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

30 de Maio de 2020

Resumo

O presente projeto consistiu no desenvolvimento de métodos para analisar dados armazenados em ficheiros e provenientes de um gerador de votos, tratamento dos mesmos, de modo a serem obtidas informações pertinentes. Para tal, foram utilizadas as linguagens de programação *Java* e a *framework Spark*. Numa fase de análise de resultados, observou-se que a solução implementada cumpriu todos os objetivos propostos, bem como solidificar os benefícios de computação distribuída para tratamento de grandes conjuntos de dados.

Conteúdo

Lista de Figuras

Lista de Extratos

1 Introdução

O presente relatório é o resultado da resolução do segundo 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 gerir e tratar dados provenientes de uma aplicação geradora de votos sobre filmes, bem como métodos de processamento em *batch* sobre ficheiros estáticos. Para tal, teve-se por base a utilização das funcionalidades fornecidas pela linguagem de programação *Java*, bem como pela biblioteca *Spark*.

A linha de pensamento base da resolução do projeto passou por encontrar os melhores algoritmos para os problemas, ou seja, algoritmos que tirassem proveito da computação paralela fornecida pelo *Spark*. Deste modo, seria possível executar várias tarefas sobre grandes quantidades de dados, sem um peso computacional extremamente elevado por máquina do *cluster*.

O grande objetivo deste projeto consistiu no aumento da experiência dos alunos no âmbito do desenvolvimento de aplicações para gerir grandes quantidades de dados, especialmente num contexto em que novos dados vão chegando constantemente à aplicação. Além disso, o projeto visava a consolidação da utilização de processos de *Spark*, quer em contexto de *streaming*, quer em contexto de processamento em *batch*, 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 de métodos e de classes capazes de extrair informação proveniente de dados gerados por uma aplicação de votos e de ficheiros estáticos, existentes no *dataset* público da *IMDb*.

2.2 Especificação de Requisitos

O principal objetivo deste projeto foi tratar da maior quantidade de dados provenientes de um contexto de *streaming*, tirando proveito da biblioteca *Spark*. Além disso, eram pedidas as execuções de algumas interrogações sobre ficheiros estáticos, utilizando, também, a biblioteca em questão. Assim, serão apresentados, em seguida, os requisitos associados ao projeto de forma detalhada:

1. Devem ser gerados ficheiros de *log* a cada 10 minutos sobre os dados que chegam à aplicação, no contexto de *streaming*. Além disso, cada um destes registos deve ter a si associada uma marcação do momento de chegada, truncada aos minutos;
2. Devem ser apresentados, a cada minuto, os títulos dos filmes que obtiveram melhor classificação média nos últimos 10 minutos;
3. A cada 15 minutos, devem ser apresentados os títulos dos filmes cujo número de votos no intervalo atual seja maior do que no intervalo anterior, independentemente do valor dos votos;
4. Devem ser calculados os 10 atores que participaram em mais filmes distintos, sendo este um método de tratamento de dados em *batch*;
5. Deve ser possível efetuar o cálculo dos colaboradores dos atores, ou seja, associar a cada ator os restantes atores que participaram nos mesmos filmes;
6. Deve ser criado uma pasta com os novos valores de *ratings* dos filmes, tendo por base o ficheiro anterior, bem como os ficheiros de *logs* gerados até ao momento;
7. Devem ser testadas várias configurações, para um mesmo *hardware*, que permitam receber e tratar o maior débito de eventos.

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 *Streaming*

A primeira componente do projeto baseava-se na definição de métodos capazes de tratar de dados provenientes de uma aplicação que gerava votos sobre os filmes em tratamento. Desta forma, em seguida serão apresentados os diversos objetivos desta fase do projeto. Note-se que estas interrogações funcionam em simultâneo.

