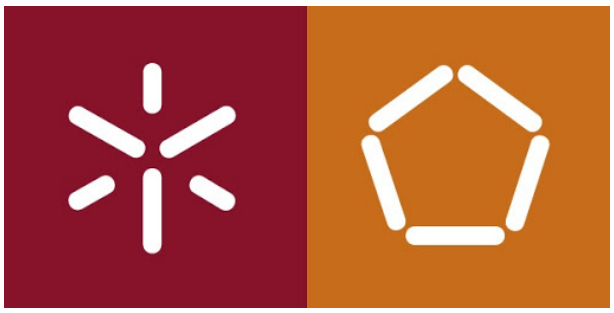

Gestão de Grandes Conjuntos de Dados
Trabalho Prático 2 - Spark
Grupo 11

Daniel Regado (PG42577)

Conteúdo

1 | Introdução

Este trabalho prático tem como objetivo aplicar os conhecimentos adquiridos nas aulas relativos à utilização do *stack* Spark, Hive e HDFS. Além disso, de forma a aprofundar conhecimentos adicionais, embora não diretamente especificado no enunciado, irá também ser abordado o HBase para armazenamento de dados de forma distribuída. Todos estes conceitos e componentes estão disponíveis recorrendo ao Google Cloud Platform (GCP), pelo que também é objetivo deste trabalho prático compreender a interação entre os componentes, assim como o funcionamento de *clusters* no *Dataproc*.

2 | Preparação e Contextualização

Tendo em conta o desenvolvimento do trabalho prático anterior, existe trabalho que pode ser aproveitado, por exemplo, configurações Maven, arquitetura básica do *cluster* e ferramentas para interação com este. Por essa razão, existirão similaridades nos desenvolvimentos em componentes que sejam comuns a ambos trabalhos práticos.

2.1 Maven

Nesta secção abordamos as configurações presentes em *pom.xml*, ou seja, as configurações do Maven. Em termos de propriedades, definimos como *compiler* a versão 8 do Java, de forma a ser compatível com a versão presente do *Dataproc*. Além disso, foi definido também o *encoding* como UTF-8, de forma a suprimir *warning* na fase de *packaging*.

2.1.1 Dependências

Spark e Spark-SQL

Estas dependências já vêm instaladas no *cluster*, portanto podemos assinalar como *provided*. Trata-se das dependências *spark-core* e *spark-sql* na versão 3.1.1.

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.12</artifactId>
    <version>3.1.1</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.12</artifactId>
    <version>3.1.1</version>
    <scope>provided</scope>
</dependency>
```

Hive

A dependência *Hive* também já está pré-instalada no *cluster*, portanto também é assinalada como *provided*. Está em utilização a versão 3.1.1.

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-hive_2.12</artifactId>
    <version>3.1.1</version>
    <scope>provided</scope>
</dependency>
```

HBase

A seguinte dependência não vem instalada por *default* no *cluster*, pelo que esta não será assinalada como *provided*. Está em utilização a versão 2.1.9, pelo que foi a última versão que funciona no contexto do *Dataproc*. Foi experimentada a última versão disponível (2.4.3), mas ocorreram problemas de conexão no contexto da API Java.

```
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-shaded-client</artifactId>
    <version>2.1.9</version>
</dependency>
```

2.1.2 Plugins

Jar

Este plugin serve para a geração do *jar* na altura de *package*, e ativamos a configuração *forceCreation* para que este não ativasse um warning na criação do jar, pois este iria entrar em conflito com uma versão anterior que tivesse sido gerada.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.2.0</version>
    <configuration>
        <forceCreation>true</forceCreation>
    </configuration>
</plugin>
```

Shade

O *shade* serve para incluir as dependências associadas ao projeto no jar, de forma a que o ficheiro resultante possa ser executado no *Dataproc* sem instalações adicionais. Abaixo temos a configuração do *shade*. Foi desativa a criação do *dependency-reduced-pom.xml*, pois ocasionalmente o IntelliJ associava esse ficheiro a uma nova configuração Maven. Não foi ativa a configuração de *minimizeJar* pois isso fazia com que fossem retiradas dependências que eram utilizadas posteriormente no cluster. A configuração deste plugin está feita de forma a evitar warnings e a descartar dependências duplicadas.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
```

```
            <version>3.2.4</version>
            <configuration>
                <createDependencyReducedPom>false</createDependencyReducedPom>
            </configuration>
            ...
        </plugin>
```

