Universidade do Minho
Mestrado Integrado em Engenharia Informática

Unidade Curricular de
Bases de Dados NOSQL

Ano Letivo de 2019/2020

Migração *Sakila*

Ana Ribeiro - A82474
Jéssica Lemos - A82061
Pedro Freitas - A80975

Janeiro, 2020

Data de Recepção	
Responsável	
Avaliação	
Observações	

Migração *Sakila*

Ana Ribeiro - A82474
Jéssica Lemos - A82061
Pedro Freitas - A80975

Janeiro, 2020

Resumo

Este trabalho que foi realizado no âmbito da unidade curricular de Base de Dados NoSQL, consistiu na migração da base de dados *Sakila* para uma outra base de dados relacional, em *Oracle* e duas não relacionais, uma orientada a documentos, *Mongo*, e outra a grafos *Neo4j*.

Inicialmente foi criado um esquema para cada um dos novos sistemas, equivalente ao esquema da base de dados relacional fornecido, sendo posteriormente explicado os processos para fazer a migração dos dados contidos na base de dados fornecida. Foi ainda implementado um conjunto de *queries* com o intuito de demonstrar a operacionalidade dos sistemas.

Por fim realizamos uma análise crítica do trabalho realizado, comparando os modelos e as funcionalidades agora implementadas.

Palavras-Chave: Bases de Dados Relacional, Bases de Dados Relacional, MongoDB, Neo4j, Sakila, SQL, noSQL, Oracle, Migração de dados

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

1.1 Contextualização

Nos dias de hoje existe uma grande evolução tecnológica e com isso surgem novas ferramentas e abordagens aos mais variados problemas. Havendo uma enorme procura de conseguirmos responder a todas as necessidades dos utilizadores da maneira mais eficiente possível, é recorrente existir propostas de atualizações do sistema como um todo, o que pode implicar uma mudança brusca no produto em si e como os dados são armazenados.

Assim iremos neste projeto executar diferentes migrações de dados de forma a tornar o nosso sistema mais funcional, apelativo e com melhor *performance* para o cliente, abordando sistemas relacionais e não relacionais.

1.2 Apresentação do Caso de Estudo

Para podermos realizar a migração de dados para os diferentes novos sistemas para ter como caso de estudo a base de dados *Sakila*, implementada num sistema relacional com *MySQL*.

Esta base de dados servirá de exemplo para as novas estratégias e implementações sendo expectável obtermos as mesmas respostas que obtemos na implementação atual.

Visto que a *Sakila* já vem como exemplo para quem começa a trabalhar com ferramentas de *MySQL* esta é bastante conhecida a toda a comunidade que trabalha com bases de dados desse tipo e dada a sua complexidade e grandeza consideráveis é um exemplo perfeito para aplicarmos os ensinamentos adquiridos.

1.3 Motivação e Objetivos

Este projeto tem como objetivo mostrar-nos e percebermos as principais diferenças entre três modelos de sistema de base de dados bastante conhecidos e usados em situações reais.

Com isto poderemos consolidar os conhecimentos obtidos nas aulas e ao mesmo tempo termos contacto com um possível cenário real de construção e implementação de um novo sistema de base de dados a partir de um sistema já existente.

1.4 Estrutura do Relatório

Ao longo deste relatório vamos falar de cada um dos modelos assim como a base de dados base para executar todo o processo.

Começaremos por falar da base de dados *Sakila* e como está implementada em *MySQL*. Depois iremos abordar para cada novo sistema de base de dados o novo modelo de dados, assim como os processos de migração e exportação dos dados, como executar os pedidos ao sistema e os resultados obtidos a esses mesmo pedidos.

2 Sakila

De forma a facilitar o processo de migração da base de dados *Sakila* para os modelos pretendidos, começamos por realizar uma análise detalhada. Assim sendo, de seguida é possível observar todas as tabelas existentes no modelo fornecido, bem como uma breve descrição para perceber a sua função:

Tabela	Descrição
Ator	Lista a informação de todos os atores. Encontra-se ligada à tabela *film* através da tabela *film_actor*
Address	Contém informações de endereços para clientes, funcionários e lojas.
Category	Lista as categorias que podem ser atribuídas a um filme. Está ligada a tabela *film* através da tabela *film_actor*
City	Contém uma lista de cidades
Country	Possui uma lista de países
Customer	Dispõe uma lista de todos os clientes
Film	A tabela dos filmes tem uma lista de todos os filmes que podem existir em stock nas lojas. As que existem de facto estão na tabela inventário.
Film_actor	Esta tabela é utilizada para suportar o relacionamento de muitos para muitos existente entre as tabelas *film* e *actor*. Para cada ator em um determinado filme haverá uma linha neste tabela
Film_category	É usada para apoiar um relacionamento de muitos para muitos entre as tabelas *film* e *category*. Assim, para cada categoria associada a um *film* existirá uma linha nesta tabela
Film_text	Utilizada para guardar a descrição associada ao *film*. Esta é mantida em coerência com a tabela *film* através de triggers
Inventory	Contém um registo para cada *film* existente numa determinada loja
Language	Lista os possíveis idiomas que os filmes podem ter
Payment	Regista cada pagamento feito por um cliente, com informações como o valor e o aluguer em questão.
Rental	Contém uma linha para cada aluguer de um item do inventário. Esta tem informações sobre quem alugou quando o fez e a quando devolveu
Staff	Lista as informações relativas a todos os trabalhadores

Store	Lista todas as lojas do sistema. Todo o stock é atribuído a lojas específicas.

Tabela 1 - Tabelas Sakila

Uma vez estudadas as tabelas, foi necessário verificar as relações estabelecidas de modo a perceber como toda a informação se encontra associada bem como identificar todas as chaves primárias e estrangeiras contida nas mesmas. Este é uma etapa fundamental já que para conseguirmos elaborar o esquema de dados para os novos sistemas é essencial entender como tudo se procede no *MySQL*. Deste modo, de seguida ilustramos o modelo lógico apenas como as chaves primárias e secundárias de cada tabela para que seja mais intuitivo e percetível.

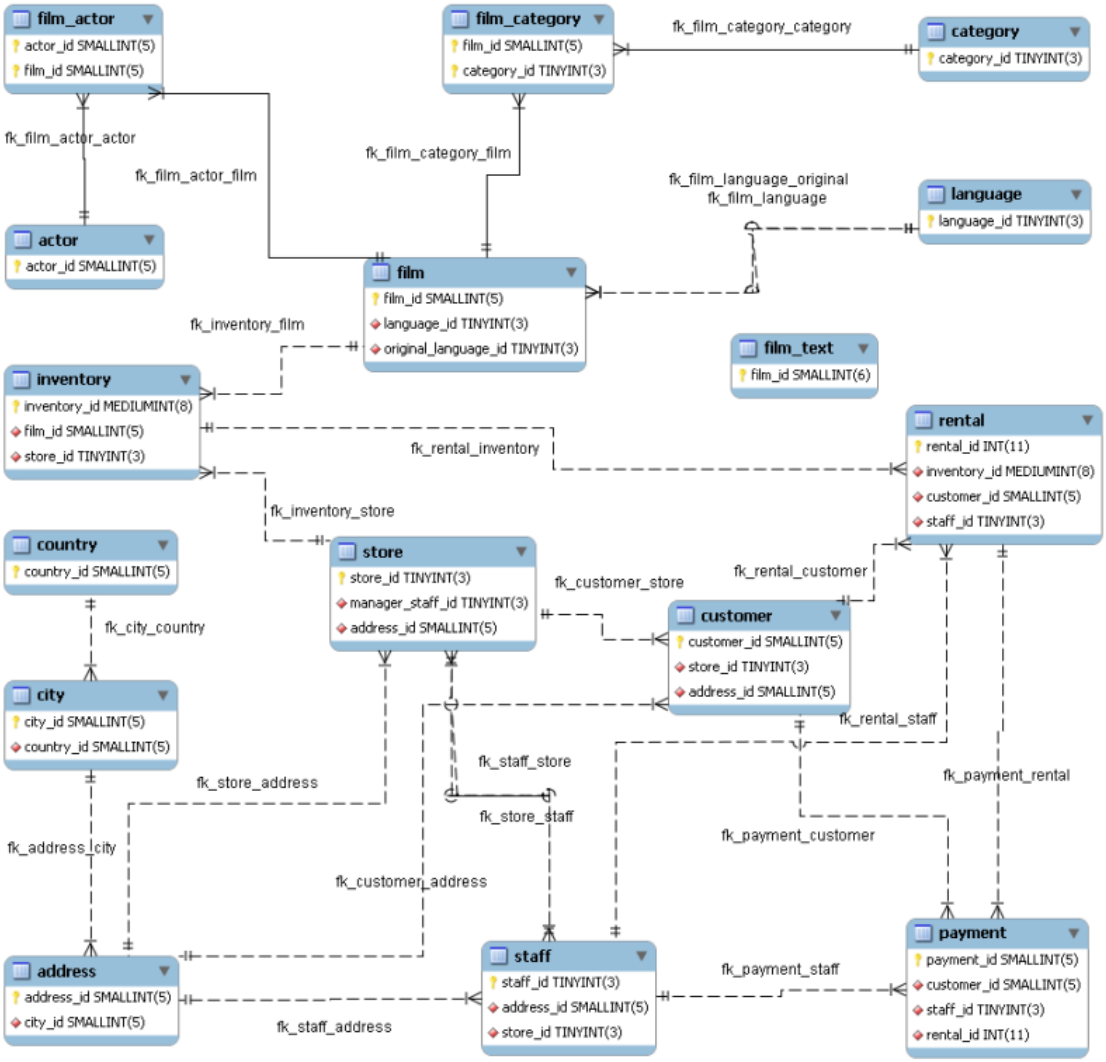


Figura 1 - Modelo lógico Sakila

3 Queries

De modo a comparar os resultados obtidos nas implementações da Base de Dados *Sakila* em *Oracle*, *Mongo* e *Neo4j* bem como testar as mesmas, e desta forma demonstrar a operacionalidade dos sistemas implementado, decidimos definir 7 queries que são apresentadas de seguida. Após o processo de definição do modelo e migração da base de dados é expectável que os resultados obtidos das queries no modelo relacional e nos dois não relacionais sejam iguais aos obtidos em *MySQL*.