3.1.1 *Log*

O primeiro objetivo relativo ao contexto de *streaming* era a geração de ficheiros de *log* a cada 10 minutos, sendo que os dados deveriam ter a marcação do seu momento de chegada aproximado ao minuto. Assim, tirando proveito do método *transform*, com a variável tempo associada a cada *rdd*, foi possível efetuar esta marcação. Observando o Extrato 1, note-se que a janela para gerar uma pasta com os ficheiros do intervalo é definida depois desta transformação. Além disso, tendo em conta que cada nodo trata de vários registos de forma paralela aos restantes nodos, cada um destes nodos gera um ficheiro distinto, sendo que todos eles ficam contidos na mesma diretoria.

```
int windowA = 10;
input.transform(
    (rdd, time) ->
        rdd.map(s -> s + "\t" + new Timestamp(time.milliseconds()).toLocalDateTime
            ().truncatedTo(ChronoUnit.MINUTES).toString()))
    .window(Durations.minutes(windowA), Durations.minutes(windowA))
    .foreachRDD(rdd -> {
        String folder = LocalDateTime.now().truncatedTo(ChronoUnit.MINUTES).
            toString().replace(":", "-");
        rdd.saveAsTextFile("hdfs://namenode:9000/logs/" + folder);
    });
```

Extrato 1 - Algoritmo para gerar ficheiros de *log* a cada 10 minutos.

3.1.2 *Top* 3 Filmes

Já no caso do cálculo dos 3 filmes com melhor cotação média nos últimos 10 minutos, a ideia era apresentar no ecrã este valor a cada minuto. Desta forma, foi definida uma janela de tempo com tamanho de 10 minutos e *slide* de 60 segundos, ou seja, a cada minuto era efetuado o cálculo, sendo utilizados os dados presentes nos últimos 10 minutos. No que toca ao algoritmo, a ideia passou por gerar pares do tipo **(Filme, (voto, 1))**, sendo estes reduzidos por chave e janela, somando-se todos os votos e todos os contadores de valor 1. Assim, apenas foi necessária a aplicação de um mapeamento em que o segundo elemento do par correspondia à divisão entre o valor da soma dos votos sobre o contador, como se pode ver pelo Extrato 2.

Por fim, tendo em conta que eram pretendidos os títulos dos filmes e não os seus identificadores, foi efetuado um *join* com um *rdd* previamente calculado e armazenado em *cache*, de modo a reduzir o custo computacional da operação. Após o *join*, os tuplos foram mapeados para ficar com o nome e a cotação do filme, sendo retirados de forma ordenada os 3 filmes com melhor cotação com auxílio do método *takeOrdered* e um comparador de tuplos definido para os ordenar da maior cotação para a menor. Note-se que, como se trata de uma operação em que é pretendido o cálculo do *top* aplicado a todos os dados em questão, foi necessária a utilização de um método que coleciona dados de vários nodos, como o *takeOrdered*.

```
int windowB = 10;
int slideB = 60;
```

```
input.window(Durations.minutes(windowB),Durations.seconds(slideB))
    .map(t -> t.split("\t+"))
    .mapToPair(v -> new Tuple2<>(v[0], new Tuple2<>(Double.parseDouble(v[1]), 1)))
    .reduceByKeyAndWindow((Function2<Tuple2<Double, Integer>, Tuple2<Double,
        Integer>, Tuple2<Double, Integer>>) (i1, i2) -> new Tuple2<>(i1._1 + i2._1,
         i1._2 + i2._2), Durations.minutes(10), Durations.seconds(60))
    .mapToPair(v -> new Tuple2<>(v._1, v._2._1 / v._2._2))
    .foreachRDD(rdd -> {
        List<Tuple2<String, Double>> res = rdd
            .join(movies)
            .mapToPair(t -> new Tuple2<>(t._2._2,t._2._1))
            .takeOrdered(3, new ComparadorTuplos());

        System.out.println(ANSI_GREEN + "TOP␣3:␣" + res.toString() + ANSI_RESET);
    });
```

Extrato 2 - Algoritmo para cálculo dos 3 filmes com melhor cotação média nos últimos 10 minutos.