2.2 Estrutura do projeto

Em termos de estrutura do projeto, mais propriamente na pasta *src*, temos o seguinte conteúdo:

- gcloud
- main
- hive

Em *gcloud*, temos um *bash script* para interação com o cluster, que será abordado de forma mais elaborada seguidamente.
Em *main*, temos o *source code* Java do projeto. Está estruturado por packages, e estes estão organizados de forma intuitiva.
Por fim, em *hive* temos os ficheiros *hql*, que tratam da passagem dos dados em formatos de texto para dados organizados no Hive, tomando assim partido de uma estrutura organizada, assim como da existência dos metadados.

2.3 Cluster

De modo a não estar sujeito apenas ao browser para interações com o cluster, ou a relembrar comandos extensos, foi desenvolvido um *bash script*, *cluster.sh*, para interagir com o cluster. Neste ficheiro, estão definidos parâmetros que são utilizados repetidamente, como por exemplo o *project name* e *cluster name*, entre outros. Além disso, fazer alterações de arquitetura no cluster também se torna mais fácil deste modo. Este script tem como pré-requisito o *Google Cloud SDK* em funcionamento.

2.3.1 Arquitetura

Como parâmetros do cluster, definimos os seguintes atributos:

```
PROJECT_ID="ggcd-spark"
CLUSTER_NAME="spark-cluster"
REGION="europe-west2"
ZONE="europe-west2-b"
MASTER_MACHINE_TYPE="e2-standard-4"
WORKER_MACHINE_TYPE="e2-highmem-4"
DISK_SIZE=50
DISK_TYPE="pd-ssd"
WORKERS=2
SECONDARY_WORKERS=2
```

```
IMAGE_VERSION="2.0-debian10"
MAX_IDLE_SECONDS="3600s"
```

Desta forma, temos definidos neste script as caraterísticas dos cluster. Foi escolhida uma arquitetura que excedesse as necessidades mínimas para execução dos Jobs, de forma a tornar estes mais rápidos e a suportar os *datasets* na íntegra. Foram escolhidos discos SSD de forma a melhorar a performance de I/O do HDFS. Além disso, foi colocado um *max idle time* de 1 horas, de forma a que não sejam gastos recursos desnecessariamente.
De forma a tirar partido das funcionalidades oferecidas pela GCP, foram incorporados *secondary workers* na arquitetura escolhida. Este *workers* não contribuem armazenamento para o *HDFS*, pois são máquinas *preemptible*. Isto significa que estas máquinas estão sujeitas a paragens caso sejam necessárias para outras tarefas pela parte da Google, e portanto não podem armazenar dados de forma persistente. No entanto, vêm com a vantagens de serem consideravelmente mais baratas, e dado que as tarefas que executamos têm no máximo alguns minutos de duração, conseguimos tirar proveito do poder computacional destes *workers*.