1. Top 5 dos filmes mais alugados
2. Top 3 das lojas que mais faturam
3. Top 3 dos alugueres de filmes mais longos
4. Top 3 dos filmes que tiveram alugados por mais tempo
5. Top 5 das categorias mais alugadas
6. Top 3 dos atores que participou em mais filmes
7. Top 5 dos distritos com mais clientes
8. Top 5 dos funcionários que alugaram mais filmes

4 Base de Dados Relacional - Oracle

Tal como o *MySQL* a *Oracle* também é um Sistema de Gestão de Base de Dados relacional, pelo que o esquema de dados utilizado foi o mesmo. Contudo, verificamos que os tipos de dados existentes variam e como tal foi necessário modificar os que não eram reconhecidos pela *Oracle*. Para além disso, ainda foi preciso ter em atenção o processo de migração dos dados, já que alguns tipos foram alterados e como tal não vão corresponder. Assim sendo, de seguida iremos explicar todo o processo bem como a execução das queries propostas anteriormente.

4.1 Modelo

Como optamos por manter o esquema geral da base de dados e apenas alterar as funcionalidades que não se encontram disponíveis na Oracle, na conversão das tabelas o mais complicado foi perceber quais não existiam e para que tipo deveriam migrar. Após uma análise verificamos que essencialmente existiam quatro problemas diferentes. Primeiro deparamo-nos com a incompatibilidade de dados, por exemplo o *BOOLEAN* não é um tipo definido em *Oracle* pelo que foi necessário substituí-lo por *NUMBER(1,0)*. De forma a realizar esta conversão do modo mais correto possível recorremos a um guia de conversão. Em *MySQL* no ficheiro de criação das tabelas era introduzida uma regra que basicamente atualizava o *timestamp* da tabela sempre que nesta fosse alterado algum dos seus campos. No entanto, verificamos que em *Oracle* não era permitido efetuar da mesma forma. A solução encontrada passou pela criação de um *trigger* para cada tabela que sempre que a mesma é atualizada altera o seu *timestamp* para a data atual como pretendido. Outra das diferenças encontradas foi o modo como é realizada a autoincrementação de uma variável. No *MySQL* utiliza-se o *AUTO_INCREMENT* enquanto que no *Oracle* se cria uma sequência indicando-lhe em que valor deve começar e quanto deve aumentar de cada vez. Finalmente constatamos que a sintaxe de criação de *primary keys* em *Oracle* não é igual, pelo que mais uma vez tivemos de adaptar.

Assim sendo, de seguida apresentamos a título de exemplo a criação da tabela *Country* em ambos os SGBD, onde é possível verificar as principais alterações enumeradas anteriormente inclusive o *trigger* para atualizar o *timestamp*.

- **Criação da tabela *Country MySQL***

```
CREATE TABLE country (
  country_id SMALLINT UNSIGNED NOT NULL
AUTO_INCREMENT,
  country VARCHAR(50) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT
CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (country_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- **Criação da tabela *Country Oracle***

```
CREATE TABLE country (
```

```
country_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH    1 INCREMENT BY 1),
country VARCHAR(50) NOT NULL,
last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT pk_country PRIMARY KEY (country_id));
```

4.2 Migração

De forma a realizar a migração dos dados optamos por utilizar a linguagem *Python*. Assim sendo recorremos ao módulo de extensão cx_Oracle do *Python* para estabelecer a conexão com a base de dados Oracle. Desta forma, apenas tivemos de percorrer o ficheiro de povoamento da *Sakila* e inserir os dados na base de dados *Oracle*. Contudo foi necessário proceder a algumas alterações devido à mudança efetuada em alguns tipos de dados.

De forma a facilitar este processo decidimos inserir as chaves estrangeiras apenas depois do povoamento, já que verificamos que existem bastantes relações entre tabelas. Esta alteração apenas foi possível porque é nos assegurado que os dados se encontram consistentes, já que provêm do *MySQL*. Contudo criamos um ficheiro com todas as chaves estrangeiras, de forma a manter todas as verificações necessárias.

Deste modo após efetuarmos o povoamento da base de dados *Oracle*, executamos o script que contém todas as *Constraints*. Assim, na eventualidade dos dados não serem consistentes é possível detetar. É importante referir que tanto o ficheiro de povoamento como o de inserção das chaves estrangeiras se encontram em anexo.

4.3 Queries

Nesta secção iremos apresentar a implementação de todas as queries planeadas anteriormente em *MySQL*.

1. **Top 5 dos filmes mais alugados**

```
select f.title as film, count(r.rental_id) as
num_films
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.title
order by num_films DESC
fetch first 5 rows only;
```

2. **Top 3 das lojas que mais faturam**

```
select s.store_id as Store, sum(p.amount) as Faturado
from store s
inner join staff st on (st.store_id = s.store_id)
inner join payment p on (st.staff_id = p.staff_id)
group by s.store_id
```

```
order by Faturado DESC
fetch first 3 rows only;
```

1. **Top 3 dos alugueres de filmes mais longos**

```
select f.title as Title, sum(r.return_date-
r.rental_date) as TotalDays
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.title
order by TotalDays DESC
fetch first 3 rows only;
```

3. **Top 5 das categorias mais alugadas**

```
select count(r.rental_id) as num_films, c.name as
category
from category c
inner join film_category fc on (fc.category_id =
c.category_id)
inner join film f on (f.film_id = fc.film_id)
INNER JOIN inventory i ON (i.film_id=f.film_id)
INNER JOIN rental r ON (r.inventory_id=i.inventory_id)
group by c.name
order by num_films DESC
fetch first 5 rows only;
```

4. **Top 3 dos atores que participaram em mais filmes**

```
select a.first_name, a.last_name, count(fa.film_id) as
num_films
from actor a
inner join film_actor fa on (fa.actor_id = a.actor_id)
group by a.actor_id,a.first_name, a.last_name
order by num_films DESC
fetch first 3 rows only;
```

5. **Top 5 dos distritos com mais clientes**

```
select a.district as District, count(c.customer_id) as
Total
from address a
inner join customer c on (c.address_id = a.address_id)
group by a.district
order by Total DESC
fetch first 5 rows only;
```

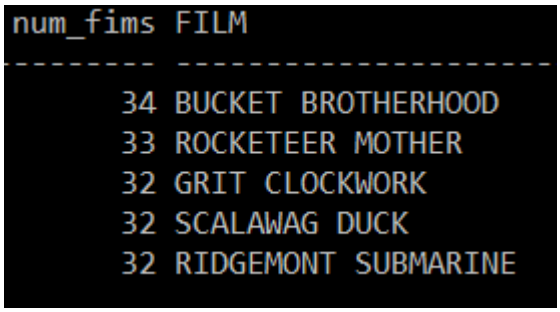
6. **Top 5 dos funcionários que alugaram mais filmes**