3.1.3 *Trending*

O último objetivo em contexto de *streaming* era o cálculo dos filmes em destaque, ou seja, os filmes que no intervalo de tempo atual tiveram mais votos do que no intervalo anterior, independentemente do valor desses votos. Ora, o primeiro passo foi a definição da janela, possuindo esta uma dimensão e um *slide* iguais e com dimensão de 15 minutos. Em seguida, cada registo foi mapeado com o seu identificador e um contador inicializado a 1, sendo que, logo após, foi aplicada uma redução para calcular o número de votos por identificador. De modo a comparar os valores do intervalo atual com o intervalo anterior, foi utilizado o método *updateStateByKey*, em que é possível obter um valor presente no *checkpoint* referente à chave em questão no intervalo de tempo anterior. Desta forma, caso a lista de novos valores, ou seja, a lista que possui os vários valores para aquela chave nos vários nodos do *cluster*, seja vazia, significa que o filme em questão não teve votos neste intervalo. Tal implica que não está em estado de *trend*, sendo atualizado o seu estado com o valor 0.

No caso de existirem novos valores, primeiramente, é calculado o valor do intervalo anterior. Caso este exista, é armazenado, senão é atribuído o valor de 0 ao intervalo anterior. Em seguida, são somados todos os novos contadores, obtendo-se, assim, o total de votos para um determinado filme no intervalo atual. Caso o valor de votos atual seja maior que o anterior, o valor é armazenado como o novo valor do estado daquela chave. Caso contrário, é armazenado o valor simétrico do valor atual, permitindo ter uma indicação do valor de votos no intervalo para a próxima análise. Assim, torna-se mais simples de filtrar os filmes em estado de *trend*. Dito isto, a comparação do valor atual é efetuada com o módulo do valor anterior, como se pode observar pelo Extrato 3.

Por fim, apenas é necessário filtrar pelos valores positivos, já que se tratam dos filmes em voga. Tal como a situação anterior, são pedidos os títulos destes filmes, pelo que é efetuado um *join* com o *rdd* previamente calculado, sendo impressos os vários pares de títulos de filmes e números de votos no intervalo atual.

```
int windowC = 15;
input.window(Durations.minutes(windowC),Durations.minutes(windowC))
    .map(t -> t.split("\t+"))
    .mapToPair(v -> new Tuple2<>(v[0],  1))
    .reduceByKey(Integer::sum)
    .updateStateByKey((List<Integer> values, Optional<Integer> current) -> {
        if (values.isEmpty())
            return Optional.of(0);
        else {
            int old = current.or(0);
            int novo = 0;
            for(Integer i : values)
                novo += i;
            // Se o valor destes 15 minutos for maior do que nos 15 min passados, o
                 valor e positivo
            // caso contrario, e guardado o valor negativo para filtrar em seguida
```

```
            return novo > Math.abs(old) ? Optional.of(novo) : Optional.of(-novo);
        }
    })
    .filter(p -> p._2 > 0)
    .foreachRDD(rdd -> {
        List<Tuple2<String, Integer>> res = rdd
            .join(movies)
            .mapToPair(t -> new Tuple2<>(t._2._2,t._2._1))
            .collect();
        System.out.println(ANSI_GREEN + "Trending: " + res.toString() + ANSI_RESET)
            ;
    });
```

Extrato 3 - Algoritmo para cálculo dos filmes em destaque a cada 15 minutos.

3.2 *Batch*

A segunda componente do projeto baseava-se na definição de métodos para tratamento de dados em ficheiros, de modo a obter informações sobre os mesmos. Assim, tendo em conta que cada uma destas ações não dependia das restantes, foram definidas 3 classes, uma por problema.