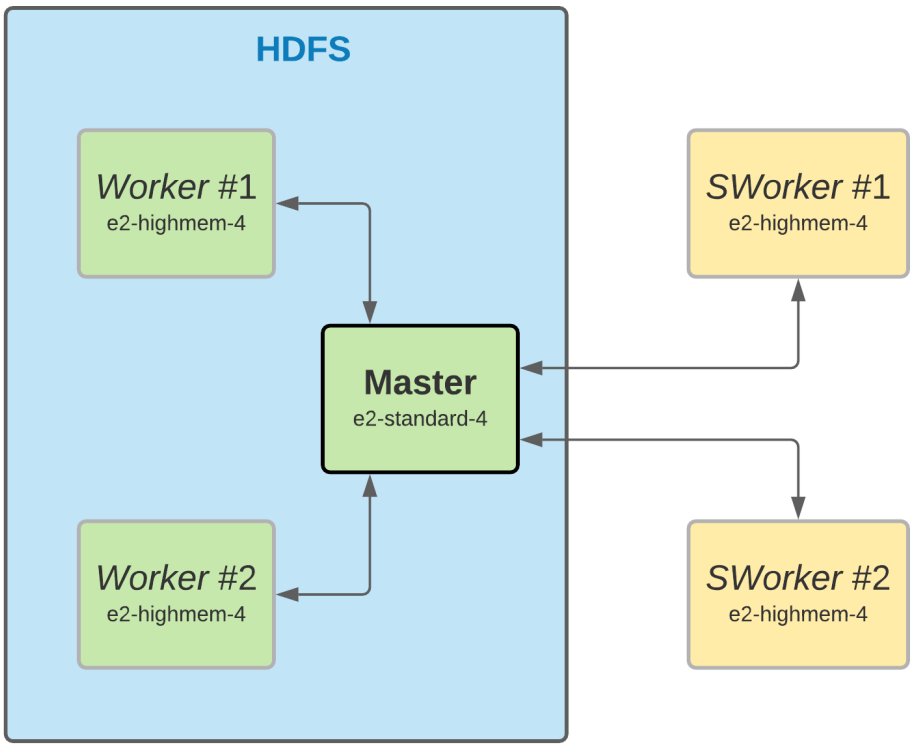

Figura 2.1: Arquitetura do cluster

2.3.2 Ações

Estão implementadas as seguintes ações de interação com o cluster:

Descrição	**Comando**
Cria o cluster	$ ./cluster.sh create
Apaga o cluster	$ ./cluster.sh delete
Inicia o cluster	$ ./cluster.sh start
Pára o cluster	$ ./cluster.sh stop
Submite job Spark para cluster	$ ./cluster.sh spark `<JAR> <CLASS>`
Submite job Hive para cluster	$ ./cluster.sh hive `<QUERY_PATH>`
Criar diretoria no HDFS	$ ./cluster.sh hdfs_mkdir `<DIR>`
Upload de ficheiro para HDFS	$ ./cluster.sh hdfs_upload `<SRC> <DEST>`
Download de ficheiro para HDFS	$ ./cluster.sh hdfs_download `<SRC> <DEST>`
Apagar ficheiro no HDFS	$ ./cluster.sh hdfs_delete `<FILE/DIR>`
Listar diretoria no HDFS	$ ./cluster.sh hdfs_ls `<DIR>`

3 | Hive

Neste capítulo, iremos abordar as *queries* desenvolvidas para cumprir o objetivo 1 deste trabalho prático, que consiste em carregar os ficheiros para o Hive, em vez de tratar os ficheiros diretamente em formato de texto. Ao colocar os ficheiros no Hive, obtemos metadados, que serão úteis, por exemplo, para optimizar operações de *join* em *queries*.

3.1 Localização dos ficheiros

De modo a não ter que especificar na execução de cada Job quais seriam os ficheiros de input, foi decidido a utilização de diretorias pré-definidas para localização dos datasets. Esta decisão foi tomada de modo a simplificar a execução dos Jobs, de modo a não ser necessário definir explicitamente a localização de cada ficheiro como argumento. Portanto, para a geração das tabelas, estas assumem a existência dos datasets nas seguintes diretorias:

```
location 'hdfs:///titleBasics'      /* titleBasics.hql      */
location 'hdfs:///titleRatings'     /* titleRatings.hql     */
location 'hdfs:///titlePrincipals'  /* titlePrincipals.hql  */
location 'hdfs:///nameBasics'       /* nameBasics.hql       */
```

3.2 Nomes das tabelas

Mais uma vez, de forma a simplificar a execução dos jobs, o nome das colunas onde são guardados os dados também estão definidas nestes ficheiros, e são utilizados esses nomes no resto do projeto. Essas tabelas são as seguintes:

Dataset	**Tabela Hive**
title.basics	titleBasics
title.ratings	titleRatings
title.principal	titlePrincipals
name.basics	nameBasics

3.3 Especificação dos datasets

Retirando a localização, todas as *queries* definidas para a criação das tabelas no Hive contém as seguintes especificações para os datasets:

```
row format delimited
    fields terminated by '\t'
    lines terminated by '\n'
    NULL defined as '\\N'
stored as textfile
tblproperties ("skip.header.line.count"="1");
```

Além disso, os datasets que contêm campos compostos, como por exemplo o caso de *genres* em *title.basics*, definem adicionalmente a seguinte especificação:

```
collection items terminated by ','
```

