

```
SELECT
    c.customer_id AS ID,
    (c.first_name || ' ' || c.last_name) AS nome,
    AVG(r.return_date-r.rental_date) AS demora (dias)
FROM customer c
INNER JOIN rental r ON (r.customer_id=c.customer_id)
WHERE r.return_date IS NOT NULL
  AND c.active=1
GROUP BY c.customer_id, (c.first_name || ' ' || c.last_name)
ORDER BY demora DESC
FETCH FIRST 10 ROWS ONLY;
```

7. Top 10 das lojas com maior número de alugueres efetuados

```
SELECT
    s.store_id AS ID,
    COUNT(r.rental_id) as alugueres
FROM store s
INNER JOIN staff sf ON (sf.store_id=s.store_id)
INNER JOIN rental r ON (r.staff_id=sf.staff_id)
GROUP BY s.store_id
ORDER BY alugueres DESC
FETCH FIRST 10 ROWS ONLY;
```

5 Base de dados orientada a documentos - MongoDB

5.1 Modelo de Dados

Relativamente ao modelo de dados de dados criado, seguem-se algumas considerações sobre o mesmo.

Número de Identificação No modelo original da *Sakila Database*, em `SQL`, toda as entidades possuem um *ID* como chave primária, dado que é essencial em qualquer base de dados relacional. Na versão criada em *Mongo*, apesar de não ser obrigatória a sua existência, manter os *IDs* tem as suas vantagens e, portanto, foi essa a decisão tomada. No entanto, existem algumas exceções, nomeadamente *IDs* que deixaram de ser necessários. Temos como exemplo, a tabela *city* que, quando unida com a tabela *address*, os seus identificadores deixam de ser necessários. Para além deste caso, existem também *IDs* que apenas têm uso temporário durante a importação, como por exemplo, o *ID* da tabela *address* que é útil apenas para a agregação de tabelas no *Mongo* sendo então posteriormente eliminado. Ainda relativamente a números de identificação, é importante referir que devido à tabela *inventory* ser eliminada, algumas tabelas passar a incluir certos *IDs* de modo a facilitar algumas *queries*.

Nomeadamente:

- *rental* recebe o ID do *film* e da *store*.
- *film* recebe ID das *stores* onde existe.
- *store* recebe ID dos *films* que tem.

Atributos removidos Aquando da criação do esquema da base de dados em *MongoDB* apercebemo-nos que alguns atributos da *Sakila* se encontram sempre a `null` em todo o povoamento. Assim, uma vez que *MongoDB* não obriga a uma uma estrutura rígida como em `SQL`, optamos por não incluir esses atributos nos documentos criados.

Os atributos dos quais falamos são:

- o atributo address2 (tabela *address*);
- o original_language (tabela *film*).

Casos excecionais Durante o processo de criação do novo esquema, notamos que alguns pagamentos (*payments*) não se encontravam associados a nenhum aluguer (*rental*). Assim, como são casos esporádicos, optamos por isolar estes dados em documentos à parte numa coleção a que demos o nome de *Other Payments*.

Esquema desenvolvido O esquema desenvolvido para *MongoDB* conta com 5 coleções: uma com as informações relativas aos clientes, uma segunda com as informações relativas aos filmes, uma terceira com as informações relativas às lojas, uma quarta com as informações relativas aos alugueres/pagamentos e, por fim, uma para os pagamentos não associados a nenhum aluguer, tal como mencionado acima.

A estrutura do esquema é a que se segue:

```
>Customer
    *Address
```

```
>Film
    >Category
    >Actor
    >Stores (ID da store apenas)

>Store
    *Address
    *Manager
    >Staff
        *Address
    >Films (ID apenas)

>Rental + Payment
    *Film (nome/ID apenas)
    *Store (ID apenas)
    *Staff (ID apenas)
    *Customer (ID apenas)

>Other Payments
```

5.2 Migração de dados

A migração foi marcada pela fase de exportação no *MySQL* e posterior importação no *MongoDB*. Após investigação, concluímos que o melhor método de migração entre *MySQL* e *MongoDB* era com o uso de CSVs.