3.2.1 *Top* 10 Atores

O primeiro problema era o cálculo dos 10 atores que participaram em mais filmes. Assim, o algoritmo passou por filtrar o cabeçalho do ficheiro, bem como filtrar os registos que não representassem um papel de ator/atriz ou a si próprios numa determinada produção. Em seguida, cada par **(Filme, Ator)** foi convertido em **(Ator, 1)**, sendo este 1 um contador dos filmes em que o ator participou. Desta forma, reduzindo estes pares pelo identificador do ator e somando os valores, foi obtido o par **(Ator, Número de Filmes)** como pretendido e como se pode ver pelo Extrato 4. De modo a ordenar estes pares, foi invertida a ordem do par, ou seja, passou a ser **(Número de Filmes, Ator)**, permitindo a aplicação do método *sortByKey* com ordenação descendente. Por fim, os registos foram mapeados de volta ao seu estado original, sendo a chave o identificador do ator e não o seu número de filmes, mantendo-se a ordenação, sendo efetuado um *take* de 10 unidades.

```
JavaPairRDD<String, Integer> movies_by_actor = sc.textFile("hdfs://namenode:9000/
    input/title.principals.tsv")
    .map(l -> l.split("\t+"))
    .filter(l -> !l[0].equals("tconst"))
    .filter(l -> l[3].contains("actor") || l[3].contains("actress") || l[3].
        contains("self"))
    .mapToPair(l -> new Tuple2<>(l[2], 1))
    .foldByKey(0, Integer::sum)
    .mapToPair(p -> new Tuple2<>(p._2, p._1))
    .sortByKey(false)
    .mapToPair(p -> new Tuple2<>(p._2, p._1))
    .cache();

List<Tuple2<String, Integer>> top10 = movies_by_actor.take(10);
```

Extrato 4 - Algoritmo para cálculo dos 10 atores com maior número de filmes.

3.2.2 *Friends*

No que toca ao cálculo dos colaboradores de cada ator, a ideia passou por calcular os pares **(Filme, Ator)**, apenas para os registos que tivessem passado a filtragem do seu papel no filme, sendo estes resultados armazenados em *cache*, como se pode observar pelo Extrato 5. Em seguida, foi efetuado um *join* do *rdd* consigo mesmo, permitindo obter uma espécie de produto cartesiano, ou seja, o valor do par passa agora a ser um par de dois atores. Tendo em conta que esta estratégia leva a que um ator seja relacionado consigo mesmo, foi necessário filtrar os pares cujo primeiro e elemento fossem iguais. Por fim, os registos foram mapeados de forma a que o identificador do filme fosse ignorado, ficando-se apenas com pares do tipo **(Ator, Ator)**, apenas sendo necessária

a agrupação pela chave destes pares para obtenção do resultado final. De modo a tirar o máximo proveito da computação distribuída fornecida pelo *Spark*, em vez da chamada do método *collect*, levando a que todos os dados tivessem que ser enviados para um só nodo, foi utilizado o sistema de ficheiros distribuído, ou seja, os vários nodos guardaram os registos em ficheiro, estando todos estes numa mesma pasta.

```
// Filme Ator
JavaPairRDD<String, String> pares = sc.textFile("hdfs://namenode:9000/input/title.
    principals.tsv")
    .map(l -> l.split("\t+"))
    .filter(l -> !l[0].equals("tconst"))
    .filter(l -> l[3].contains("actor") || l[3].contains("actress") || l[3].
        contains("self"))
    .mapToPair(l -> new Tuple2<>(l[0], l[2]))
    .cache();

// Ator [Ator]
pares.join(pares)
    .filter(p -> !p._2._1.equals(p._2._2))
    .mapToPair(p -> p._2)
    .groupByKey()
    .saveAsTextFile("hdfs://namenode:9000/output/collaborators/");
```

Extrato 5 - Algoritmo para cálculo dos colaboradores de cada ator.

3.2.3 *Ratings*

No que toca ao cálculo dos *ratings* atuais, tendo por base os votos presentes nos *logs*, a ideia foi calcular separadamente um *rdd* para a leitura do ficheiro já existente e um para a leitura e cálculo com base nos valores presentes nos ficheiros de *log*.

Assim, no que toca ao ficheiro original, depois de ignorado o cabeçalho, foram criados pares **(Filme, (Cotação, Número de Votos))**, como se pode observar pelo Extrato 6. Já no que toca aos *logs*, o resultado final pretendido era o mesmo, ou seja, um par cuja chave fosse o identificador do filme e o valor fosse um par com a cotação e o número de votos, permitindo a fácil junção e cálculo dos novos valores. Desta forma, cada voto foi convertido para um registo na forma **(Filme, (Cotação, 1))**, sendo este 1 um valor utilizado como contador de votos. Em seguida, foi efetuada uma redução pelas chaves, onde se somavam todos os votos e todos os contadores, permitindo, assim, o cálculo da média. Por fim, de modo a atualizar os dados dos *ratings*, foi necessário efetuar um *join* do *rdd* dos *logs* ao original, sendo efetuado um mapeamento aos registos em que o novo valor do total de votos era a soma do total original com o total vindo do *rdd* dos *logs*.