3.4 Definição de tipos

De forma a minimizar o espaço ocupado e a definir devidamente os tipos para cada atributo, foi tida em atenção a amplitude e casos possíveis para cada atributo. Na secção abaixo, conseguimos observar alguns exemplos os tipos de dados utilizados:

```
averageRating decimal(10,1)
numVotes integer
tconst string
isAdult boolean
genres array<string>
startYear smallint
```

3.5 Estruturação dos dados

Em termos de estruturação dos dados, iremos fazer testes comparando resultados com estruturação em *ORC* e *Parquet*, de forma a verificar qual a melhor abordagem, para os jobs em causa. Em ambos os casos, são estruturas colunares. Na análise de resultados, irá ser feita a execução dos Jobs recorrendo às tabelas com as diferentes estruturas, de forma a concluir se existem diferenças significativas em termos de utilização de recursos, analisando métricas e logs.

3.6 Execução

Para executar as 4 *queries* desenvolvidas, uma para cada dataset, utiliza-se o *script* cluster.sh referido anteriormente. Recorre-se então ao comando `$ ./cluster.sh hive <HQL_FILE>` para executar os jobs no Hive. Podemos depois executar o comando *describe* no Hive, para observar as tabelas geradas:

```
> describe titleRatings

+----------------+----------------+----------+
|    col_name    |   data_type    | comment  |
+----------------+----------------+----------+
| tconst         | string         |          |
| averagerating  | decimal(10,1)  |          |
| numvotes       | int            |          |
+----------------+----------------+----------+
```

```
> describe titlePrincipals

+-------------+----------------+----------+
|  col_name   |   data_type    | comment  |
+-------------+----------------+----------+
| tconst      | string         |          |
| ordering    | smallint       |          |
| nconst      | string         |          |
| category    | string         |          |
| job         | string         |          |
| characters  | array<string>  |          |
+-------------+----------------+----------+


> describe nameBasics

+--------------------+----------------+----------+
|      col_name      |   data_type    | comment  |
+--------------------+----------------+----------+
| nconst             | string         |          |
| primaryname        | string         |          |
| birthyear          | smallint       |          |
| deathyear          | smallint       |          |
| primaryprofession  | array<string>  |          |
| knownfortitles     | array<string>  |          |
+--------------------+----------------+----------+


> describe titleBasics

+-------------------------+----------------+----------+
|         col_name        |   data_type    | comment  |
+-------------------------+----------------+----------+
| tconst                  | string         |          |
| primarytitle            | string         |          |
| originaltitle           | string         |          |
| isadult                 | boolean        |          |
| startyear               | smallint       |          |
| endyear                 | smallint       |          |
| runtimeminutes          | smallint       |          |
| genres                  | array<string>  |          |
| titletype               | string         |          |
|                         | NULL           | NULL     |
| # Partition Information | NULL           | NULL     |
| # col_name              | data_type      | comment  |
| titletype               | string         |          |
+-------------------------+----------------+----------+
```

3.7 Resultados

Os datasets processados correspondem aos datasets totais, retirados de imbd.com/interfaces, e comprimidos no formato bz2. É de salientar que de todas as operações definidas nos ficheiros *hql*, apenas a operação de *insert* é que é realmente tratada como um job, visto que as operações de criar tabelas não acedem diretamente a dados. Tendo isso em conta, vamos analisar utilização de recursos na etapa de *insert*. As tabelas apresentadas seguidamente exibem valores provenientes do Yarn ResourceManager:

Parquet			
Table	**Time**	**MB-seconds**	**vcore-seconds**
titleBasics	1m45s	1414571	199
titleRatings	25s	284192	39
titlePrincipals	3m41s	3077490	435
nameBasics	1m55s	1584839	223

Tabela 3.1: Métricas ResourceManager para tabelas em Parquet.

ORC			
Table	**Time**	**MB-seconds**	**vcore-seconds**
titleBasics	1m33s	1236798	174
titleRatings	20s	245580	33
titlePrincipals	6m16s	5272503	745
nameBasics	2m25s	1985334	280

Tabela 3.2: Métricas ResourceManager para tabelas em ORC.

Os valores *MB-seconds* e *vcore-seconds* são métricas que multiplicam o valor agregado de memória e vcores pelo tempo de que a aplicação está a ser executada, em segundos.