```
select s.staff_id as ID, s.first_name as FirstName,
s.last_name as LastName, count(p.payment_id) as Total
from staff s
inner join payment p on (p.staff_id = s.staff_id)
inner join rental r on (r.rental_id = p.rental_id)
group by s.staff_id, s.first_name, s.last_name
order by Total DESC
fetch first 5 rows only;
```

4.4 Resultados

De seguida iremos expor os resultados obtidos de algumas das queries definidas previamente, nomeadamente a 1, 4 e 5.

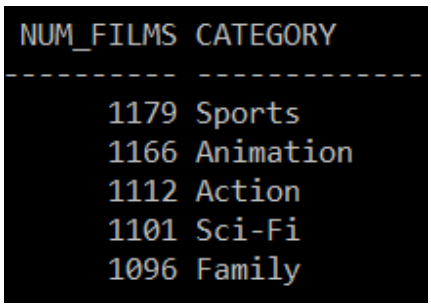
1. **Top 5 dos filmes mais alugados**



num_fims	FILM
34	BUCKET BROTHERHOOD
33	ROCKETEER MOTHER
32	GRIT CLOCKWORK
32	SCALAWAG DUCK
32	RIDGEMONT SUBMARINE

Figura 2 - Resutado da query 1 em *MySQL*

4. **Top 5 das categorias mais alugadas**



NUM_FILMS	CATEGORY
1179	Sports
1166	Animation
1112	Action
1101	Sci-Fi
1096	Family

Figura 3 - Resultado da query 4 em *MySQL*

5. **Top 3 dos atores que participaram em mais filmes**

FIRST_NAME	LAST_NAME	num_fims
GINA	DEGENERES	42
WALTER	TORN	41
MARY	KEITEL	40

Figura 4 - Resultado da query 5 em *MySQL*

5 Base de Dados Orientada a Grafos - Neo4j

5.1 Modelo

De modo a obtermos o modelo apropriado para o *Neo4j* da Base de Dados *Sakila* tornou-se necessário realizarmos algumas alterações no modelo da mesma, entre as quais:

- **Retirar a Identificação –** No Modelo Relacional todas as entidades possuem uma chave estrangeira, que se caracterizam por um id. Uma vez que no *Neo4j*, por ser uma base de dados orientada a grafos, não são necessários estes mesmos identificadores para criarmos as entidades e os correspondentes relacionamentos decidimos remover os mesmos. Contudo, constatamos que a *Store* não possuía mais nenhum elemento que a caracteriza-se pelo que esta é a única entidade com respetivo identificador.

- **Retirar tabelas desnecessárias –** Ao longo deste processo tornou-se percetível que existiam algumas tabelas que não eram necessárias serem convertidas, sendo estas as correspondentes aos relacionamentos muitos para muitos, ou seja, as tabelas *film_category*, *film_actor* e *inventory*. No entanto, nesta última existia um indentificador como chave estrangeira, que permitia obtermos o *stock* de filmes existente em cada *store*. Para armazenar este valor optamos por criar uma propriedade no relacionamento entre a *Store* e o *Film* com o *stock* correspondente àquela loja. Por último, optamos por não converter a tabela *film_text*, tendo em conta que consideramos que esta não acrescentava informação útil nem necessária às seguintes fases do projeto.

Terminada a fase de tomada de decisão, podemos então esboçar o esquema que pretendemos para o nosso modelo, que é apresentado na Figura 5 - Esquema Modelo Neo4j.

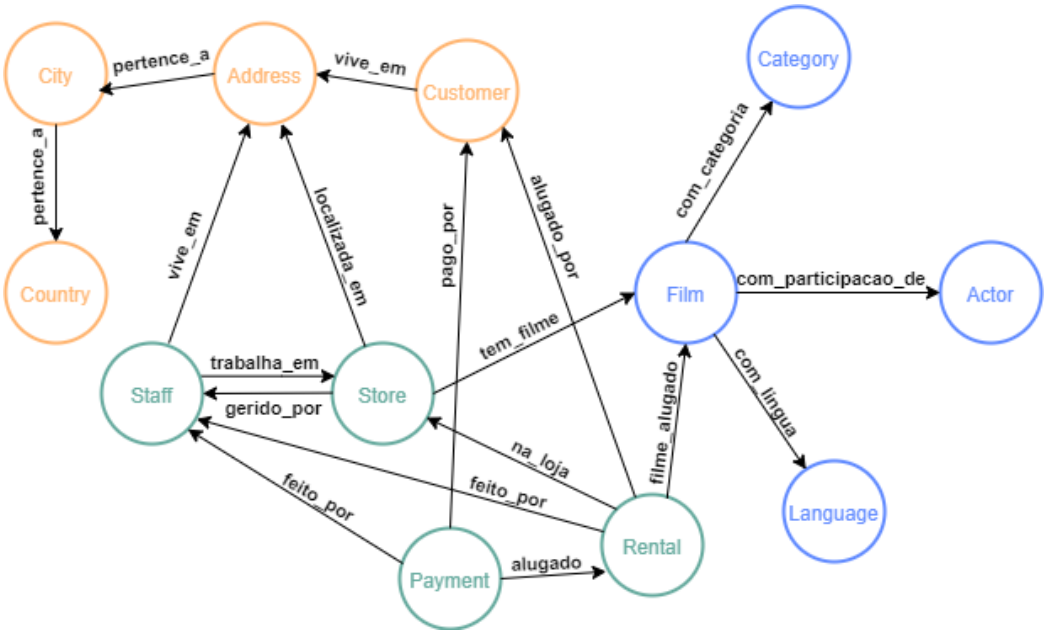


Figura 5 - Esquema Modelo Neo4j

5.2 Migração

Nesta fase passamos para a exportação dos dados da *Sakila* para ficheiros *CSV* de modo a importá-los posteriormente para o *Neo4j*. Tendo em conta as decisões indicadas anteriormente criamos um ficheiro *sql*, que se encontra em anexo, para a extração destes mesmos dados. Basicamente o processo que adotamos para a migração foi o seguinte:

- **Exportação –** nesta fase recorrendo ao *sql* extraímos todos os dados pretendidos tendo em conta as decisões tomadas. Tal como foi indicado anteriormente, decidimos remover os identificadores uma vez que não eram necessários, contudo devido à existência de algumas entidades, como é o caso do *Payment* e da *Rental*, que não tinham nenhum atributo que as pudessem identificar houve a necessidade de migrar estas com os seus id's para a posterior criação dos relacionamentos. Para além disso, uma vez que constatamos que existia um ator repetido optamos também por migrar o seu id. A título exemplificativo apresentamos de seguida um excerto referente à exportação do *Country* e da *City.*

```
select * from country
into outfile '/var/lib/mysql-files/country.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select city, country from city
inner join country on city.country_id =
country.country_id
into outfile '/var/lib/mysql-files/city.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';
```

- **Importação** – realizada a exportação podemos agora importar os dados para *Neo4j*. Nesta fase, tivemos o cuidado de eliminar os identificadores das entidades *Payment*, *Rental* e *Actor,* que foram migrados tal com explicado anteriormente, após a criação dos relacionamentos. De seguida, é exemplificado a importação para os mesmos casos, em que para o *Country* são criados os nós e de seguida um índice no nome do mesmo. Enquanto que para a *City* para além da criação dos nós é criado o relacionamento entre o anterior e esta, sendo por fim criado o índice no nome da mesma.

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

create index on :Country(country);

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

create index on :City(city);
```

5.3 Queries

2. **Top 5 dos filmes mais alugados**

```
match(f:Film)<-[a:filme_alugado]-(r:Rental)
return f, count(a) as num_films
order by num_films desc
limit 5
```

3. **Top 3 das lojas que mais faturam**

```
match(s:Store)<-[:trabalha_em]-(st:Staff)<-
[:feito_por]-(p:Payment)
return s, sum(p.amount) as num_stores
order by num_stores desc
limit 3
```

4. **Top 3 dos alugueres de filmes mais longos**

```
match(r:Rental)-[:filme_alugado]->(f:Film)
where exists(r.return_date)
return f.title,
avg(duration.between(datetime(r.rental_date),
datetime(r.return_date))) as dur_avg
order by dur_avg desc
limit 3
```

5. **Top 5 das categorias mais alugadas**

```
match(r:Rental)-[:filme_alugado]->(f:Film)-
[:com_categoria]->(c:Category)
return count(f) as num_films,c
order by num_films desc
limit 5
```

6. **Top 3 dos atores que participaram em mais filmes**

```
match(f:Film)-[:com_participacao_de]->(a:Actor)
return count(f) as num_films,a
order by num_films desc
limit 3
```

7. **Top 5 dos distritos com mais clientes**

```
MATCH (c:Customer)-[r:vive_em]->(a:Address)
return count(c) as num_customer, a.district
order by num_customer desc
limit 5
```

8. **Top 5 dos funcionários que alugaram mais filmes**

```
match(r:Rental)<-[:alugado]-(p:Payment)-[:feito_por]-
>(s:Staff)
return count(r) as num_films, s
order by num_films desc
limit 5
```

5.4 Resultados

Através das queries definidas pelo grupo tornou-se assim possível obter várias informações relevantes da Base de Dados *Sakila*. De seguida são apresentados os resultados obtidos em algumas das apresentadas anteriormente.

1. **Top 5 dos filmes mais alugados**

Através desta query é possível obter os 5 filmes mais alugados, sendo possível observar neste caso em específico o nome dos filmes bem como o número de vezes que foi alugado. Nesta optamos por devolver os filmes, ou seja, todas as informações referentes aos mesmos, contudo no resultado seguinte apenas apresentamos o título, dado que se tornaria muita informação para ser apresentada.

film	num_films
"BUCKET BROTHERHOOD"	34
"ROCKETEER MOTHER"	33
"GRIT CLOCKWORK"	32
"FORWARD TEMPLE"	32
"RIDGEMONT SUBMARINE"	32

Figura 6 - Resultado da query 1 em *Neo4j*

4. **Top 5 das categorias mais alugadas**

num_films	category
1179	"Sports"
1166	"Animation"
1112	"Action"
1101	"Sci-Fi"
1096	"Family"

Figura 7 - Resultado da query 4 em *Neo4j*

5. **Top 3 dos atores que participaram em mais filmes**

num_films	a
42	{ "last_name": "DEGENERES", "first_name": "GINA" }
41	{ "last_name": "TORN", "first_name": "WALTER" }
40	{ "last_name": "KEITEL", "first_name": "MARY" }

Figura 8 – Resultado da query 5 em *Neo4j*

5.5 Comparação com o Modelo Relacional

Como era expectável a migração da Base de Dados Relacional *Sakila* para *Neo4j* permitiu a implementação de um modelo bastante mais simples, em parte porque os relacionamentos de muitos para muitos podem ser tratados de uma forma eficiente, e ainda foi possível eliminar as chaves estrangeiras. Para além disso, a implementação das queries é bastante mais percetível e rápida sendo que a performance é melhorada

uma vez que em *MySQL* as mesmas teriam uma resolução mais complexa com vários *JOINs.*

6 Base de Dados Documental - MongoDB

Tal como o nome indica uma base de dados documental tem como base os documentos. Neste tipo de base de dados temos de realmente perceber o que é necessário guardar e como as agregar. Além disso queremos evitar ao máximo os relacionamentos entre as várias entidades de forma a evitar muitos cruzamentos de dados. Isto resultará em, por exemplo, repetição de dados, porém essa repetição não é crítica pois não implica uma redução significante da performance.

6.1 Modelo

Como foi introduzido anteriormente, o modelo deste tipo de base de dados é bastante diferente ao modelo relacional.

Para realizarmos qualquer alteração foi primeiro necessário estudar e compreender como funciona o modelo relacional e perceber para que, de facto, é usada e que tipo de dados é que guardo. Esta primeira análise foi fundamental para podermos executar o processo de adaptação de um modelo de base de dados relacional para um modelo documental, de forma a poder manter a principal função da mesma.