Exportação Ainda antes de exportarmos as tabelas para CSV, por conveniência, fizemos a união de algumas tabelas ainda no *MySQL*. Tal foi feito para as tabelas:

- *address, city, country*;
- *film* e *language*;
- *category* e *film_category*;
- *actor* e *film_actor*;
- *rental* e *payment*.

De seguida, procedemos à exportação das tabelas. Segue-se o exemplo da exportação das tabelas *address*, *city* e *country*.

```
select address_id, address, district, postal_code, phone,
        ST_AsText(location), city, country, address.last_update

from address, city, country
```

```
    where city.country_id = country.country_id

    and address.city_id = city.city_id

    into outfile 'sakila_nosql/csv/address_city_countryAUX.csv'

    FIELDS TERMINATED BY ','

    OPTIONALLY ENCLOSED BY '"'

    LINES TERMINATED BY '\n';
```

Importação Para a importação criamos um script responsável por automaticamente pegar nos CSVs gerados pelo *MySQL* e terminar a migração.

Para isso primeiramente importamos os CSVs para a nossa base de dados, criando um documento para cada CSV. Contudo a importação não acaba aqui. Usamos os documentos criados como auxilio para criar os documentos finais, fieis ao modelo planeado. Esta fase usou `lookup` para juntar documentos e assim podermos criar documentos embebidos, como por exemplo *address* embebido no *customer*. Foi nesta fase que também removemos possíveis atributos nulos que ainda existissem.

Convém ainda notar que sempre que juntamos tabelas a data mantida tende a ser a da tabela principal, sendo a data da tabela que se juntou descartada. Num outro ponto, sempre que juntamos tabelas e embebemos a tabela adicionada (ex: address e costumer) a data para o last updated é mantida em ambas as tabelas. Por fim, first e last name foram sempre concatenados para uma única variável, normalmente chamada 'name'.

Segues-se um excerto do script de importação.

```
# Customer
mongoimport.exe --db nosql --type csv --file
    "./csv/address_city_countryAUX.csv" --fields "address_id","address",
        "district","postal_code","phone","location","city","country","last_update"

mongoimport.exe --db nosql --type csv --file
    "./csv/customerAUX.csv" --fields "customer_id","store_id","name",
        "email","address_id","active","create_date","last_update"

...

echo " > JOINS !!!"
## Agregar address ao customer
mongo.exe nosql --eval "db.customerAUX.aggregate([
    {\$lookup: {
        from: 'address_city_countryAUX',
```

```
        localField: 'address_id',
        foreignField: 'address_id',
        as: 'address'
    }},
    {\$unwind:'\$address'},
    {\$project:{
        'customer_id':1,
        'store_id':1,
        'name':1,
        'email':1,
        'active':1,
        'create_date':1,
        'address':{
            'address':1,
            'district':1,
            'postal_code':1,
            'location':1,
            'city':1,
            'country':1,
            'phone':1,
            'last_update':1
        },
        'last_update':1
    }},
    {\$out:'customer'}
])"
```

5.3 *Queries* implementadas

As *queries* implementadas são as que se seguem:

1. Top 10 dos clientes que mais alugueres efetuaram

```
    db.rental.aggregate([
    {$lookup: {
        from: 'customer',
        localField: 'customer_id',
        foreignField: 'customer_id',
        as: 'CUSTOMER'
    }},
    {$unwind: '$CUSTOMER'},
    {$project:{
        _id:0,
        rental_id:1,
```

```
        customer_id:1,
        customer_name: "$CUSTOMER.name",
        active: "$CUSTOMER.active"
    }},
    {$match: {active:1}},
    {$group:{
        _id:{
            customer_id:'$customer_id',
            customer_name:'$customer_name'
        },
        numberRentals: {$sum: 1},
        listRentals: {$push: '$rental_id'}
    }},
    {$sort:{numberRentals:-1}},
    {$limit:10}
])
```