No que toca ao novo valor da cotação, este foi definido como sendo a divisão da soma do número de votos multiplicada pela cotação presente em cada *rdd* pelo total de votos, como se pode observar no final do Extrato 6. Por fim, estes dados são armazenados em ficheiros, na pasta *title_ratings_new* do sistema de ficheiros do distribuído do *HDFS*, após uma conversão para o formato *tsv* por questões de simplicidade de leitura do ficheiro.

```
// Movie (Score, Number of Votes)
JavaPairRDD<String, Tuple2<Double, Integer>> movies_by_actor = sc.textFile("hdfs://
    namenode:9000/input/title.ratings.tsv")
    .map(l -> l.split("\t+"))
    .filter(l -> !l[0].equals("tconst"))
    .mapToPair(l -> new Tuple2<>(l[0], new Tuple2<Double, Integer>(Double.
        parseDouble(l[1]), Integer.parseInt(l[2]))))
    .cache();

// Movie (Score, Number of Votes)
JavaPairRDD<String, Tuple2<Double, Integer>> logs = sc.textFile("hdfs://namenode
    :9000/logs/*")
    .map(l -> l.split("\t+"))
    .mapToPair(l -> new Tuple2<>(l[0], new Tuple2<>(Double.parseDouble(l[1]), 1)))
    .reduceByKey(
```

```
        (Function2<Tuple2<Double, Integer>, Tuple2<Double, Integer>, Tuple2<Double,
            Integer>>)
            (i1, i2) -> new Tuple2<>(i1._1 + i2._1, i1._2 + i2._2))
    .mapToPair(v -> new Tuple2<>(v._1, new Tuple2<>(v._2._1 / v._2._2, v._2._2)))
    .cache();

// Merge e calculo da nova media e novo total de votos
movies_by_actor.join(logs)
    .mapToPair(p -> new Tuple2<>(p._1, new Tuple2<>((p._2._1._1 * p._2._1._2 + p._2
        ._2._1 * p._2._2._2) / (p._2._1._2 + p._2._2._2), p._2._1._2 + p._2._2._2))
        )
    .map(p -> p._1 + "\t" + p._2._1 + "\t" + p._2._2)
    .saveAsTextFile("hdfs://namenode:9000/output/title_ratings_new/");
```

Extrato 6 - Algoritmo para cálculo dos *ratings* atuais.

4 Análise de Métricas

4.1 Arquitetura do *Cluster*

O *cluster* utilizado para testar a solução desenvolvida é constituído por 3 máquinas, tendo cada uma destas o sistema operativo *centOS-7*. Além disso, cada uma das máquinas possui 2 *vCPUs* e 7.5 *GB* de memória. É, também, importante mencionar que todas elas possuem um disco *SSD*, tendo este uma dimensão de 100 *GB*.

4.2 *Streaming*

Tendo em conta a impossibilidade de avaliar os algoritmos definidos para contexto de *streaming* com base em tempos de execução, foi definido que esta avaliação seria feita com base em uso percentual do *CPU*. Assim, em seguida serão apresentados gráficos em que se executam, em simultâneo, as 3 tarefas deste contexto, com as janelas pretendidas, por um período de 30 minutos, ou seja, permitindo a execução de 30 cálculos do *top* 3 filmes, 3 *logs* e 2 *trending*.

Note-se, ainda, que em cada um dos gráficos, a linha laranja representa a máquina *master*, a verde representa a máquina *worker1* e a roxa o *worker2*.

Numa primeira fase, a ideia consistiu em testar os algoritmos sem limitação do número de núcleos, ou seja, todos os núcleos eram utilizados de forma a distribuir a carga de execução. Como se pode observar pela Figura 1, permitindo um uso de até 4 *GB* de memória por parte do *Spark*, a carga do *CPU*, em média, ronda os 30-40% nos nodos trabalhadores, permitindo um tratamento de dados bastante elevado.

