Universidade do Minho

Mestrado em Engenharia Informática

Ciência de Dados

Gestão de Grandes Conjuntos de Dados

João Pimentel A80874
José Carvalho A80424
Rafael Fernandes PG38936
Ricardo Martins A78914

16 de Abril de 2020

Resumo

O presente projeto consistiu no desenvolvimento de métodos para analisar dados armazenados em ficheiros e convertê-los em registos de tabelas *HBase*. Para tal, foram utilizadas as linguagens de programação *Java* e a *framework Hadoop*, mais especificamente *Map Reduce*. Numa fase de análise de resultados, observou-se que a solução implementada cumpriu todos os objetivos propostos, permitindo, ainda, um aumento no desempenho comparativamente a outras soluções possíveis.

Conteúdo

Lista de Figuras

Lista de Extratos

1 Introdução

O presente relatório é o resultado da resolução do primeiro trabalho prático da unidade curricular de Gestão de Grandes Conjuntos de Dados, do perfil de Ciência de Dados. O foco deste trabalho passou por implementar métodos capazes de converter dados armazenados em ficheiros em registos de uma tabela *HBase*. Para tal, teve-se por base a utilização das funcionalidades fornecidas pela linguagem de programação *Java*, bem como pela biblioteca *Hadoop Map Reduce*.

A linha de pensamento base da resolução do projeto passou por encontrar os algoritmos que melhor performance forneciam, bem como uma forma eficiente de ligar os vários passos necessários para completar a tarefa. Deste modo, seria possível obter duas tabelas *HBase*, uma das quais referente a filmes, contendo as suas informações e *rating*, e outra referente a atores. Nesta última, além dos dados pessoais dos mesmos, é possível observar o número total de filmes de cada ator, bem como os seus 3 filmes com melhor cotação.

O grande objetivo deste projeto consistiu no aumento da experiência dos alunos no âmbito do desenvolvimento de aplicações para gerir grandes quantidades de dados. Além disso, o projeto visava a consolidação da utilização de processos de *Map Reduce*, realçando a utilidade e simplicidade desta ferramenta para a resolução deste tipo de problemas.

Nos capítulos seguintes serão demonstrados os problemas, formas de resolução, métodos de desenvolvimento e testes efetuados de modo a obter os resultados ideais para o problema proposto.

2 Análise e Especificação

2.1 Descrição do Projeto

O projeto em questão consistiu no desenvolvimento métodos e classes que extendessem as interfaces de *Map Reduce*, permitindo o povoamento de tabelas *HBase* com dados existentes em ficheiros. No que toca a estes, foram utilizados ficheiros com dados de filmes e atores, existentes no *dataset* público da *IMDb*.

2.2 Especificação de Requisitos

O principal objetivo deste projeto foi povoar duas tabelas *HBase* com diversas informações referentes a filmes e a atores, tirando proveito das interfaces de *Map Reduce* existentes na *framework Apache Hadoop*. Assim, serão apresentados, em seguida, os requisitos associados ao desenvolvimento de forma detalhada:

1. Devem ser carregadas informações de cada filme para uma tabela *HBase*, sendo estas provenientes do ficheiro *title.basics.tsv*;
2. As informações de cada filme incluem as colunas: *EndYear*, *Genres*, *OriginalTitle*, *PrimaryTitle*, *Runtime*, *StartYear*, *TitleType* e *isAdult*;
3. Deve ser carregado o *rating* de cada filme, sendo este proveniente do ficheiro *title.ratings.tsv*;
4. Usando os dados da tabela previamente gerada e os restantes ficheiros do *dataset*, deve ser gerada uma tabela *HBase* com os dados referentes aos atores;
5. Cada registo referente a um ator deve possuir as suas informações pessoais: nome, data de nascimento e morte, se existentes;
6. Um ator deve ter a si associado o número total de filmes em que participou;
7. Cada ator deve ter uma lista com os títulos dos 3 filmes com melhor cotação em que participou.