Dito isto e após a análise pudemos identificar as duas principais entidades como os alugueres (*rentals)* e os clientes (*customers*). *Rentals* diz respeito a qualquer aluguer feito por um cliente que alugou um filme. *Customers* diz respeito a qualquer cliente da entidade que disponibiliza os filmes para alugar. Cada uma destas entidades será uma coleção na nossa base de dados. Existirá ainda outra coleção que corresponderá aos filmes. Visto que os filmes são a base dos alugueres, achamos por bem guardar toda a coleção de filmes existentes.

A coleção *rentals* será uma coleção com alguma informação. Para poder responder a alguns pedidos da entidade que usufrui da base de dados, foi necessário agregar informação de várias tabelas nas entradas de *rentals*. Esta será formada por:

- _id : ObjectId
- id: Inteiro
- customers_id: Inteiro
- rental_date: Data
- return_date: Data
- rental_duration: Timestamp
- payment_Value: Double
- staff: {
    - staff_id: String
    - name: String
    - email: String
    - store_id: String
    - store_address: String
    - store_city: String
    - store_district: String
    - store_country: String

  }
- film: {
    - film_id: String
    - title: String
    - description: String
    - special_features: String
    - language: String
    - category: String,
    - actors: [String]

  }

Faremos agora uma breve análise aos dados que guardados nesta entrada. Optamos por manter os id's atribuídos no modelo relacional no próprio *rental*, assim como para o *staff*, a *store*, e para o *film*. Foi necessário guardar para a *store*, visto que esta não tinha um nome que a identificasse e por isso optamos por guardar para as restantes entidades relevantes. Depois sendo um *rental* achamos que faria sentido guardar a data do aluguer e a data de entrega. Para facilitar futuras operações adicionamos a duração da mesma. Para respondermos a algumas querys teríamos de guardar informação sobre a loja onde um *rental* foi realizado e o funcionário que atendeu. Por isso decidimos guardar tudo isso dentro do *staff.* Aí guardamos as informações do funcionário (id, nome, email) e da loja (id, morada, cidade, distrito, país). Quanto ao filme que foi alugado decidimos guardar o seu id, o título, descrição e conteúdos extra, a linguagem, a categoria na qual se insere e o nome e todos os atores que participam no filme. Por uma questão de

conveniência guardamos também o id do cliente, sendo esta a única ligação com a coleção *customers*.

A coleção *customers* é uma coleção com um conjunto de informação mais pequeno, visto que se foca apenas no cliente em si. Assim temos:

- _id: ObjectId
- customer_id: String
- name: String
- email: String
- address: {
    - address: String
    - district: String
    - postal_code: String
    - city: String
    - country: String

  }

Quanto aos dados guardados temos o id do cliente (para podermos interligar com o/os *rental/rentals* associados). Além do id guardamos o nome, o email e a morada do cliente. Esta morada é composta através da rua, cidade, distrito, país e código postal.

A coleção filmes será uma coleção com todas as informações de um dado filme:

- _id: ObjectId
- film_id: String
- title: String
- description: String
- release_year: String
- rental_duration: String
- rental_rate: String
- length: String
- replacement_cost: String
- rating: String
- special_features: String
- last_update: String
- language: String
- category: String
- actors: [String]

6.2 Migração de dados

Quanto à migração dos dados podemos identificar duas fases distintas:

- **Exportação -** diz respeito ao processo de extrair os dados do modelo relacional e adaptar para o formato pretendido. Também esta fase poderá ser dividida em duas transformações.
  A primeira transformação baseia-se na extração de todos os dados possíveis do modelo relacional *MySQL*. Copiamos todas as informações para vários ficheiros *SQL*, onde cada um dizia respeito a uma tabela e continha o script de criação da tabela e o povoamento da mesma. Após isto, com a ajuda de uma ferramenta online, convertemos cada ficheiro num ficheiro *JSON*.
  A segunda transformação trata-se de moldar os ficheiros *JSON* para guardar as informações pretendidas e para poder suportar a base de dados documental. Para tal fizemos uso de 4 scripts diferentes desenvolvidos em *Python*. Cada script irá moldar ficheiros *JSON* para poder guardar *Addresses, Films, Rentals* e por fim *Customers*. De realçar que destes quatro novos ficheiros *JSON*, os principais serão o de *Rentals*, o dos *Customers* e o dos *Films*, visto que estes são o suporte para cada coleção. Os restantes novos ficheiros são usados como ferramenta para estes dois pois já se encontram no formato que queremos.

- **Importação -** diz respeito à fase de importar os dados dos ficheiros *JSON* para a base de dados em *Mongo*. Para tal existe ainda uma pequena alteração aos dois ficheiros. Teremos de retirar as primeiras duas linhas de cada um, assim como a última, para o documento começar e terminar com **[** e **]** respetivamente. Após isto será necessário introduzir na linha de comandos:

  mongoimport -d Sakila -c rentals --jsonArray
  ficheiros_JSON/newJson/rental.json
  e
  mongoimport -d Sakila -c customers --jsonArray
  ficheiros_JSON/newJson/customers.json
  e
  mongoimport -d Sakila -c films --jsonArray
  ficheiros_JSON/newJson/film.json

6.3 Querys

Nesta secção vamos apresentar a implementação de todas as querys planeadas:

1 Top 5 dos filmes mais alugados

Nesta query temos de percorrer a coleção **rental** e agrupamos cada entrada através do atributo **title** do object **film**.

```
db.rentals.aggregate([{$group: {_id: "$film.title",
numRentals:{$sum:1}}},{$sort: {numRentals:-
1}},{$limit:5}])
```

2 Top 3 das lojas que mais faturam

Nesta query temos de percorrer a coleção **rental** e agrupar cada entrada pelo atributo **store_id** do object **staff**. Após esse agrupamento temos de somar cada pagamento recebido através do atributo **payment_value**.

```
db.rentals.aggregate([{$group: {_id:
"$staff.store_id",
store_profit:{$sum:"$payment_value"}}},{$sort:
{store_profit:-1}},{$limit:3}])
```

3 Top 3 filmes dos alugueres mais longos

Nesta query temos, mais uma vez, de percorrer a coleção **rental** e agrupar cada entrada pelo atributo **id** ou **title** do *object* **film**. Com este agrupamento poderemos ordenar o mesmo pelo atributo **rental_duration** e obter assim o top 3.

```
db.rentals.find({},{"_id":0,"film.title":1,"rental_dur
ation":1,"staff.store_id":1}).sort({rental_duration:-
1}).limit(3)
```

4 Top 5 das categorias mais alugadas

```
db.rentals.aggregate([{$group: {_id: "$film.category",
numRentals:{$sum:1}}},{$sort: {numRentals:-
1}},{$limit:5}])
```

5 Top 3 dos atores que participaram em mais filmes

```
db.films.aggregate([{$unwind: "$actors"},{$group:
{_id: "$actors", numFilmes: {$sum:1}}},{$sort:
{numFilmes:-1}},{$limit:3}])
```

6 Top 5 dos distritos com mais clientes

```
db.customers.aggregate([{$group: {_id:
"$address.district", numCustomers:{$sum:1}}},{$sort:
{numCustomers:-1}},{$limit:5}])
```

7 Top 5 dos funcionários que alugaram mais filmes

```
db.rentals.aggregate([{$group: {_id: "$staff.name",
number_rentals:{$sum:1}}},{$sort: {number_rentals:-
1}},{$limit:5}])
```

6.4 Resultados

Vamos agora apresentar os resultados para as queries 1, 4 e 5.

1. **Top 5 dos filmes mais alugados**

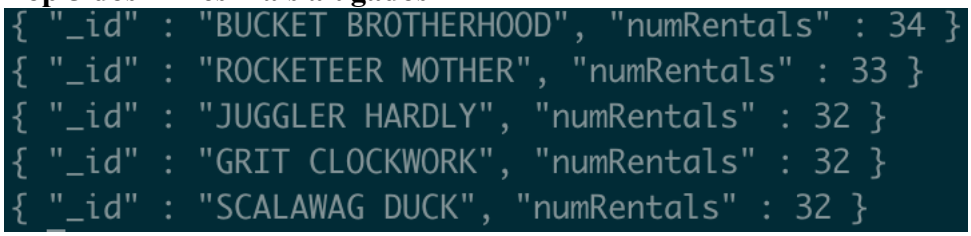


Figura 9 - Resultado da query 1 em MongoDB

De realçar que estes valores podem variar com os outros sistemas devido ao grande número de filmes que foram alugados 32 vezes.