Além disso, também podemos observar os logs no *Tez*, para verificar operações de escrita e leitura no HDFS. Temos abaixo alguma informação proveniente desses logs.

Parquet				
Table	**Write Ops**	**Write MB**	**Read Ops**	**Read MB**
titleBasics	41	490751	52	110875
titleRatings	5	22547	8	5253
titlePrincipals	3	1545982	5	271092
nameBasics	3	563768	5	151617

Tabela 3.3: Métricas Tez para tabelas em Parquet.

ORC				
Table	**Write Ops**	**Write MB**	**Read Ops**	**Read MB**
titleBasics	41	178766	52	110875
titleRatings	5	5517	8	5253
titlePrincipals	3	563768	5	271092
nameBasics	3	175023	5	151617

Tabela 3.4: Métricas Tez para tabelas em ORC.

Dos dados apresentados nas tabelas 3.4 e 3.3, os dados relativos a *Read Ops* e *Read MB* estão expostos para confirmar que os dados de entrada são iguais. Observando e comparando os valores de *Write MB* e as métricas do *ResourceManager*, verifica-se que existe um trade-off entre armazenamento e recursos gastos (e consequentemente, tempo de execução). Por exemplo, para a table *titlePrincipals*, verifica-se que os valores de tempo de execução, memória gasta e vcores utilizados são bastante superiores no caso do ORC. No entanto, este escreve 3 vezes menos conteúdo no HDFS. Isto deve-se à utilização de *stripes* pela parte do ORC, pelo que apresenta uma melhor compactação de dados. Dependendo do ambiente, a decisão da estrutura das tabelas poderá ser feita tendo em conta limites computacionais ou de armazenamento. No entanto, ainda só consideramos o carregamento do dataset para o Hive, ainda será necessário verificar as diferenças na execução dos próprios jobs.

Os ficheiros de onde foram retirados todos estes dados estão disponíveis na pasta *output*, pelo que é possível observar mais métricas analisando esses ficheiros.

4 | Jobs

4.1 ShowRddOperations

Este job corresponde à execução das operações sobre RDDs que satisfazem o objetivo 2 do enunciado. Para tal, são executados os métodos sobre RDDs definidos no *package* **RddOperations**. Depois de executadas, o resultado dessas operações é impresso no ecrã, visto que não é pedido que estes resultados sejam guardados em ficheiros.
Para realizar estas operações, as tabelas correspondentes aos datasets são convertidas para RDDs.

```
JavaRDD<Row> titleBasicsWithStartYear = sparkSession
        .table("titleBasics")
        .select("tconst", "primaryTitle", "startYear", "genres")
        .toJavaRDD()
        .filter(row -> !row.isNullAt(2))
        .cache();
JavaRDD<Row> titleRatings = sparkSession
        .table("titleRatings")
        .select("tconst", "averageRating")
        .toJavaRDD();
JavaRDD<Row> titlePrincipals = sparkSession
        .table("titlePrincipals")
        .select("tconst", "nconst", "category")
        .toJavaRDD();
JavaRDD<Row> nameBasics = sparkSession
        .table("nameBasics")
        .select("nconst", "primaryName")
        .toJavaRDD();
```

O RDD *titleBasicsWithStartYear* fica em cache por ser utilizado duas vezes. Esses datasets são complementados com seleção de colunas, de modo a não carregar dados desnecessários.

RddOperations.TopGenres

Operação sobre o RDD *titleBasicsWithStartYear*, que trata de calcular o género mais comum para cada década. Portanto, é obrigatório que os títulos não tenham o campo *startYear* nulo. No caso de TV Series, que possuem endYear, é considerado apenas o startYear, não tendo em consideração a duração da série, apenas a estreia.

RddOperations.SeasonHits

Operação sobre os RDDs *titleBasicsWithStartYear* e *titleRatings*, para calcular o título mais bem classificado para cada ano.

RddOperations.TopActors

Operação sobre os RDDs *titlePrincipals* e *nameBasics*, para calcular os atores que participaram em mais títulos. Estes valores ficam um pouco incomparáveis entre atores que participam em TV Series e atores que não participam, visto que cada episódio conta como uma participação.