3 Concepção da Resolução

Ao longo deste capítulo serão apresentados os métodos associados a cada tarefa do projeto desenvolvido, bem como a forma de funcionamento destes, problemas encontrados e forma de resolver os mesmos.

3.1 Filmes

O primeiro objetivo deste projeto prático consistia em ler dados referentes a filmes, povoando uma tabela com os mesmos. Dito isto, foram necessários dois *jobs*, um para carregar a informação associada a um filme e outro para carregar o seu *rating*.

3.1.1 Informação

No que toca à informação, através da leitura do ficheiro *title.basics.tsv*, foi possível observar o formato em que os dados se encontram. As colunas presentes consistiam nas seguintes: *tconst*, *titleType*, *primaryTitle*, *originalTitle*, *isAdult*, *startYear*, *endYear*, *runtimeMinutes* e *genres*. Note-se que estas encontram-se na ordem apresentada e estão divididas por *tabs*, já que se trata de um ficheiro *tab-separated values*.

O primeiro problema a ter em atenção foi o cabeçalho do ficheiro, pelo que, no *mapper* (Extrato 1), caso a chave seja 0, essa linha é ignorada. Caso contrário, são lidos os vários campos, sendo estes inseridos num objeto de *PUT* que é enviado para a tabela previamente criada para alocar os registos dos filmes.

```
@Override
protected void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException{
    // In case its the header line
    if(key.get() == 0)
        return;
    // otherwise
    String[] fields = value.toString().split("\t+");
    Put put = new Put(Bytes.toBytes(fields[0]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("TitleType"), Bytes.
        toBytes(fields[1]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("PrimaryTitle"), Bytes.
        toBytes(fields[2]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("OriginalTitle"), Bytes.
        toBytes(fields[3]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("isAdult"), Bytes.toBytes
        (fields[4]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("StartYear"), Bytes.
        toBytes(fields[5]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("EndYear"), Bytes.toBytes
        (fields[6]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("Runtime"), Bytes.toBytes
        (fields[7]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("Genres"), Bytes.toBytes(
        fields[8]));
    context.write(null, put);
}
```

Extrato 1 - Inserção da informação dos filmes na tabela.

É de destacar que, como cada entrada do ficheiro apenas possui infrmação de um filme, não havendo repetições, a atualização da tabela pode e deve ser efetuada no *mapper*, pelo que não foi necessário definir um *reducer* para este *job*.

3.1.2 *Rating*

No que toca aos *ratings*, foi necessário ler o ficheiro *title.ratings.tsv*. Note-se que a inserção destes na tabela não era obrigatória segundo o enunciado, mas permite obter registos mais completos na

base de dados, bem como possíveis interrogações sobre a mesma com base neste campo.

Tal como no caso anterior, o ficheiro possui uma linha de cabeçalho que não deve ser considerada para povoar a tabela. Além disso, este ficheiro possui 3 colunas, sendo que apenas a primeira e a segunda possuem informação relevante para o problema em estudo, pelo que aterceira coluna deve ser ignorada. Como se pode observar pelo Extrato 2, após a separação dos registos num *array* de *strings*, é gerado um objeto de *PUT* cuja chave associada se encontra na primeira posição do *array* e o valor de *rating* na segunda. Mais uma vez, como não é necessário agrupar os registos de alguma forma, a atualização da tabela é feita diretamente no *mapper*.

```
@Override
protected void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException{
    // In case its the header line
    if(key.get() == 0)
        return;
    // Otherwise
    String[] fields = value.toString().split("\t+");
    Put put = new Put(Bytes.toBytes(fields[0]));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("Rating"), Bytes.toBytes(
        fields[1]));
    context.write(null, put);
}
```

Extrato 2 - Inserção dos *ratings* dos filmes na tabela.

3.2 Atores

Povoada a tabela dos filmes, o passo seguinte consistiu em povoar a tabela dos atores. Assim, utilizando a mesma linha de pensamento dos dois *jobs* previamente mencionados, a informação pessoal de cada ator não apresentou grande mudança. Já o número total de filmes e o *top* 3 implicaram algoritmos mais complexos, como será explicado de seguida.