4. **Top 5 das categorias mais alugadas**

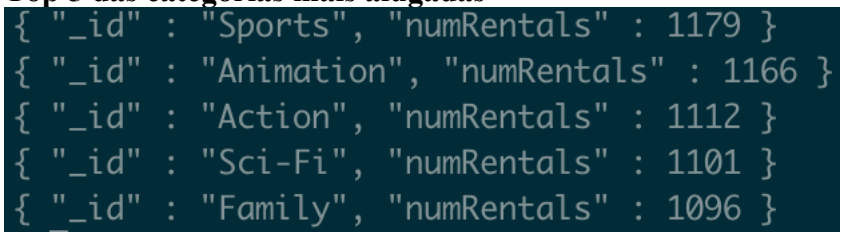


Figura 10 - Resultado da query 4 em MongoDB

5. **Top 3 dos atores que participaram em mais filmes**

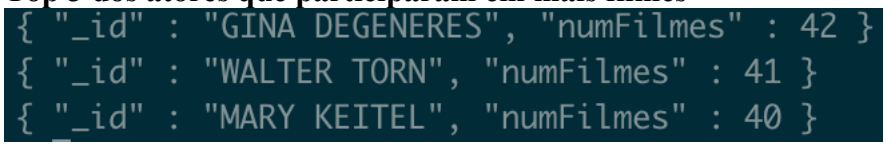


Figura 11 - Resultado da query 5 em MongoDB

6.5 Comparação com o Modelo Relacional

Tal como seria de esperar esta migração de dados tornou o modelo bastante mais simples e intuitivo. Com este modelo documental perdemos a dependência dos dados que, apesar de haver referência entre a coleção *rentals* e a coleção *customers*, nenhuma delas depende da outra. Isto leva a que se houver uma alteração de dados na coleção que corresponde aos clientes, a coleção de alugueres continua a poder responder a qualquer questão. Outro aspeto positivo é o facto de assim não haver o rigor no formato dos dados das coleções. Além disso, todo o processo de procura e inserção de informação é bastante mais rápida.

7 Conclusões e Trabalho Futuro

Com a realização deste trabalho foi possível ficar a conhecer mais detalhadamente cada um do SGBD implementados. Ao longo do seu desenvolvimento verificamos que cada um possui as suas características próprias que o torna mais adequado para determinados contextos. Desta forma, no futuro ser-nos-á mais fácil escolher o sistema indicado já que conhecemos melhor cada um.

Assim como o *MySQL* a *Oracle* também é um SGBD relacional, pelo que a migração foi bastante simples sendo necessário apenas algumas alterações principalmente relacionadas com a sintaxe de ambos os modelos. Tendo a *Sakila* já apresenta uma complexidade considerável constatamos na resolução das queries que é necessário realizar vários *Join* o que não é muito bom a nível de desempenho.

No caso do *Neo4j* é notório que a criação do modelo é bastante flexível e a sua modificação também, sendo a sua criação foi bastante simples bem como as queries criadas que foram bastante menos complexas e com melhor desempenho.

Quanto à base de dados documental, *MongoDB*, vimos que o processo de migração e exportação é bastante simples e que facilmente foi realizado. Também foi simples o processo de construção de queries e foi notável o melhoramento do desempenho e legibilidade do processo.

Em última instância, concluímos que o trabalho foi realizado com sucesso, uma vez que conseguimos migrar a *Sakila* para todos os modelos de base de dados propostos.

8 Referências

1. Ian Robinson, Jim Webber and Emil Eifrem 2015. Graph Databases New Opportunities for Connected Data. Second Edition / O'Reilly Media.

9 Lista de Siglas e Acrónimos

BD Base de Dados

SGBD Sistema de Gestão de Base de Dados

SQL Structured Query Language

SBD Sistema de Base de Dados

SBDR Sistema de Base de Dados Relacional

NoSQL Not only SQL

10 Anexos

1. **Criação das tabelas MySQL**
2. **Foreign keys MySQL**
3. **Povoamento MySQl**
4. **Importação dos Dados Neo4j**
5. **Exportação dos Dados Neo4j**

I. Anexo 1 – Criação das tabelas 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 pk_actor PRIMARY KEY (actor_id));

CREATE INDEX idx_actor_last_name ON actor (last_name);

CREATE OR REPLACE TRIGGER actor_timestamp_update_trigger
    BEFORE UPDATE ON actor
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE country (
  country_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1),
  country VARCHAR(50) NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_country PRIMARY KEY  (country_id));

CREATE OR REPLACE TRIGGER country_timestamp_update_trigger
    BEFORE UPDATE ON country
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE city (
  city_id SMALLINT GENERATED BY DEFAULT AS IDENTITY (START
WITH 1 INCREMENT BY 1),
  city VARCHAR(50) NOT NULL,
  country_id SMALLINT NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_city PRIMARY KEY  (city_id),
  CHECK(country_id>0));

CREATE INDEX idx_fk_city_country_id ON city (country_id);

CREATE OR REPLACE TRIGGER city_timestamp_update_trigger
    BEFORE UPDATE ON city
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
```

```
    END;
/

CREATE TABLE address (
  address_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1),
  address VARCHAR(50) NOT NULL,
  address2 VARCHAR(50) DEFAULT NULL,
  district VARCHAR(20) NOT NULL,
  city_id SMALLINT NOT NULL,
  postal_code VARCHAR(10) DEFAULT NULL,
  phone VARCHAR(20) NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_address PRIMARY KEY  (address_id),
  CHECK(city_id>0));

CREATE INDEX idx_fk_address_city_id ON address (city_id);

CREATE OR REPLACE TRIGGER address_timestamp_update_trigger
    BEFORE UPDATE ON address
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE category (
  category_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1),
  name VARCHAR(25) NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_category PRIMARY KEY  (category_id));

CREATE OR REPLACE TRIGGER category_timestamp_update_trigger
    BEFORE UPDATE ON category
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE staff (
  staff_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,
  address_id SMALLINT NOT NULL,
  picture BLOB DEFAULT NULL,
  email VARCHAR(50) DEFAULT NULL,
  store_id SMALLINT NOT NULL,
  active NUMBER(1,0) DEFAULT 1,
```

```
  username VARCHAR(16) NOT NULL,
  password varchar(45) DEFAULT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_staff PRIMARY KEY  (staff_id),
  CHECK(address_id > 0),
  CHECK(store_id > 0));

CREATE INDEX idx_fk_staff_store_id ON staff (store_id);

CREATE INDEX idx_fk_staff_address_id ON staff (address_id);

CREATE OR REPLACE TRIGGER staff_timestamp_update_trigger
    BEFORE UPDATE ON staff
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE store (
  store_id SMALLINT GENERATED BY DEFAULT AS IDENTITY (START
WITH 1 INCREMENT BY 1),
  manager_staff_id SMALLINT NOT NULL,
  address_id SMALLINT NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_store PRIMARY KEY  (store_id),
  CHECK(manager_staff_id > 0),
  CHECK(address_id > 0));

CREATE INDEX idx_fk_store_address_id ON store (address_id);

CREATE  INDEX idx_store_manager_staff_id ON
store(manager_staff_id);

CREATE OR REPLACE TRIGGER store_timestamp_update_trigger
    BEFORE UPDATE ON store
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE customer (
  customer_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1),
  store_id SMALLINT NOT NULL,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  email VARCHAR(50) DEFAULT NULL,
  address_id SMALLINT NOT NULL,
  active NUMBER(1,0) DEFAULT 1,
```

```
  create_date DATE NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_customer PRIMARY KEY  (customer_id),
  CHECK(address_id > 0),
  CHECK(store_id > 0));

CREATE INDEX idx_fk_customer_store_id ON customer
(store_id);

CREATE INDEX idx_fk_customer_address_id ON customer
(address_id);

CREATE INDEX idx_fk_customer_last_name ON customer
(last_name);

CREATE OR REPLACE TRIGGER customer_timestamp_update_trigger
    BEFORE UPDATE ON customer
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE language (
  language_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1),
  name CHAR(20) NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_language PRIMARY KEY (language_id),
  CHECK(language_id>0));

CREATE OR REPLACE TRIGGER language_timestamp_update_trigger
    BEFORE UPDATE ON language
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE film (
  film_id SMALLINT GENERATED BY DEFAULT AS IDENTITY (START
WITH 1 INCREMENT BY 1),
  title VARCHAR(255) NOT NULL,
  description CLOB DEFAULT NULL,
  release_year NUMBER DEFAULT NULL,
  language_id SMALLINT NOT NULL,
  original_language_id SMALLINT DEFAULT NULL,
  rental_duration SMALLINT DEFAULT 3 NOT NULL,
  rental_rate DECIMAL(4,2) DEFAULT 4.99 NOT NULL,
  length SMALLINT DEFAULT NULL,
  replacement_cost DECIMAL(5,2) DEFAULT 19.99 NOT NULL,
```

```
  rating VARCHAR(10) DEFAULT 'G',
  special_features VARCHAR(64) DEFAULT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_film PRIMARY KEY  (film_id),
  CHECK(rating IN ('G','PG','PG-13','R','NC-17')),
  CHECK(language_id > 0),
  CHECK(original_language_id > 0),
  CHECK(rental_duration > 0),
  CHECK(length > 0));

CREATE INDEX idx_fk_film_title ON film (title);

CREATE INDEX idx_fk_film_language_id ON film (language_id);

CREATE INDEX idx_fk_film_original_language_id ON film
(original_language_id);

CREATE OR REPLACE TRIGGER film_timestamp_update_trigger
    BEFORE UPDATE ON film
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE film_actor (
  actor_id SMALLINT NOT NULL,
  film_id SMALLINT NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_film_actor PRIMARY KEY  (actor_id,film_id),
  CHECK(actor_id > 0),
  CHECK(film_id > 0));

CREATE INDEX idx_fk_film_actor_film ON film_actor
(film_id);

CREATE OR REPLACE TRIGGER
film_actor_timestamp_update_trigger
    BEFORE UPDATE ON film_actor
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE film_category (
  film_id SMALLINT NOT NULL,
  category_id SMALLINT NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_film_category PRIMARY KEY (film_id,
category_id),
```

```
  CHECK(film_id > 0),
  CHECK(category_id > 0));

CREATE OR REPLACE TRIGGER
film_category_timestamp_update_trigger
    BEFORE UPDATE ON film_category
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE film_text (
  film_id SMALLINT NOT NULL,
  title VARCHAR(255) NOT NULL,
  description CLOB,
  CONSTRAINT pk_film_text PRIMARY KEY  (film_id));