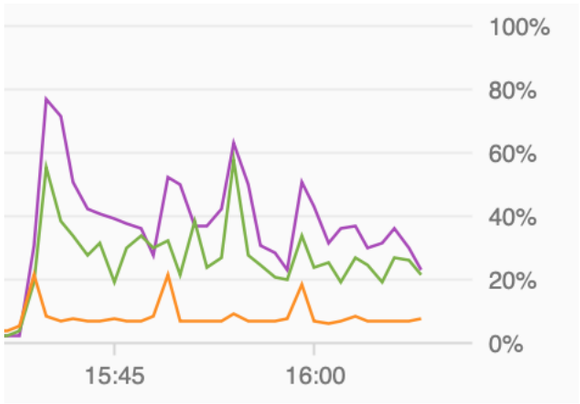

Figura 1 - Uso de *CPU* com 4 *GB* de memória.

Em seguida, de modo a confirmar que um valor relativamente elevado de memória alocada a este tipo de problemas era apropriado, sem se comprometer a execução das restantes tarefas nas máquinas, a memória fornecida ao *Spark* foi reduzida para apenas 2 *GB*. Observando-se a Figura 2, é notório que a carga computacional nos *workers* subiu bastante comparativamente à experiência anterior, não permitindo o tratamento de tantos dados.

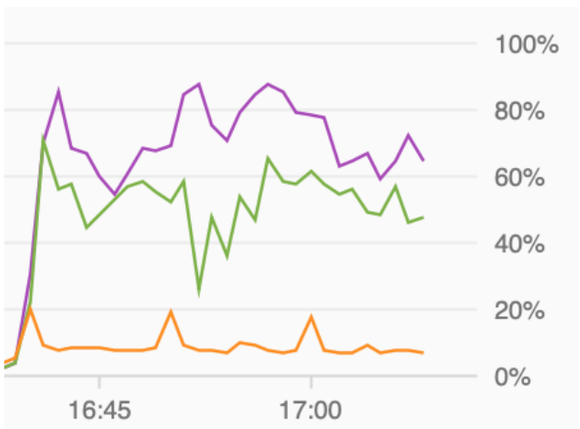

Figura 2 - Uso de *CPU* com 2 *GB* de memória.

Por fim, observando a Figura 3, é notório que a restrição do número total de núcleos implicou que a carga computacional caísse sobre uma só máquina. Note-se que neste gráfico a linha azul representa a máquina *worker2*, a linha verde a máquina *worker1* e a linha roxa o *master*. Tendo em conta esta carga elevada sobre uma só máquina, esta solução permite um menor tratamento de dados do que em comparação com a topologia inicial em que todos os núcleos eram utilizados.

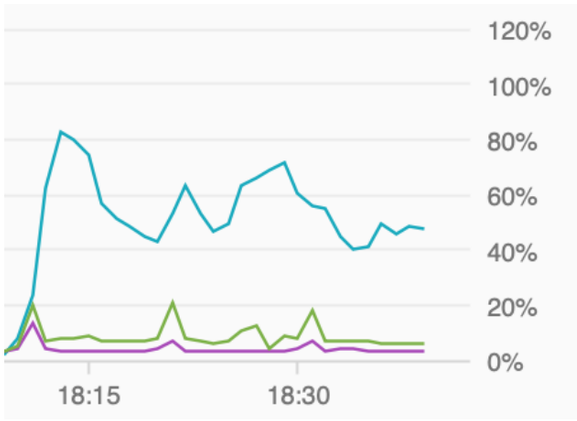

Figura 3 - Uso de *CPU* com 4 *GB* de memória e 2 núcleos no total.

4.3 *Batch*

De modo a analisar a influência dos parâmetros na execução dos algoritmos de processamento em *batch*, a ideia passou por analisar as várias combinações possíveis para o algoritmo mais pesado computacionalmente, sendo este o cálculo dos colaboradores. Desta forma, pela análise da Tabela 1, é notório que o algoritmo é executado de forma mais rápida quando utiliza dois núcleos executores, ou seja, um *CPU* por cada *worker*, permitindo que seja possível efetuar trocas de dados dentro do *cluster* sem grandes problemas. Além disso, em termos de memória, os melhores resultados ocorreram quando foi alocado 1 *GB*, podendo isto ser explicado por ser a quantidade que o algoritmo necessita para executar de forma desejável, sem retirar memória aos restantes componentes que executam nas máquinas.