3.2.1 Informação Pessoal

Através da leitura do ficheiro *name.basics.tsv*, é possível obter a informação pessoal de cada ator, ou seja, o seu nome, data de nascimento e óbito. Tendo em conta a ordem das colunas deste ficheiro, apenas as primeiras 4 colunas possuem informação importante nesta fase, sendo estas as *nconst*, *primaryName*, *birthYear* e *deathYear*.

Foi necessário, mais uma vez, ignorar a primeira linha do ficheiro por se tratar de um cabeçalho. No que toca ao objeto de *PUT*, visto que as datas podem conter valores nulos, não trazendo estes informação, apenas são contabilizadas se o seu valor for diferente de *N*. No entanto, o nome é uma constante para todos os registos, pelo que não é necessário verificar a existência do mesmo, como se pode ver pelo Extrato 3. É de destacar que, apesar de não ser utilizada para o objeto de atualização da tabela, a quarta coluna do ficheiro permite filtrar registos que não sejam atores ou atrizes, sendo que são apenas estes que se pretende armazenar.

```
@Override
protected void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException{
    // In case its the header line
    if(key.get() == 0)
        return;
    // Otherwise
    String[] fields = value.toString().split("\t+");
    // Verify if the entry is an actor or actress
    if(fields[4].contains("actor") || fields[4].contains("actress")){
        Put put = new Put(Bytes.toBytes(fields[0]));
        put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("PrimaryName"), Bytes
            .toBytes(fields[1]));
        // Verify if the Birth Date is Null
        if(!fields[2].equals("\\N"))
```

```
            put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("BirthYear"),
                Bytes.toBytes(fields[2]));
        // Verify if the Death Date is Null
        if(!fields[3].equals("\\N"))
            put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("DeathYear"),
                Bytes.toBytes(fields[3]));
        context.write(null, put);
    }
}
```

Extrato 3 - Inserção da informação pessoal dos atores na tabela.

Tal como anteriormente, apenas foi necessário utilizar um *mapper* e nenhum *reducer*, já que cada ator não é repetido ao longo do ficheiro.

3.2.2 Número Total de Filmes

No que toca ao cálculo do número total de filmes a ideia foi semelhante ao exemplo bastante conhecido de *Map Reduce*, *Word Counter*, aplicado ao ficheiro *title.principals.tsv*. Desta forma, é gerado um contexto em que a chave se trata do identificador do ator e um valor que toma o valor 1, ou seja, um contador. Mais uma vez, para este *mapper* funcionar de forma desejada, a primeira linha do ficheiro deve ser ignorada e é necessário garantir que o registo em análise se trata de um ator, atriz ou alguém que se representou a ele mesmo, como se pode ver pelo Extrato 4.

```
@Override
protected void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException {
    // In case its the header line
    if(key.get() == 0)
        return;
    // Otherwise
    String[] fields = value.toString().split("\t+");
    // Verify if the entry is an actor or actress or plays him/her self
    if(fields[3].contains("actor") || fields[3].contains("actress") || fields[3].
        contains("self")){
        context.write(new Text(fields[2]), new LongWritable(1));
    }
}
```

Extrato 4 - *Mapper* para cálculo do número de filmes por ator.

De modo a obter o valor de filmes em que o ator participou, é necessário agrupar e somar todos os contadores gerados, sendo aqui utilizado o primeiro *reducer* do projeto. Este *reducer* recebe uma chave, correspondendo, neste caso, ao identificador de um ator e a um conjunto de valores, o qual engloba vários valores de *LongWritable* de valor 1, previamente agrupados pela chave graças à *framework*.