CREATE TABLE inventory (
  inventory_id  NUMBER(7,0) GENERATED BY DEFAULT AS
IDENTITY (START WITH 1 INCREMENT BY 1),
  film_id SMALLINT NOT NULL,
  store_id SMALLINT NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_inventory PRIMARY KEY  (inventory_id),
  CHECK(film_id > 0),
  CHECK(store_id > 0));

CREATE INDEX idx_fk_inventory_film_id ON inventory
(film_id);

CREATE INDEX idx_fk_inventory_store_id_film_id ON inventory
(store_id,film_id);

CREATE OR REPLACE TRIGGER
inventory_timestamp_update_trigger
    BEFORE UPDATE ON inventory
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE rental (
  rental_id INT GENERATED BY DEFAULT AS IDENTITY (START
WITH 1 INCREMENT BY 1),
  rental_date DATE NOT NULL,
  inventory_id NUMBER(7,0) NOT NULL,
  customer_id SMALLINT NOT NULL,
  return_date DATE DEFAULT NULL,
  staff_id SMALLINT NOT NULL,
```

```
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_rental PRIMARY KEY (rental_id),
  CHECK(inventory_id > 0),
  CHECK(customer_id > 0),
  CHECK(staff_id > 0));

CREATE UNIQUE INDEX idx_rental_date_inventory_customer ON
rental (rental_date,inventory_id,customer_id);

CREATE INDEX idx_fk_rental_inventory_id ON rental
(inventory_id);

CREATE INDEX idx_fk_rental_customer_id ON rental
(customer_id);

CREATE INDEX idx_fk_rental_staff_id ON rental (staff_id);

CREATE OR REPLACE TRIGGER rental_timestamp_update_trigger
    BEFORE UPDATE ON rental
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/

CREATE TABLE payment (
  payment_id SMALLINT GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1),
  customer_id SMALLINT NOT NULL,
  staff_id SMALLINT NOT NULL,
  rental_id INT DEFAULT NULL,
  amount DECIMAL(5,2) NOT NULL,
  payment_date DATE NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_payment PRIMARY KEY  (payment_id),
  CHECK(customer_id > 0),
  CHECK(staff_id > 0));

CREATE INDEX idx_fk_payment_staff_id ON payment (staff_id);

CREATE INDEX idx_fk_payment_customer_id ON payment
(customer_id);

CREATE OR REPLACE TRIGGER payment_timestamp_update_trigger
    BEFORE UPDATE ON payment
    FOR EACH ROW
    BEGIN
        :new.last_update := current_timestamp;
    END;
/
```

II. Anexo 2 – Foreign keys MySQL

```
ALTER TABLE city
ADD CONSTRAINT fk_city_country FOREIGN KEY (country_id)
REFERENCES country(country_id);

ALTER TABLE address
ADD CONSTRAINT fk_address_city FOREIGN KEY (city_id)
REFERENCES city(city_id);

ALTER TABLE staff
ADD CONSTRAINT fk_staff_address FOREIGN KEY (address_id)
REFERENCES address (address_id);

ALTER TABLE store
ADD CONSTRAINT fk_store_staff FOREIGN KEY
(manager_staff_id) REFERENCES staff (staff_id);

ALTER TABLE store
ADD CONSTRAINT fk_store_address FOREIGN KEY (address_id)
REFERENCES address (address_id);

ALTER TABLE staff
ADD CONSTRAINT fk_staff_store FOREIGN KEY (store_id)
REFERENCES store (store_id);

ALTER TABLE customer
ADD CONSTRAINT fk_customer_address FOREIGN KEY (address_id)
REFERENCES address (address_id);

ALTER TABLE customer
ADD CONSTRAINT fk_customer_store FOREIGN KEY (store_id)
REFERENCES store (store_id);

ALTER TABLE film
ADD CONSTRAINT fk_film_language FOREIGN KEY (language_id)
REFERENCES language (language_id);



ALTER TABLE film
ADD CONSTRAINT fk_film_language_original FOREIGN KEY
(original_language_id) REFERENCES language (language_id);

ALTER TABLE film_actor
ADD CONSTRAINT fk_film_actor_actor FOREIGN KEY (actor_id)
REFERENCES actor (actor_id);

ALTER TABLE film_actor
ADD CONSTRAINT fk_film_actor_film FOREIGN KEY (film_id)
REFERENCES film (film_id);
```

```
ALTER TABLE film_category
ADD CONSTRAINT fk_film_category_film FOREIGN KEY (film_id)
REFERENCES film (film_id);

ALTER TABLE film_category
ADD CONSTRAINT fk_film_category_category FOREIGN KEY
(category_id) REFERENCES category (category_id);

ALTER TABLE inventory
ADD CONSTRAINT fk_inventory_store FOREIGN KEY (store_id)
REFERENCES store (store_id);

ALTER TABLE inventory
ADD CONSTRAINT fk_inventory_film FOREIGN KEY (film_id)
REFERENCES film (film_id);


ALTER TABLE rental
ADD CONSTRAINT fk_rental_staff FOREIGN KEY (staff_id)
REFERENCES staff (staff_id);

ALTER TABLE rental
ADD CONSTRAINT fk_rental_inventory FOREIGN KEY
(inventory_id) REFERENCES inventory (inventory_id);

ALTER TABLE rental
ADD CONSTRAINT fk_rental_customer FOREIGN KEY (customer_id)
REFERENCES customer (customer_id);

ALTER TABLE payment
ADD CONSTRAINT fk_payment_rental FOREIGN KEY (rental_id)
REFERENCES rental (rental_id);

ALTER TABLE payment
ADD CONSTRAINT fk_payment_customer FOREIGN KEY
(customer_id) REFERENCES customer (customer_id);

ALTER TABLE payment
ADD CONSTRAINT fk_payment_staff FOREIGN KEY (staff_id)
REFERENCES staff (staff_id);
```

III. Anexo 3 – Povoamento MySQL

```
#!/usr/bin/env python3
import re
import cx_Oracle
import datetime

tabelas = {}
tabelas["rental"] = "insert into rental(rental_id,
rental_date, inventory_id, customer_id, return_date,
staff_id, last_update) values"
tabelas["film_category"] = "insert into
film_category(film_id, category_id, last_update) values"
tabelas["language"] = "insert into language(language_id,
name, last_update) values"
tabelas["inventory"] = "insert into inventory(inventory_id,
film_id, store_id, last_update) values"
tabelas["actor"] = "insert into actor(actor_id, first_name,
last_name, last_update) values"
tabelas["category"] = "insert into category(category_id,
name, last_update) values"
tabelas["country"] = "insert into country(country_id,
country, last_update) values"
tabelas["city"] = "insert into city(city_id, city,
country_id, last_update) values"
tabelas["address"] = "insert into address(address_id,
address, address2, district, city_id, postal_code, phone,
last_update) values"
tabelas["store"] = "insert into store(store_id,
manager_staff_id, address_id, last_update) values"
tabelas["film_text"] = "insert into film_text(film_id,
title, description) values"
tabelas["customer"] = "insert into customer(customer_id,
store_id, first_name, last_name, email, address_id, active,
create_date, last_update) values"
tabelas["film"] = "insert into film(film_id, title,
description, release_year, language_id,
original_language_id,rental_duration,rental_rate, length,
replacement_cost, rating, special_features, last_update)
values"
tabelas["film_actor"] = "insert into film_actor(actor_id,
film_id, last_update) values"
tabelas["payment"] = "insert into payment(payment_id,
customer_id, staff_id, rental_id, amount, payment_date,
last_update) values"
tabelas["staff"] = "insert into
staff(staff_id,first_name,last_name,address_id,email,store_
id,active,username,password,last_update) values"

def leLinhas(inserir):
```

```
     conn =
cx_Oracle.connect('final/123@localhost:1521/orcl')
     mycursor = conn.cursor()
     with open("sakila-data (1).sql") as file:
          for line in file:
               after = re.search("VALUES",line)
               if(after is not None):
                    firstLine = line.split("VALUES")
                    tabela = line.split(" ")
                    chave = tabela[2].split("`")
                    if(len(chave)>1):
                         c = chave[1]
                    else:
                         c = chave[0]
                    inicio = tabelas[c]
                    inserir = 1 #indica que vamos começar a
povoar
                    line = firstLine[1]
               if(inserir==1):
                    if(re.search(";",line) is not None):
                         inserir = 0
                         l = line.split(';')
                         line = l[0]
                    else:
                         virgulas = line.split(",\n")
                         line = virgulas[0]
                    if(re.match("customer",c) is not None):
                         match = re.search(r'\'\d{4}\-
                         \d{2}\-\d{2}
                         \d{2}\:\d{2}\:\d{2}\'', line)
                         nova = match.group()
                              nData = 'TO_DATE(' +
     match.group()             + ",'" + 'yyyy/mm/dd
     hh24:mi:ss\')'
                         nData.lstrip("T")
                         line = line.replace(nova,nData)
                         run = inicio + line
                    elif(re.match("film",c) is not None and
                    re.match("film_",c) is None):
                         match =
                         re.search(r'\d{1,2}\.\d{2}', line)
                         nova = match.group()
                         line = line.replace("'" + nova
+"'",nova)
                         match2 =
re.search(r'\'\d{1,2}\.\d{2}\'', line)
                         nova2 = match2.group()
                         n = nova2.split("'")
                         i = n[1]
                         line = line.replace(nova2, i)
                         run = inicio + line
```