2. Top 10 dos clientes que mais filmes distintos alugaram

```
db.rental.aggregate([
{$lookup: {
    from: 'customer',
    localField: 'customer_id',
    foreignField: 'customer_id',
    as: 'CUSTOMER'
}},
{$unwind: '$CUSTOMER'},
{$project:{
    _id:0,
    rental_id:1,
    customer_id:1,
    customer_name: "$CUSTOMER.name",
    active: "$CUSTOMER.active"
}},
{$match: {active:1}},
{$group:{
    _id:{
        customer_id:'$customer_id',
        customer_name:'$customer_name'
    },
    setFilms: {$addToSet: '$rental_id'}
}},
{$project:{
```

```
        _id:1,
        numberFilms: {$size: '$setFilms'},
        // setFilms:1
    }},
    {$sort:{numberFilms:-1}},
    {$limit:10}
])
```

3. Top 10 dos clientes que mais dinheiro gastaram

```
    db.rental.aggregate([
    {$lookup: {
        from: 'customer',
        localField: 'customer_id',
        foreignField: 'customer_id',
        as: 'CUSTOMER'
    }},
    {$unwind: '$CUSTOMER'},
    {$project:{
        _id:0,
        rental_id:1,
        customer_id:1,
        customer_name: "$CUSTOMER.name",
        active: "$CUSTOMER.active",
        amount:1
    }},
    {$match: {active:1}},
    {$group:{
        _id: {
            customer_id:'$customer_id',
            customer_name:'$customer_name'
        },
        totalAmount: {$sum: '$amount'},
        listAmounts: {$push: '$amount'}
    }},
    {$sort:{totalAmount:-1}},
    {$limit:10}
])
```

4. Top 10 dos filmes mais alugados

```
    db.rental.aggregate([
```

```
    {$project:{
        _id:0,
        rental_id:1,
        film_id:1,
        film_title:1,
    }},
    {$group:{
        _id: {
            film_id: '$film_id',
            filmTitle: '$film_title'
        },
        nRentals: {$sum: 1},
        listAmounts: {$push: '$rental_id'}
    }},
    {$sort:{nRentals:-1}},
    {$limit:10}
])
```

5. Top 10 dos membros do staff que processaram o maior volume de pagamentos

```
    db.store.aggregate([
    {$project:{
        _id:0,
        store_id:1,
        staff:{
            staff_id:1,
            staff_name:1,
            active:1
        }
    }},
    {$unwind: '$staff'},
    {$match: {'staff.active':1}},
    {$group:{
        _id: {
            staff_id: '$staff.staff_id',
            staff_name: '$staff.staff_name',
        },
    }},
    {$lookup: {
        from: 'rental',
        localField: '_id.staff_id',
        foreignField: 'staff_id',
        as: 'RENTALS'
```

```
    }},
    {$unwind: '$RENTALS'},
    {$group:{
        _id: '$_id',
        totalAmount: {$sum: '$RENTALS.amount'},
        numberAmounts: {$sum: 1},
        // listAmounts: {$push: '$RENTALS.amount'}
    }},
    {$sort:{totalAmount:-1}},
    {$limit:10}
])
```