4.1.1 Execução

Para executar este job, recorremos mais uma vez ao script *cluster.sh*, pelo seguinte comando:

```
$ ./cluster.sh spark <JAR> Jobs.ShowRddOperations
```

O resultado deste job está junto com o source code, na pasta *output*. Retirando uma parte do output deste job para demonstração, obtemos o seguinte:

```
Top Genres:
    Decade: 1940, Genre: Film-Noir
    Decade: 2010, Genre: Action
    ...

Season Hits:
    Year: 1895, Title: The Waterer Watered, Rating: 7.1
    Year: 1940, Title: Tom and Jerry, Rating: 9.3
    ...

Top 10 Actors:
    Actor: Sameera Sherief, Appearances: 9518
    Actor: Subhalekha Sudhakar, Appearances: 6816
    ...
```

4.2 CreateActorPage

Este job corresponde à execução de operações SQL e sobre RDDs, de forma a gerar um único ficheiro relativo a nome, idade, 10 títulos mais bem classificados, etc.. Para tal, foram geradas 2 operações totalmente em SQL, para recolher as informações *Base* e *Friends*. Existem 2 outras operações que utilizam SQL para agregar alguns dados, que posteriormente são processados por RDDs. As classes que implementam estas operações estão no *package* **ActorPage**. Após a execução destas operações, é efetuado um join sobre o resultados das operações *Base* e *Friends*, pelo campo *nconst*. Sendo estas operações totalmente em SQL, o resultado deste join é ainda um `Dataset<Row>`. Este é posteriormente convertido para um JavaPairRDD, usando como *key* o *nconst*, para depois executar mais um join, desta vez com o JavaPairRDD correspondente a *Hits*. Por fim, é obtido um HashMap a partir do RDD correspondente a *Generation*. No fim de todas estas operações, a informação proveniente do RDD com informação de *Base*, *Friends* e *Hits* é passada para um método que irá escrever esta informação no HBase, juntamente com a informação correspondente da *Generation*.

Actor.Base

Implementa o método *getBaseDataset*, que se trata de uma operação SQL para obter os seguintes atributos:

- nconst
- name
- age
- decade
- active_years
- titles
- avg_ratings

Para além dos atributos pedidos, existem aqui atributos adicionais. O atributo *nconst* será utilizado como *key* para associar o ator à sua informação no HBase, assim como para operações de join com as restantes informações. O atributo *decade* será necessário para identificar qual é a lista *Generation* correspondente.

Actor.Friends

Implementa o método *getFriendsDataset*, que se trata de uma operação SQL para obter os seguintes atributos:

- nbconst
- friends

À semelhança da classe anterior, o atributo *nconst* será utilizado para as operações de join.

Actor.Hits

Implementa o método *getHitsRDD*, que se trata de uma operação sobre RDDs, a partir do resultado de uma query SQL. Essa query obtém os seguintes atributos:

- nconst
- primaryTitle
- averageRating

Posteriormente, esse Dataset é convertido para um RDD, que juntamente com outras operações, resulta num JavaPairRDD que associa a um *nconst* os seus 10 títulos mais bem classificados.

Actor.Generation

Implementa o método *getGenerationRDD*, que se trata de uma operação sobre RDDs, a partir do resultado de uma query SQL. Essa query obtém os seguintes atributos:

- decade
- name
- avg_ratings

Posteriormentem, esse Dataset é convertido para um RDD, seguido de uma operação *groupBy* pela *decade*, ordena os atores pela sua classificação média, e guarda o top 10. No fim destas operações, temos um JavaPairRDD que tem como *key* uma década, e como *value* os top 10 de atores, tendo em conta a média das classificações dos títulos em qual participou.

4.2.1 Esquema de dados actorPage

Como foi referido anteriormente, uma vez realizadas todas as operações e os dados estejam prontos, estes são guardados no HBase, numa tabela com esquema apropriado. Para tal, o cluster necessitou dos componentes opcionais *HBase* e *Zookeeper*. Para guardar os dados, é criada uma tabela com 4 *column families*, correspondentes às alíneas apresentadas no enunciado no objetivo 3. Para criar a tabela, recorre-se ao método *createTableIfNotTaken* em *ActorPage.HBase*, que recebe como argumentos o nome da tabela e uma *collection* de *strings*, correspondente às *column families*:

```
HBase.createTableIfNotTaken(conf,
    "actorPage",
    Arrays.asList("base", "friends", "generation", "hits")
);
```