```
                    elif(re.match("payment",c) is not
None):
                         campos = line.split(",");
                         newN =
re.search(r'\d+.\d+',campos[4]).group() #convertTonumber
                         newDate = 'TO_DATE(' + campos[5] +
',' + "'yyyy/mm/dd hh24:mi:ss')"
                         run = inicio + campos[0] + "," +
campos[1] + "," + campos[2] + "," + campos[3] +"," + newN
+"," + newDate + "," + campos[6]
                    elif(re.match("rental",c) is not None):
                         campos = line.split(",");
                         newDate = 'TO_DATE(' + campos[1] +
',' + "'yyyy/mm/dd hh24:mi:ss')"
                         newD = 'TO_DATE(' + campos[4] +
',' + "'yyyy/mm/dd hh24:mi:ss')"
                         run = inicio + campos[0] + "," +
newDate + "," + campos[2] + "," + campos[3] +"," + newD
+"," + campos[5] + "," + campos[6]
                    elif(re.match("staff",c) is not None):
                         campos = line.split(",");
                         new10 = campos[10].split(")")
                         newT = 'TO_TIMESTAMP(' + new10[0]
+ ',' + "'yyyy/mm/dd hh24:mi:ss.FF')"
                         run = inicio + campos[0] + "," +
campos[1] + "," + campos[2] + "," + campos[3] +"," +
campos[5] + "," + campos[6] + "," + campos[7] + "," +
campos[8]  + "," + campos[9] + "," + newT + ")"
                    else:
                         nLine = line.replace(r"''","' '")
                         run = inicio + nLine
                    print(run)
                    mycursor.execute(run)
     conn.commit()
     mycursor.close()
     conn.close()


leLinhas(0)
```

IV. Anexo 4 – Exportação de Dados Neo4j

```
select * from country
into outfile '/var/lib/mysql-files/country.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select city, country from city
inner join country on city.country_id = country.country_id
into outfile '/var/lib/mysql-files/city.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select address, address2, district, city, postal_code,
phone, country from address
inner join city on address.city_id = city.city_id
inner join country on city.country_id = country.country_id
into outfile '/var/lib/mysql-files/address.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select first_name, last_name, address, city, email, active,
username, password, store_id from staff
inner join address on address.address_id = staff.address_id
inner join city on city.city_id = address.city_id
into outfile '/var/lib/mysql-files/staff.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select address, city, username, store.store_id from store
inner join address on address.address_id = store.address_id
inner join city on city.city_id = address.city_id
inner join staff on staff.staff_id = store.manager_staff_id
into outfile '/var/lib/mysql-files/store.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select name from category
into outfile '/var/lib/mysql-files/category.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select actor_id, first_name, last_name from actor
into outfile '/var/lib/mysql-files/actor.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select first_name, last_name, email, active, create_date,
store_id, address, city from customer
inner join address on address.address_id =
customer.address_id
```

```
inner join city on address.city_id = city.city_id
into outfile '/var/lib/mysql-files/customer.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select rental_id, customer.first_name, customer.last_name,
amount, payment_date, staff.username, payment_id from
payment
inner join customer on customer.customer_id =
payment.customer_id
inner join staff on staff.staff_id = payment.staff_id
into outfile '/var/lib/mysql-files/payment.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select name from language
into outfile '/var/lib/mysql-files/language.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

select title, description, release_year, rental_duration,
rental_rate, film.length, replacement_cost, rating,
special_features, language.name from film
inner join language on language.language_id =
film.language_id
into outfile '/var/lib/mysql-files/film.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED
BY '\n';

select title, actor.actor_id from film
inner join film_actor on film.film_id = film_actor.film_id
inner join actor on film_actor.actor_id = actor.actor_id
into outfile '/var/lib/mysql-files/film_actor.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED
BY '\n';

select film.title, store_id, film_text.description,
count(*) from inventory
inner join film on inventory.film_id = film.film_id
inner join film_text on film_text.film_id =
inventory.film_id
group by film.film_id, inventory.store_id
into outfile '/var/lib/mysql-files/inventory.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED
BY '\n';

select title, category.name from film
inner join film_category on film.film_id =
film_category.film_id
inner join category on category.category_id =
film_category.category_id
```

```
into outfile '/var/lib/mysql-files/film_category.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED
BY '\n';

select rental_id, rental_date, ifnull(return_date,""),
customer.first_name, customer.last_name, staff.username,
film.title, inventory.store_id from rental
inner join inventory on inventory.inventory_id =
rental.inventory_id
inner join customer on customer.customer_id =
rental.customer_id
inner join staff on staff.staff_id = rental.staff_id
inner join film on inventory.film_id = film.film_id
into outfile '/var/lib/mysql-files/rental.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED
BY '\n';

select payment.payment_id, rental.rental_id from payment
inner join rental on payment.rental_id = rental.rental_id
into outfile '/var/lib/mysql-files/rental-payment.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED
BY '\n';
```

V. Anexo 5 – Importação de Dados Neo4j

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

create index on :Country(country);

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

create index on :City(city);

using periodic commit 100
load csv from "file:///address.csv" as line
match (ci: City {city: line[3]})-[:pertence_a]->(co:
Country {country: line[6]})
create (adr: Address { address: line[0], address2: line[1],
district: line[2], code: line[4], phone: line[5]}),
       (adr)-[r:pertence_a]->(ci);

create index on :Address(address);

load csv from "file:///staff.csv" as line
match (adr: Address {address: line[2]})-[:pertence_a]->(ci:
City {city : line[3]})
create (st: Staff { first_name: line[0], last_name:
line[1], email: line[4], active: line[5], username:
line[6], password: line[7]}),
       (st)-[r:vive_em]->(adr);

load csv from "file:///store.csv" as line
match (adr: Address {address: line[0]})-[:pertence_a]->(ci:
City {city : line[1]})
match (m: Staff {username: line[2]})
create (adr)<-[:localizada_em]-(sto: Store {store_id:
line[3]})-[:gerido_por]->(m);

create index on :Store(store_id);

load csv from "file:///staff.csv" as line
match (staff: Staff {username: line[6]})
match (store: Store {store_id: line[8]})
create (staff)-[:trabalha_em]->(store);

create index on :Staff(username);

load csv from "file:///category.csv" as line
create (:Category{name:line[0]});
```

```
create index on :Category(name);

load csv from "file:///actor.csv" as line
create (:Actor{actor_id: line[0], first_name: line[1],
last_name: line[2]});

create index on :Actor(actor_id);
create index on :Actor(first_name, last_name);

load csv from "file:///customer.csv" as line
match (adr: Address {address: line[6]})-[:pertence_a]->(c:
City {city: line[7]})
match (st: Store {store_id: line[5]})
create (adr)<-[:vive_em]-(:Customer {first_name: line[0],
last_name: line[1], email: line[2], active: line[3],
create_date: replace(line[4],' ','T')})-[:membro_em]->(st);

load csv from "file:///payment.csv" as line
match (c: Customer {first_name: line[1], last_name:
line[2]})
match (s: Staff {username: line[5]})
create (c)<-[:pago_por]-(p:Payment {amount:
toFloat(line[3]), payment_date: replace(line[4],' ','T'),
payment_id: line[6]})-[:feito_por]->(s);

create index on :Payment(payment_id);

load csv from "file:///language.csv" as line
create (:Language {name: line[0]});

create index on :Language(name);

load csv from "file:///film.csv" as line
match (l:Language {name: line[9]})
create (:Film {title: line[0], description: line[1],
release_year: line[2], rental_duration: line[3],
rental_rate: line[4], length: line[5], replacement_cost:
line[6], rating: line[7], special_features: line[8]})-
[:com_lingua]->(l);

create index on :Film(title);

load csv from "file:///film_actor.csv" as line
match (f: Film {title: line[0]})
match (a: Actor {actor_id: line[1]})
create (f)-[:com_participacao_de]->(a);

match (a:Actor)
remove a.actor_id;
```

```
load csv from "file:///inventory.csv" as line
match (f: Film {title: line[0]})
match (s: Store {store_id: line[1]})
create (s)-[:tem_filme {numero: line[3]}]->(f);

load csv from "file:///film_category.csv" as line
match (f: Film {title: line[0]})
match (c: Category {name: line[1]})
create (f)-[:com_categoria]->(c);

load csv from "file:///rental.csv" as line
match (c: Customer {first_name: line[3], last_name:
line[4]})
match (sta: Staff {username: line[5]})
match (f: Film {title: line[6]})
match (sto: Store {store_id: line[7]})
create (sta)<-[:feito_por]-(r:Rental {rental_id: line[0],
rental_date: replace(line[1],' ','T'), return_date:
replace(line[2], " ", "T")})-[:filme_alugado]->(f),
       (sto)<-[:na_loja]-(r)-[:alugado_por]->(c);

create index on :Rental(rental_id);

load csv from "file:///rental-payment.csv" as line
match (p: Payment {payment_id: line[0]})
match (r: Rental {rental_id: line[1]})
create (p)-[:alugado]->(r);

match (p:Payment)
remove p.payment_id;

match (r:Rental)
remove r.rental_id;
```

VI. Anexo 6 – Conversão do ficheiro adresses.json

```
import json
# Função que encontra o objeto cidade com o id passado
def find_city(id):
    f = open("../oldJson/city.json","rb")
    citys = json.loads(f.read())
    i = 0
    while(i < 600):
        if(citys["city"][i]["city_id"] == id):
            return citys["city"][i]
        i+=1
    return

# Função que encontra o nome do país com o id passado
def find_country(id):
    f= open("../oldJson/country.json","rb")
    countries = json.loads(f.read())
    i = 0
    countr = countries["country"]
    while(i < 109):
        if(countr[i]["country_id"] == id):
            return countr[i]["country"]
        i+=1
    return

#Função que processa os endereços
def processAddresses():
    f = open("../oldJson/addresses.json","rb")
    addresses = json.loads(f.read())
    i = 0
    while(i < 603):
        if(addresses["address"][i]):
            address = addresses["address"][i]
            # FIND CITY AND COUNTRY
            city = find_city(address["city_id"])
            city_name = city["city"]
            country = find_country(city["country_id"])
            # CHANGE FIELDS AND VALUES
            address.pop("city_id")
            address["city"] = city_name
            address["country"] = country
            address.pop("address2")
            address.pop("last_update")
            address.pop("phone")
        i+=1

    new = open("../newJson/addresses.json","w")
    json.dump(addresses,new,indent=4)

processAddresses()
```