Tabela 1 - Métricas do algoritmo de cálculo dos colaboradores

total_executor_cores	**executor_memory (Gb)**	**driver_memory (Gb)**	**Tempo (ms)**
1	1	1	595140.0
1	2	2	596798.0
1	4	4	607631.0
2	1	1	**347565.0**
2	2	2	349118.0
2	4	4	399921.0
3	1	1	620087.0

Devido ao elevado custo computacional deste algoritmo e necessidade de trocas de dados, foi comparada a melhor configuração do mesmo com uma configuração sem restrições de núcleos para o cálculo do *top* 10 atores com mais filmes. A execução com apenas dois núcleos demorou 110174.0 ms, enquanto que sem restrição dos mesmos foi mais rápida, tendo executado em 99660.0 ms. Assim, tal como no caso do contexto em *streaming*, a utilização dos núcleos permite uma computação distribuída e, por isso, menos pesada pelos nodos. No entanto, no caso de algoritmos como o cálculo dos colaboradores, em que é necessária a transferência constante de dados e execução de operações complexas, como o produto cartesiano de vários pares e consequente agrupamento dos mesmos, a execução numa só máquina pode ser proveitosa, explicando o tempo de execução menor nesse caso.

5 Utilização da Solução

Em seguida serão apresentados os vários passos de modo a colocar o *cluster* em funcionamento, bem como para enviar e executar um serviço no mesmo.

5.1 Criar Máquinas

De modo a iniciar o processo de criação do *cluster*, o primeiro passo é a criação das máquinas. Para isso, foi definido que a arquitetura deste *cluster* passaria por um *master* e dois *workers*.

Veja-se o Extrato 7 para a criação da máquina mestre.

```
docker-machine create --driver google --google-project ggcdtests-275814 --google-
    zone europe-west1-b --google-machine-type n1-standard-2 --google-disk-size=100
    --google-disk-type=pd-ssd --google-machine-image https://www.googleapis.com/
    compute/v1/projects/centos-cloud/global/images/centos-7-v20200309 master
```

Extrato 7 - Comando para criação da máquina *master*.

No Extrato 8 encontra-se o comando para criação da máquina *worker1*.

```
docker-machine create --driver google --google-project ggcdtests-275814 --google-
    zone europe-west1-b --google-machine-type n1-standard-2 --google-disk-size=100
    --google-disk-type=pd-ssd --google-machine-image https://www.googleapis.com/
    compute/v1/projects/centos-cloud/global/images/centos-7-v20200309 worker1
```

Extrato 8 - Comando para criação da máquina *worker1*.

Tal como nos casos anteriores, a criação do segundo *worker* implica a execução de um comando, podendo este ser visto no Extrato 9.

```
docker-machine create --driver google --google-project ggcdtests-275814 --google-
    zone europe-west1-b --google-machine-type n1-standard-2 --google-disk-size=100
    --google-disk-type=pd-ssd --google-machine-image https://www.googleapis.com/
    compute/v1/projects/centos-cloud/global/images/centos-7-v20200309 worker2
```

Extrato 9 - Comando para criação da máquina *worker2*.

5.2 *Swarm*

Tendo as máquinas criadas e prontas a correr serviços, é necessário interligar as mesmas entre si. Desta forma, foi utilizada a funcionalidade *swarm* do *docker*.

No que toca à máquina *master*, a execução do comando no Extrato 10 permite a obtenção de um *token*, sendo este utilizado para associar máquinas trabalhadoras ao *cluster*.

```
docker-machine ssh master sudo docker swarm init
```

Extrato 10 - Comando para iniciação do *swarm*.

De modo a associar os *workers* ao *swarm*, devem ser executados os comandos presentes no Extrato 11, em que *TOKEN* deve ser o obtido pela execução do comando anterior (Extrato 10).

```
docker-machine ssh worker1 sudo docker swarm join --token TOKEN

docker-machine ssh worker2 sudo docker swarm join --token TOKEN
```

Extrato 11 - Comandos para adição dos *workers* ao *swarm*.