Posteriormente, cada ator irá associar ao seu *nconst* os seus atributos, preenchendo as colunas na respetiva *column family*. Além disso, após a criação da tabela, podemos usar a *hbase shell* para obter uma descrição da tabela, executando hbase(main):001:0> `describe "actorPage"`:

```
hbase(main):001:0> describe "actorPage"
Table actorPage is ENABLED
actorPage
COLUMN FAMILIES DESCRIPTION
{NAME => 'base', VERSIONS => '1', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS =>
'FALSE',DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0',
REPLICATION_SCOPE => '0', BLOOMFILTER => 'ROW', IN_MEMORY => 'false', COMPRESSION =>
'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536', METADATA => {'EVICT_BLOCKS_ON_CLOSE'
=> 'false', 'CACHE_DATA_ON_WRITE' => 'false', 'CACHE_INDEX_ON_WRITE' => 'false',
'CACHE_BLOOMS_ON_WRITE' => 'false', 'PREFETCH_BLOCKS_ON_OPEN' => 'false'}}
{NAME => 'friends', VERSIONS => '1', ...}}
{NAME => 'generation', VERSIONS => '1', ...}}
{NAME => 'hits', VERSIONS => '1', ...}}
```

ActorPage.Actor

Esta classe implementa um construtor que está preparado para ir buscar informações de um dado ator à tabela actorPage, no HBase, criada pelo job *CreateActorPage*. Tem também implementado o método *main*, que exemplifica instanciar esta classe para o ator Robert Downey Jr.
Portanto, se executarmos `$ ./cluster.sh spark <JAR> ActorPage.Actor`, iremos obter um output semelhante ao abaixo, que foi reduzido para não ocupar demasiado espaço desnecessariamente. Mais uma vez, a versão completa deste output está disponível na pasta *output*:


```
{
    "name": "Robert Downey Jr.",
    "age": 56,
    "activeYears": 36,
    "titles": 196,
    "avgRatings": 6.66,
    "friends": [
        "Cary-Hiroyuki Tagawa",
        "Anton Yelchin",
        "Chris Evans"
    ],
      "generation": {
    "0": "Igor Hajdarhodzic",
    "1": "Matthew Brooks",
    "9": "Stefano Simondo"
  },
  "hits": {
    "0": "Robert Downey, Jr.",
    "1": "SmartLess",
    "9": "Iron Man"
  }
```


4.2.2 Execução

Para executar este job, recorremos mais uma vez ao script *cluster.sh*, pelo seguinte comando:

`$ ./cluster.sh spark <JAR> Jobs.CreateActorPage`

Posteriormente, podemos observar uma entrada no HBase, para verificar a representação dos dados. Por exemplo, podemos voltar a recolher os dados para o ator Robert Downey Jr., pela execução do comando `hbase(main):001:0> get "actorPage", "nm0000375"`:


```
COLUMN                CELL
base:active_years     timestamp=1622656350969, value=\x00$
base:age              timestamp=1622656350969, value=\x008
base:avg_ratings      timestamp=1622656350969, value=\x00\x00\x00\x02\x02\x9A
base:name             timestamp=1622656350969, value=Robert Downey Jr.
base:titles           timestamp=1622656350969, value=\x00\x00\x00\x00\x00\x00\x00\xC4
friends:friends       timestamp=1622656350969, value=Cary-Hiroyuki Tagawa
                      \x09Anton Yelchin\x09Nicole Kidman\x09Joaquim de Almeida
```

```
                    \x09Joe Pantoliano\x09Eddie Marsan
generation:0        timestamp=1622656350969, value=Miriam Fiordeponti
generation:1        timestamp=1622656350969, value=Matthew Brooks
...                 ...
generation:9        timestamp=1622656350969, value=Stefano Simondo
hits:0              timestamp=1622656350969, value=Robert Downey, Jr.
hits:1              timestamp=1622656350969, value=Saving the world one algorithm at a time
...                 ...
hits:9              timestamp=1622656350969, value=The Age of A.I.
```

Como podemos observar, os tipos de dados são preservados. Assim, garantimos uma utilização eficiente de disco.