VII. Anexo 7 – Conversão do ficheiro customers.json

```
import json

#----------------------- ADDRESS INFORMATION -------------#
def find_address(address_id):
    f = open("../newJson/addresses.json","rb")
    addresses = json.loads(f.read())
    i = 0 #contador para percorrer staff
    address = addresses["address"]
    size = len(address)
    while(i < size):
        if(address[i]["address_id"] == address_id):
            return address[i]
        i+=1
    return
#----------------------- ADDRESS INFORMATION -------------#

#----------------------- MAIN FUNCTION -------------------#
def customers_dataset():
    f = open("../oldJson/customer.json","rb") #abrir
ficheiro rentals
    customers = json.loads(f.read()) #transformar ficheiro
para objeto em python
    i = 0 #inicializar contador
    size = len(customers["customer"])
    while(i < size): #Enquanto houver entradas
        if(customers["customer"][i]):
            customer = customers["customer"][i]
            name = customer["first_name"] + " " +
customer["last_name"]
            mail = customer["email"]
            address = find_address(customer["address_id"])
            address.pop("address_id")

            customer.pop("active")
            customer.pop("email")
            customer.pop("create_date")
            customer.pop("last_update")
            customer.pop("store_id")
            customer.pop("address_id")
            customer.pop("first_name")
            customer.pop("last_name")

            customer["name"] = name
            customer["email"]= mail
            customer["address"] = address

        i += 1
    new = open("../newJson/customers.json","w")
    json.dump(customers,new,indent=3)
```

```
def main():
    customers_dataset()

main()
```

VIII. Anexo 8 – Conversão do ficheiro film.json

```
import json

def find_actor(id):
    f = open("../oldJson/atores.json","r")
    actores = json.loads(f.read())
    actor = actores["actor"]
    i = 0
    if(id == "110"):
        end = " (2)"
    else:
        end = ""
    while(i < 201):
        if(actor[i]["actor"] == id):
            actor[i].pop("last_update")
            actor[i].pop("actor")
            fn = actor[i]["first_name"]
            ln = actor[i]["last_name"]
            actor[i]["name"] = fn + " " + ln + end
            actor[i].pop("first_name")
            actor[i].pop("last_name")
            #return (actor[i])
            return actor[i]["name"]
        i+=1
    return

def find_actores_filme(idFilme):
    f = open("../oldJson/film_actor.json","r")
    data = json.loads(f.read())
    data = data["film_actor"]
    i = 0
    j = 0
    actors = []
    while(i < 5462):
        if(data[i]["film_id"] == idFilme):
            actor = find_actor(data[i]["actor_id"])
            actors.insert(j,actor)
            j+=1
        i+=1
    return actors

def find_language(id):
    f = open("../oldJson/language.json","rb")
    languages = json.loads(f.read())
    i = 0
    while(i < 6):
        if(languages["language"][i]["language_id"] == id):
            return languages["language"][i]["name"]
        i+=1
    return
```

```
def find_category(id):
    f = open("../oldJson/category.json","rb")
    cats = json.loads(f.read())
    i = 0
    while(i < 16):
        if(cats["category"][i]["category_id"] == id):
            return cats["category"][i]["name"]
        i+=1
    return

def find_film_category(id):
    f= open("../oldJson/film_category.json","rb")
    films = json.loads(f.read())
    i = 0
    film = films["film_category"]
    while(i < 1000):
        if(film[i]["film_id"] == id):
            cat = find_category(film[i]["category_id"])
            return cat
        i+=1
    return

def processFilms():
    f = open("../oldJson/film.json","rb")
    films = json.loads(f.read())
    i = 0
    while(i < 1000):
        if(films["film"][i]):
            film = films["film"][i]
            # FIND LANGUAGE
            language = find_language(film["language_id"])
            # FIND CATEGORY
            category = find_film_category(film["film_id"])
            actors = find_actores_filme(film["film_id"])
            # CHANGE FIELDS AND VALUES
            film.pop("language_id")
            film["language"] = language
            film["category"] = category
            film["actors"] = actors
            film.pop("original_language_id")
            film["film_id"] = int(film["film_id"])

        i+=1

    new = open("../newJson/film.json","w")
    json.dump(films,new,indent=3)


processFilms()
```

IX. Anexo 9 – Conversão do ficheiro rental.json

```
import json
from datetime import datetime

def find_payment(rental_id):
    f = open("../oldJson/payment.json","rb")
    payments = json.loads(f.read())
    i = 0
    payment = payments["payment"]
    size = len(payment)
    while(i < size):
        if(payment[i]["rental_id"] == rental_id):
            return float(payment[i]["amount"])
        i+=1
    return

def find_location(address_id):
    f = open("../newJson/addresses.json","rb")
    addresses = json.loads(f.read())
    i = 0
    address = addresses["address"]
    size = len(address)
    while(i < size):
        if(address[i]["address_id"] == address_id):
            add = address[i]["address"]
            city = address[i]["city"]
            district = address[i]["district"]
            country = address[i]["country"]
            return add,city,district,country
        i+=1
    return

def find_store(store_id):
    f = open("../oldJson/store.json","rb")
    stores = json.loads(f.read())
    i = 0
    store = stores["store"]
    size = len(store)
    while(i < size):
        if(store[i]["store_id"] == store_id):
            return find_location(store[i]["address_id"])
        i+=1
    return ("","","","")




def find_staff(staff_id):
    f = open("../oldJson/staff.json","rb")
    staffs = json.loads(f.read())
    i = 0 #contador para percorrer staff
```

```
    staff = staffs["staff"]
    size = len(staff)
    while(i < size):
        if(staff[i]["staff_id"] == staff_id):
            name = staff[i]["first_name"] + " " +
staff[i]["last_name"]
            mail = staff[i]["email"]
            store_id = staff[i]["store_id"]
            staff[i].pop("email")
            staff[i].pop("store_id")
            staff[i].pop("password")
            staff[i].pop("username")
            staff[i].pop("active")
            staff[i].pop("address_id")
            staff[i].pop("first_name")
            staff[i].pop("last_name")
            address,city,district,country =
find_store(store_id)
            staff[i]["name"] = name
            staff[i]["email"] = mail
            staff[i]["store_id"] = store_id
            staff[i]["store_address"] = address
            staff[i]["store_city"] = city
            staff[i]["store_district"] = district
            staff[i]["store_country"] = country
            return staff[i]
        i+=1
    return

def find_film(film_id):
    f = open("../newJson/film.json","rb")
    films = json.loads(f.read())
    i = 0
    film = films["film"]
    size = len(film)
    while(i < size):
        if(film[i]["film_id"] == film_id):
            film[i].pop("release_year")
            film[i].pop("rental_duration")
            film[i].pop("rental_rate")
            film[i].pop("length")
            film[i].pop("replacement_cost")
            film[i].pop("rating")
            film[i].pop("last_update")
            return film[i]
        i+=1
    return

def find_inventory(inventory_id):
    f = open("../oldJson/inventory.json","rb")
    invs = json.loads(f.read())
    i = 0
```

```
    inventory = invs["inventory"]
    size = len(inventory)
    while(i < size):
        if(inventory[i]["inventory_id"] == inventory_id):
            film = find_film(int(inventory[i]["film_id"]))
            return film
        i+=1
    return

def rental_dataset():
    f = open("../oldJson/rental.json","rb") #abrir ficheiro
rentals
    rentals = json.loads(f.read()) #transformar ficheiro
para objeto em python
    i = 0 #inicializar contador
    size = len(rentals["rental"])
    while(i < size): #Enquanto houver entradas
        if(rentals["rental"][i]):
            rental = rentals["rental"][i] #Pegar na entrada
Rental
            customer_id = rental["customer_id"] #id do
cliente
            payment_value = find_payment(str(rental["id"]))
            staff= find_staff(str(rental["staff_id"]))
            film =
find_inventory(str(rental["inventory_id"]))
            try:
                time_rental_date =
datetime.strptime(rental["rental_date"], '%Y-%m-%d
%H:%M:%S.%f')
            except(ValueError):
                time_rental_date = ""
            try:
                time_return_date =
datetime.strptime(rental["return_date"], '%Y-%m-%d
%H:%M:%S')
            except(ValueError):
                time_return_date = ""
            rental_date = rental["rental_date"]
            return_date = rental["return_date"]
            if(time_rental_date != "" and time_return_date
!= ""):
                rental_duration = time_return_date -
time_rental_date
            else:
                rental_duration = ""

            rental.pop("staff_id")
            rental.pop("last_update")
            rental.pop("customer_id")
            rental.pop("rental_date")
            rental.pop("return_date")
```

```
            rental.pop("inventory_id")

            rental["customer_id"] = customer_id
            rental["rental_date"] = rental_date
            rental["return_date"] = return_date
            rental["rental_duration"] =
str(rental_duration)
            rental["payment_value"] = payment_value
            rental["staff"] = staff
            rental["film"] = film
            print("Na iteração nº: " + str(i))

        i+=1
    new = open("../newJson/rental.json","w")
    json.dump(rentals,new,indent=3)

def main():
    rental_dataset()

main()
```