Tendo o *swarm* definido, já com as 3 máquinas pretendidas associadas entre si, deve ser ativado o ambiente da máquina *master* na *bash*, utilizando-se o comando presente no Extrato 12.

```
eval $(docker-machine env master)
```

Extrato 12 - Comando para ativação do ambiente da máquina *master* na *bash*.

5.3 *Deployment*

O passo seguinte é o *deployment* do *cluster*, ou seja, preparar o mesmo com todos os serviços necessários à execução dos programas. Para isso, é utilizado um ficheiro *docker-compose*, associado a uma *stack*, de nome *mystack*, como se pode ver pelo Extrato 13.

```
docker stack deploy -c docker-compose.yml mystack
```

Extrato 13 - Comando para *deployment* dos nodos da *stack*.

Executando o comando *docker service ls* é possível visualizar quais os componentes do *cluster* que já se encontram em funcionamento, permitindo uma melhor manutenção da *stack* desenvolvida.

5.4 *HDFS*

No que toca ao *cluster*, é, ainda, necessária a criação de pastas e a adição de ficheiros necessários à execução dos programas. Executando o comando presente no Extrato 14 é possível aceder a uma *shell* de *HDFS*, sendo possíveis as ações de alterações neste sistema de ficheiros distribuídos.

```
docker run --env-file hadoop.env --network mystack_default -it bde2020/hadoop-base
    bash
```

Extrato 14 - Comando para acesso ao sistema de ficheiros distribuídos.

Assim, executando todos os comandos presentes no Extrato 15, são criadas as pastas para armazenamento de *inputs*, *logs*, *checkpoints* e *outputs*. Além disso, são gerados os ficheiros *title.ratings.tsv*, *title.basics.tsv* e *title.principals.tsv*.

```
hdfs dfs -mkdir /input
hdfs dfs -mkdir /logs
hdfs dfs -mkdir /checkpoints
hdfs dfs -mkdir /output

curl https://datasets.imdbws.com/title.basics.tsv.gz | gunzip |
hdfs dfs -put - hdfs://namenode:9000/input/title.basics.tsv

curl https://datasets.imdbws.com/title.principals.tsv.gz | gunzip |
hdfs dfs -put - hdfs://namenode:9000/input/title.principals.tsv

curl https://datasets.imdbws.com/title.ratings.tsv.gz | gunzip |
hdfs dfs -put - hdfs://namenode:9000/input/title.ratings.tsv
```

Extrato 15 - Comandos para criar pastas e inserir ficheiros no sistema de ficheiros distribuído.

5.5 *Dockerfile*

Por fim, no que toca ao *Dockerfile*, para execução dos programas desenvolvidos ao longo do projeto, além da utilização deste associado à vertente *docker-machine*, com ligação à máquina *master*, é necessária a atribuição das opções de execução presentes no Extrato 16. Note-se que o caminho para o ficheiro *hadoop.env* varia de máquina para máquina, sendo o exemplo do Extrato um caminho numa das máquinas do grupo.

```
--network mystack_default --env-file /Users/JoaoPimentel/Desktop/4ANO/GGCD/Projeto/
    TP2/swarm-spark/hadoop.env
```

Extrato 16 - Opções de execução do *Dockerfile*.

6 Conclusões e Trabalho Futuro

Ao longo do presente relatório encontra-se representado o resultado do segundo 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 informação, bem como tratar de dados referentes a votos sendo constantemente gerados, tirando proveito da ferramenta *Spark*.

O desenvolvimento do projeto permitiu aprimorar os conhecimentos previamente adquiridos e postos em prática aquando da realização dos guiões da Unidade Curricular. Desta forma, foi possível pensar nos algoritmos numa perspetiva de otimização e não apenas conseguir realizar as tarefas. Além disso, foi possível compreender as grandes vantagens de uma computação distribuída para tratamento de grandes conjuntos de dados. Destaque em especial para a análise de métricas, em que a utilização dos núcleos das máquinas *worker* permitia um maior débito de dados, bem como a utilização de uma quantidade de memória elevada, sem por em causa as restantes operações em execução nas máquinas.

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