Assim, somando todos os valores associados a um determinado ator, é obtido o número de filmes em que este participou, como se pode observar pelo Extrato 5. Neste caso, poderia ser gerado um novo *job* em que se atualizava a tabela, sendo que seria necessário armazenar os dados temporariamente ou podia ser efetuada diretamente a atualização da tabela. De modo a gerar o mínimo de *jobs* e ficheiros temporários possível, a tabela é atualizada no *reducer*, sendo que os filmes de um determinado ator já estão todos contabilizados neste passo.

```
@Override
protected void reduce(Text key, Iterable<LongWritable> values, Context context)
    throws IOException, InterruptedException {
    // Initialize the counter
    int counter = 0;
    for(LongWritable value : values){
        counter += value.get();
    }
    // Update the value in the table
    Put put = new Put(Bytes.toBytes(key.toString()));
```

```
        put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("TotalMovies"), Bytes.
            toBytes(counter));
        context.write(null, put);
    }
```

Extrato 5 - *Reducer* para atualização do número de filmes por ator na tabela.

3.2.3 Top 3 Filmes

Por fim, o último requisito deste projeto era calcular os títulos dos 3 melhores filmes de cada ator. Inicialmente a ideia passou por utilizar os dados existentes nas tabelas, juntamente com os ficheiros, sendo efetuados *GETs* sobre determinados registos. No entanto, esta solução mostrou-se extremamente lenta e foi confirmado com o docente da unidade curricular a pouca eficiência da mesma. Assim, tirando proveito de várias funcionalidades da *framework*, foram efetuados dois *jobs*.

No que toca ao primeiro *job*, este recebe como *input* 3 ficheiros distintos, tratando-se de um job de *shuffle join*. Os ficheiros que recebe são *title.principals.tsv* (relacionar filmes com atores), *title.ratings.tsv* (cotações dos filmes) e *title.basics.tsv* (nomes dos filmes). De modo a permitir que este *join* funcione, cada um dos *mappers* tem de possuir um certo padrão, ou seja, algo que permita ao *reducer* saber de que *mapper* aquele contexto é proveniente. Para isso, foi definida uma classe personalizada *Writable*, chamada *Tuple*. Esta trata-se de um par de valores, sendo que estes possuem todas as propriedades que permitem leitura e escrita em ficheiros com auxílio dos *mappers* e *reducers*.

Assim, tenha-se o tuplo *("Rating", 9.0)* associado ao filme *tt000001*. Neste é possível observar que o filme *tt000001* possui um *rating* de 9.0. Desta forma, gerando este tipo de contextos associados com os tuplos, o *reducer* vai receber conseguir associar os vários atores, o nome do filme e o seu *rating*, com base no identificador do filme. É de destacar que, como o processo não pode ser finalizado nesta fase, é necessário um *job* extra. Além disso, como se pode observar pelo Extrato 6, apenas se gera um contexto se nenhum dos valores for nulos, já que não faz sentido essas entradas serem consideradas para o cálculo do *top* 3. De igual modo, têm de ser geradas várias entradas, uma por cada ator que participou no filme em questão, além do seu nome e cotação.

```
@Override
protected void reduce(Text key, Iterable<Tuple> values, Context context) throws
    IOException, InterruptedException {
    String rating = "", name = "";
    List<String> actors = new ArrayList<>();

    for(Tuple value: values) {
        // In case its an entry in the format: Movie Rating
        if (value.getKey().toString().equals("Rating")){
            rating = value.getValue().toString();
        }
        // In case its an entry in the format: Movie Actor
        else if (value.getKey().toString().equals("Actor")){
            String act = value.getValue().toString();
            actors.add(act);
        }
        // In case its an entry in the format: Movie Name
        else if (value.getKey().toString().equals("Name")){
            name = value.getValue().toString();
        }
    }
    // There are N actors in each movie
    for(String actor : actors){
        // Actor (Name, Rating) -> Values can't bee null
        if(!actor.equals("") && !rating.equals("") && !name.equals(""))
            context.write(new Text(actor), new Tuple(new Text(name), new Text(
                rating)));
    }
}
```

Extrato 6 - *Reducer Join* para associar nome e *rating* do filme com os atores.