4.3 Resultados

Nesta secção iremos abordar resultados para ambos os jobs descritos em 4.1 e 4.2. Idealmente, utilizaríamos o Spark History para a monitorização dos jobs, mas esta ferramenta estava com problemas no cluster, e ficava a carregar dados indefinidamente, sem progresso. Portanto, iremos observar algumas métricas provenientes do Yarn ResourceManager:

Job	**Tables**	**Time**	**MB-seconds**	**vcore-seconds**
ShowRddOperations	ORC	2m34s	6461494	454
ShowRddOperations	Parquet	2m07s	8326487	585
CreateActorPage	ORC	7m47s	23448335	1652
CreateActorPage	Parquet	8m1s	23824732	1680

Tabela 4.1: Métricas de ResourceManager para Jobs.

Comparando os resultados, não existem grandes diferenças no job *CreateActorPage*, mas no caso do job *ShowRddOperations*, a utilização de tabelas em ORC resultou num tempo de execução 21% superior. No entanto, tabelas em Parquet resultam 28% mais utilização de memória e CPU.
Por curiosidade, foram também executados os jobs recorrendo apenas à tabela externa, com os dados guardados em ficheiro. Mesmo com 4 workers com 32GB de memória RAM cada, os jobs não conseguiram executar por falta de memória, pelo que é importante a escolha acertada da representação dos dados.

Tendo em consideração os resultados aqui expostos na execução dos jobs, e os resultados expostos na execução das *queries* para carregamento dos dados para o Hive, já é possível tirar algumas conclusões sobre qual a estrutura de tabela mais apropriada, neste contexto. Como os datasets sobre os quais é feito o processamento não possuem um tamanho na grandeza de dezenas de gigabytes ou terabytes, talvez não se justificará a utilização de tabelas em ORC, pelo que demoram mais no carregamento para o Hive, e não apresentam melhorias proporcionais na execução dos jobs. No entanto, caso o armazenamento fosse limitado, já faria mais sentido a utilização desta estrutura para as tabelas.

Mais uma vez, os ficheiros de onde foram retirados estes dados estão disponíveis na pasta *output*.

5 | Automatização da execução das tarefas

De forma a facilitar a execução de todas as tarefas, foi criado um *bash script* (*run.sh*). Basta indicar como argumentos os datasets, e este trata de importar os ficheiros para o HDFS e executar todos os Jobs.

O fluxo de execução deste script pode ser dividido nas seguintes fases:

- Compilar o JAR
- Criar o cluster
- Criar pastas para dados no HDFS
- Copiar dados para as pastas criadas no HDFS
- Executar os jobs Hive
  - titleBasics.hql
  - titleRatings.hql
  - titlePrincipals.hql
  - nameBasics.hql
- Executar os jobs Spark
  - Jobs.ShowRddOperations
  - Jobs.CreateActorPage
  - ActorPage.Actor

Execução

O script verifica se os argumentos fornecidos são ficheiros existentes. A execução deste script deve ser feita com o cluster desligado, pois a intenção deste script é executar tudo o que for necessário, de inicio a fim. No entanto, este *script* não elimina o cluster, pois como os resultados ficam no Hive e no HBase, fica à responsabilidade do utilizador a manipulação final destes ficheiros e a eliminação do cluster. Para executar este script, basta executar um comando semelhante a este:

```
$ ./run.sh <TITLE_BASICS> <TITLE_RATINGS> <TITLE_PRINCIPALS> <NAME_BASICS>
```

6 | Conclusão

Com este trabalho prático, conseguimos aplicar os conhecimentos obtidos nas aulas, e ainda aprofundar estes conhecimentos, como por exemplo, com os desenvolvimentos no HBase, no objetivo 3. Foi possível também ganhar mais experiência no que toca ao *Dataproc*, visto que foi necessária alguma configuração adicional para que houvesse o funcionamento de todos os componentes, nomeadamente com a integração do HBase, com o objetivo de trabalhar com dados num contexto não relacional. Além disso, a execução dos jobs desenvolvidos e carregamento de dados recorrendo a mais do que uma estrutura de tabela no Hive, reitera que não existe uma opção universalmente certa, no que toca a estruturação de dados. Depende da finalidade e das prioridades no contexto dos dados, tendo em conta fatores, como por exemplo, a frequência de atualização de campos ou a inserção de novos dados.