6. Top 10 dos clientes que mais tempo demoram a devolver os filmes em média

```
    db.rental.aggregate([
    {$match:{return_date:{$exists:true}}},
    {$project:{
        _id:0,
        rental_id:1,
        rental_date: {$dateFromString: {
            dateString:"$rental_date",format:"%Y-%m-%d %H:%M:%S"}},
        return_date:  {$dateFromString: {
            dateString:"$return_date",format:"%Y-%m-%d %H:%M:%S"}},
        customer_id:1,
    }},
    {$project:{
        _id:0,
        rental_id:1,
        rental_date:1,
        return_date:1,
        SECS: {$divide: [{$subtract: ['$return_date', '$rental_date']}, 1000]},
        // HOURS: {$divide: [{$subtract: ['$return_date', '$rental_date']}, 3600000]},
        // DAYS: {$divide: [{$subtract: ['$return_date', '$rental_date']}, 86400000]},
        customer_id:1,
    }},
    {$lookup: {
        from: 'customer',
        localField: 'customer_id',
        foreignField: 'customer_id',
        as: 'CUSTOMER'
    }},
    {$unwind: '$CUSTOMER'},
```

```
    {$match: {"CUSTOMER.active":1}},
    {$group:{
        _id:{
            customer_id:'$customer_id',
            customer_name:'$CUSTOMER.name'
        },
        avgReturnSeconds: {$avg: '$SECS'},
        listAVG: {$push: '$SECS'}
    }},
    {$project:{
        _id:1,
        avgReturnSeconds:1,
        avgReturnDays: {$divide: ['$avgReturnSeconds', 86400]},
        // listAVG:1
    }},
    {$sort:{avgReturnSeconds:-1}},
    {$limit:10}
])
```

7. Top 10 das lojas com maior número de alugueres efetuados

```
    db.rental.aggregate([
    {$group:{
        _id:{
            store_id:'$store_id',
        },
        numberRentals: {$sum: 1},
        // listRentals: {$push: '$rental_id'}
    }},
    {$sort:{numberRentals:-1}},
    {$limit:10}
])
```

5.4 Comparação com o modelo relacional

A modelação em *MongoDB*, por ser orientada a documentos, é bastante diferente da modelação relacional. Assim sendo o nosso esquema em *MongoDB* agregou varias tabelas cuja a informação era similar (ex: *address*, *city*, *country*) ou que podia ser mais facilmente percetível quando reunida num mesmo documento (ex: *films*, *category*, *language*, *actor*).

Por fim, outra diferença prende-se com o facto de o*MongoDB* não obrigar a uma estrutura rígida e esta flexibilidade levou-nos a não incluir nos documentos

criados alguns atributos que se encontravam sempre a `null` e ainda a ajustar os identificadores conforme estes fossem ou não necessários.

6 Base de dados orientada a grafos - Neo4j

6.1 O Modelo

Para fazer a conversão de uma base de dados de tipo relacional para uma orientada a grafos foi necessário realizar diversas alterações ao esquema, com a finalidade de obter um modelo do qual o *Neo4j* retiraria mais proveito.

Número de Identificação No modelo original *Sakila*, toda as entidades têm um *ID* como chave primária, dado que é essencial em qualquer base de dados relacional. Nesta base de dados orientada a grafos, não é necessário estabelecer números de identificação manualmente para criar relacionamentos entre entidades, portanto, a nossa primeira decisão foi remover todos os atributos *ID* a entidades que não precisassem, isto é, que tivessem outro elemento identificativo. A única entidade à qual não foi removida foi a *Store*.

Tabelas Desnecessárias No processo de conversão, a maior parte das tabelas foram convertidas de um forma muito direta para a base de dados não relacional, enquanto que outras se revelaram desnecessárias e não foram incluídas. Todas as tabelas que representavam um relacionamento *many to many* não fazia sentido serem migradas para *Labels de Neo4j*, daí as seguintes tabelas terem-se tornado em relações:

- `film_category`
- `film_actor`
- `inventory`

Porém, no último caso, `inventory` não era uma tabela *many to many* típica, tendo em conta que tinha um *ID* próprio, o que permitia criar mais que uma relação entre as duas entidades que conectava, sendo utilizada para contabilizar o stock de filmes duma loja através do número de entradas. Para substituir essa funcionalidade, foi criado um relacionamento `(Store)-[HAS_FILM]->(Film)`, com uma propriedade `How_many`, um inteiro que representava o stock daquele filme naquele loja. Para além das tabelas *many to many*, uma tabela que optámos por desconsiderar completamente foi a `film_text`, pois tendo em conta que apenas tem informação já contida na tabela `film` é totalmente redundante e provavelmente apenas foi criada para acelerar alguma *query*, o que não nos fez sentido trazer para o *Neo4j*,

Elaboração do Modelo Após tirarmos estas conclusões, optámos por fazer um esboço do esquema que o nosso modelo iria seguir, estando este representado na figura 6.