Finalizado o primeiro *job*, o formato dos contextos é algo semelhante a *Ator (Nome do Filme, Rating)*. Tendo em conta que se pretende agrupar os vários filmes em que cada ator participa, é necessário realizar um *mapper* de identidade. Desta forma, os filmes de cada ator serão agrupados no *reducer*, permitindo o cálculo do *top* 3. Note-se que foram utilizados *Sequence Files* para armazenar temporariamente os dados do *job* anterior e serem utilizados como argumento do segundo. Isto permite que não seja necessário um *parsing* das estruturas previamente definidas, ou seja, a chave já tem a si associada o identificador do ator e os valores tratam-se de tuplos com nome e cotação de filmes, como se pode observar pelo Extrato 7.

```
@Override
protected void map(Text key, Tuple value, Context context) throws IOException,
    InterruptedException {
    // Identity
    context.write(key, value);
}
```

Extrato 7 - *Mapper* identidade para cálculo do *top* 3.

No que toca ao *reducer*, como mencionado anteriormente, a ideia deste é agrupar os filmes em que um determinado ator participou, permitindo o cálculo dos 3 filmes com melhor cotação do mesmo. Tendo em conta as funcionalidades da *framework*, é sabido que os filmes já se encontram agrupados pelo identificador do ator nesta fase. Assim, apenas é necessário a ordenação da lista com os filmes e consequente criação de uma sublista de dimensão 3 ou inferior, caso o ator não tenha participado, no mínimo, em 3 filmes.

Observando o Extrato 8, é possível constatar que, utilizando uma classe *Pair*, semelhante à classe *Tuplo*, diferindo no facto de esta não ser *Writable*, os vários pares de filmes e cotações são armazenados numa lista. Estando a lista completa, esta é ordenada pelo valor das cotações de cada par, sendo que os valores maiores se encontram nos índices mais pequenos da lista. Assim, aplicando o método *subList*, os três melhores filmes do ator são obtidos, tendo em consideração que, no caso da dimensão da lista previamente formada ser inferior a 3, utiliza-se a lista gerada. É, ainda, de notar que a lista é armazenada no formato de *string*, permitindo a fácil leitura e inclusão em projetos escritos em linguagens que não *Java*. Também é importante mencionar que a aplicação do método *toString* na lista aplica este mesmo método a cada um dos elementos, sendo que, pela definição da classe *Pair*, a lista ficará formada pelos nomes dos filmes, omitindo o seu *rating*.

```
@Override
protected void reduce(Text key, Iterable<Tuple> values, Context context) throws
    IOException, InterruptedException {
    // Initialize the list for the actor
    List<Pair<String, Float>> movies = new ArrayList<>();
    for(Tuple value: values) {
        // Each entry in the list has to be considered
        movies.add(new Pair<>(value.getKey().toString(), Float.parseFloat(value.
            getValue().toString())));
    }
    // Sort the list by the rating
    movies.sort((a, b) -> b.getValue().compareTo(a.getValue()));
    // get the top 3 from the list
    List<Pair<String, Float>> top3 = movies.subList(0, movies.size() >= 3 ? 3 :
        movies.size());
    // Convert the list to string, this way it can be easily used in an non java
        application
    String top_movies = top3.toString();
    // Update the value in the table
    Put put = new Put(Bytes.toBytes(key.toString()));
    put.addColumn(Bytes.toBytes("Details"), Bytes.toBytes("Top3Movies"), Bytes.
        toBytes(top_movies));
    context.write(null, put);
}
```

Extrato 8 - *Reducer* para cálculo do *top* 3 e atualização da tabela.

Desta forma, tal como no cálculo do número total de filmes por ator, a tabela é atualizada direamente no *reducer*, permitindo que não seja efetuado outro *job* e consequente escrita em memória dos dados.

4 Análise de Métricas

De modo a confirmar que os algoritmos implementados trouxeram uma melhoria na performance das tarefas, face à resolução do Guião número 4 da Unidade Curricular, os tempos de execução foram comparados como se pode observar pela Tabela 1. Nesta é visível que, o cálculo e carregamento dos dados referentes ao número total de filmes é sensivelmente 20% mais rápido na versão implementada no projeto prático. Os algoritmos diferem no facto de que no Guião, a ideia foi associar uma lista de identificadores de filmes a um determinado ator, sendo o valor a colocar na base de dados o tamanho desta lista. Tendo em conta a redução no tamanho dos dados em estudo, já que neste caso os valores tratam-se de contador e não uma lista de *strings*, a redução no tempo de execução era expectável.

ambém é importante analisar os dados referentes aos cálculo dos filmes com melhor cotação. No que toca à resolução implementada no Guião, não eram utilizados *Sequence Files*, mas sim ficheiros de texto. Aliado a isto, em vez de os dados serem armazenados em tuplos entre *mappers* e *reducers*, estes eram guardados em formato de *string*, sendo sempre necessário efetuar um *parsing* da mesma. Dito isto, é possível observar que existe uma redução próxima de 10% no tempo de execução da tarefa. Note-se que o tempo de execução do Guião não calcula os nomes dos filmes, mas sim os seus identificadores, pelo que o tempo de execução seria ainda maior, implicando a execução de mais um *mapper* sobre um determinado ficheiro.

É de realçar que em ambas as situações comparadas, no Guião era efetuado um *mapper* extra após o *reducer* em que se atualizava a tabela, contrariamente ao efetuado no projeto, onde a tabela é atualizada diretamente no *reducer*.

Tabela 1 - Alguns registos do *dataset*.

	gzipped	**unzipped**	**Guião 4 - unzipped**
Informação Pessoal	33762.0ms	30724.0ms	138930.0ms
Total de Filmes	126035.0ms	131794.0ms	162807.0ms
Top 3 Filmes	182546.0ms	188460.0ms	207175.0ms

Note-se que não foram comparados os tempos para o povoamento da tabela de filmes, pois os algoritmos são exatamente iguais quer no Guião, quer neste projeto.

Além disso, é importante mencionar que a utilização de ficheiros sem estar no formato *gzip* permite a execução paralela de vários passos dos *jobs*, reduzindo a carga de trabalho em cada *core*, já que os ficheiros *gzip* não são separáveis.

5 Utilização da Solução

É, também, importante explicar o funcionamento da solução desenvolvida. Assim, o primeiro passo é inicializar o servidor *HBase*. Para isso, é necessário executar o comando **docker-compose -f docker-compose-distributed-local.yml up** na diretoria *docker-hbase*.

Tendo o servidor a correr, o passo seguinte é criar uma pasta para colocar ficheiros de *input* e outra para os ficheiros temporários associados ao *job* do cálculo dos 3 filmes com melhor cotação. Assim, é necessário executar, na mesma diretoria previamente mencionada, os seguintes comandos: **docker run –network docker-hbase_default –env-file hadoop.env bde2020/hadoop-base hdfs dfs -mkdir /input** e **docker run –network docker-hbase_default –env-file hadoop.env bde2020/hadoop-base hdfs dfs -mkdir /output**.

O próximo passo consiste em inserir os ficheiros na pasta de *input*, sendo que para o fazer deve ser executado o comando **docker run –network docker-hbase_default –env-file hadoop.env –mount type=bind,source=PATH_TO_DATA/Dados,target=/data bde2020/hadoop-base hdfs dfs -put /data/FICHEIRO /input**, em que *PATH_TO_DATA* representa o *path* até à pasta onde os dados estão armazenados e *FICHEIRO* o nome do ficheiro a colocar no *container* do *docker*.

No que toca à execução do código *Java*, tirando proveito do *IDE IntelliJ*, devem ser gerados os binários, criada uma imagem para o *docker file* e, por fim, este pode ser executado. Note-se que devido à forma como foi desenvolvido o projeto, é necessário alterar no *Dockerfile* qual a classe que se pretende executar. É, também, importante mencionar que de modo a executar partes das tarefas pedidas, é necessário comentar *jobs* na função *main*, uma vez que o projeto foi desenvolvido com intuito de executar cada alínea como uma só. Além disso, devido às restrições nas capacidades de memória das máquinas possuídas pelos alunos, é mais seguro executar as tarefas uma a uma.