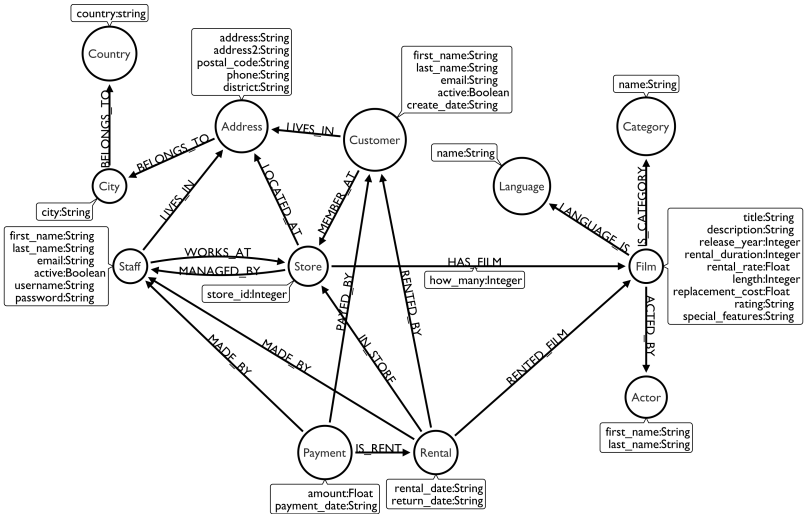

Figura 6. Modelo em *Neo4j*

6.2 Migração de dados

Após termos definido aquilo que queríamos obter, o passo seguinte foi averiguar qual o melhor método para o alcançar.

O Método Após investigarmos, concluímos que os melhores métodos de migração de uma base de dados relacional para uma orientada a grafos, consistem em primeiro passar os dados para ficheiros *CSV* e posteriormente importá-los para o *Neo4j* [3]. Começámos então a criar os ficheiros *CSV* de acordo com a estrutura que planeámos implementar no *Neo4j*. Exemplificando com as tabelas `Country` e `City`, criámos dois ficheiros *CSV* na diretoria onde está instalado o *Neo4j*, através dos seguinte comandos:

—Exportação de `Country`:

```
select * from country
into outfile '/path/to/neo/libexec/import/country.csv'
fields terminated by ','
lines terminated by '\n';
```

—Exportação de `City`:

```
select city, country from city
inner join country on city.country_id = country.country_id
into outfile '/path/to/neo/libexec/import/city.csv'
fields terminated by ','
lines terminated by '\n';
```

O passo seguinte seria importar os dados dos ficheiros para o *Neo4j*. No mesmo exemplo, foram criados os seguintes comandos:

```
load csv from "file:///country.csv" as line
create (m :Country {country : line[1]})
```

—Criação de índice no nome do país:

```
create index on :Country(country)
```

—Criação das cidades já relacionadas com os respetivos países:

```
load csv from "file:///city.csv" as line
match (co : Country)
where co.country = line[1]
create  (c : City {city: line[0]}),
(c)-[r:BELONGS_TO]->(co)
```

Algo relevante a notar, é que entidades como os Pagamentos e os Alugueres não têm nenhum atributo suficientemente identificativo, portanto a nossa abordagem para migrar essas entidades consistiu em, inicialmente, migrarmos com *ID* e após criar os relacionamentos apagarmos o atributo para todas as instâncias.