Por fim, para confirmar o povoamento das tabelas, pode ser executado o comando **docker run -it –network docker-hbase_default –env-file hbase-distributed-local.env bde2020/hbase-base hbase shell**, sendo gerada uma *shell HBase*. Nesta podem ser executados *scans*, *gets*, entre outros comandos. A título de exemplo observe-se a Figura 1 que representa um registo referente a um filme.

```
get 'movies_tp1', 'tt0000001'
COLUMN                CELL
 Details:EndYear      timestamp=1585839000677, value=\x5CN
 Details:Genres       timestamp=1585839000677, value=Documentary,Short
 Details:OriginalTitl timestamp=1585839000677, value=Carmencita
 e
 Details:PrimaryTitle timestamp=1585839000677, value=Carmencita
 Details:Rating       timestamp=1585839400106, value=5.6
 Details:Runtime      timestamp=1585839000677, value=1
 Details:StartYear    timestamp=1585839000677, value=1894
 Details:TitleType    timestamp=1585839000677, value=short
 Details:isAdult      timestamp=1585839000677, value=0
9 row(s) in 0.1170 seconds
```

Figura 1 - Registo de um filme na tabela *HBase*.

No que toca aos atores, observe-se a Figura 2, em que se pode constatar uma atriz sem data de óbito, a qual nasceu em 1934 e os seus 3 filmes com melhor cotação correspondem a *The Passions of Louis Malle*, *Bridgitte Bardot: ANimal Attraction* e *Contempt*.

```
get 'actors_tp1', 'nm0000003'
COLUMN                CELL
 Details:BirthYear    timestamp=1586945290858, value=1934
 Details:PrimaryName  timestamp=1586945290858, value=Brigitte Bardot
 Details:Top3Movies   timestamp=1586946006534, value=[The Passions of Louis Mall
                      e, Brigitte Bardot: Animal Attraction, Contempt]
 Details:TotalMovies  timestamp=1586945721242, value=\x00\x00\x00\x7F
4 row(s) in 0.0210 seconds
```

Figura 2 - Registo de uma atriz na tabela *HBase*.

6 Conclusões e Trabalho Futuro

Ao longo do presente relatório encontra-se representado o resultado do primeiro trabalho prático da unidade curricular de Gestão de Grandes Conjuntos de Dados. Neste contexto foram abordados os passos referentes ao desenvolvimento e implementação de vários métodos que permitissem converter dados de filmes e atores em ficheiros em duas tabelas *HBase*. Foi, desta forma, necessário pensar nos algoritmos mais apropriados, bem como a melhor forma de tirar proveito da *framework Hadoop*.

O desenvolvimento do projeto permitiu aprimorar os conhecimentos previamente adquiridos e postos em prática aquando da realização do quarto Guião da Unidade Curricular. Desta forma, foi possível pensar nos algoritmos numa perspetiva de otimização e não apenas conseguir realizar as tarefas. Algumas das alterações envolveram a atualização direta das tabelas em *reducers*, contrariamente à execução de um novo *job* cujo único objetivo era essa atualização, a utilização de *Sequence Files*, sendo que estes mantêm os tipos de dados associados, além de serem ficheiros separáveis. Esta caraterística permite a execução paralela em vários *cores*, dividindo-se, assim, a carga computacional pelos recursos disponíveis.

Outra grande diferença implementada foi a utilização de classes personalizadas para armazenar os dados, ao invés do que tinha sido aplicado no Guião, em que se geravam *strings* com determinados padrões, sendo depois necessário analisar as mesmas para retirar a informação útil. Assim, as alterações implementadas permitiram um aumento na performance geral das tarefas propostas, graças ao maior cuidado com a qualidade dos algoritmos

Em suma, a realização deste trabalho exigiu a aplicação de todos os conhecimentos lecionados em contexto de aula no que toca a ambas as unidades curriculares, permitindo que o grupo cumprisse todos os objetivos propostos no enunciado, conciliando assim a teoria e a prática.