6.3 *Queries* implementadas

1. Top 10 dos clientes que mais alugueres efetuaram

```
match (c:Customer)<-[:RENTED_BY]-(r:Rental)
return c, count(r)
order by count(r) desc
limit 10;
```

2. Top 10 dos clientes que mais filmes distintos alugaram

```
match (c:Customer)<-[:RENTED_BY]-(r:Rental)-[:RENTED_FILM]->(f:Film)
return c, count(distinct f)
order by count(distinct f) desc
limit 10;
```

3. Top 10 dos clientes que mais dinheiro gastaram

```
match (c:Customer)<-[:PAYED_BY]-(p:Payment)
return c, sum(p.amount)
order by sum(p.amount) desc
limit 10
```

4. Top 10 dos filmes mais alugados

```
match (f:Film)<-[a:RENTED_FILM]-(r:Rental)
return f, count(a)
order by count(a) desc
limit 10
```

5. Top 10 dos membros do staff que processaram o maior volume de pagamentos

```
match (s:Staff)<-[:MADE_BY]-(p:Payment)
return s, sum(p.amount)
order by sum(p.amount) desc
limit 10
```

6. Top 10 dos clientes que mais tempo demoram a devolver os filmes em média

```
match (c:Customer)<-[:RENTED_BY]-(r:Rental)
where exists(r.return_date)
with c,
avg(duration.between(datetime(r.rental_date), datetime(r.return_date)))
as avg_dur
return c, avg_dur
order by avg_dur desc
limit 10
```

7. Top 10 das lojas com maior número de alugueres efetuados

```
match (s:Store)<-[:WORKS_AT]-(c:Staff)<-[:MADE_BY]-(p:Rental)
return s,count(p)
order by count(p)
limit 5
```

6.4 Comparação com o modelo relacional

A migração de *MySQL* para *Neo4j* trouxe um modelo mais simples, por não haver necessidade de serem criadas tabelas somente para os relacionamentos *n para n*. Como também é típico para o *Neo4j*, exploração que envolvesse mais que uma entidade também se tornou muito mais simples. Para resolver as mesmas queries em *MySQL*, seriam precisos scripts consideravelmente mais complexos. Uma desvantagem sentida, porém, terá sido devido ao facto de os atributos serem fracamente tipados, o que durante a implementação da nova base de dados por várias vezes resultou em erros que seriam mais fáceis de perceber em *MySQL*.

7 Conclusão

Para cada um dos motores de bases de dados, foram necessárias diferentes estratégias e técnicas para que a transferência da informação fosse possível.

Cada um destes, apresenta vantagens e desvantagens, o que faz com que estes sejam aplicáveis em diferentes situações. Desta forma, o grupo ficou a conhecer com maior detalhe cada um dos SGBD tornando assim no futuro a escolha de um destes conforme o contexto de aplicação, uma vez que, foram implementados neste trabalho prático num contexto de complexidade já elevada.

Concluindo, todos os SGBD's foram implementados com sucesso e toda a informação foi transferida na integra. Como um primeiro contacto em grande escala com estas bases de dados, a experiência revelou-se benéfica e bastante desafiante. No futuro, estes mesmos métodos poderão ser também aplicados a SGBD's da mesma natureza apenas com pequenos ajustes, uma vez que se encontram representados neste trabalho prático os principais tipos de bases de dados.

Referências

1. Sakila Sample Database,
   `https://dev.mysql.com/doc/sakila/en/sakila-introduction.html`.
   Último acesso 21/11/2018
2. *NoSQL databases explained*,
   `https://www.mongodb.com/nosql-explained`.
   Último acesso: 22/11/2018
3. *Tutorial: Import Data Into Neo4j*
   `https://neo4j.com/developer/guide-importing-data-and-etl/`
   Último acesso: 06/01/2019
4. *Oracle Database SQL Developer Supplementary Information for MySQL Migrations*,
   `https://docs.oracle.com/cd/E12151_01/doc.150/e12155.pdf`,
   Chuck Murray, Oracle, Release 1.5, 2008.