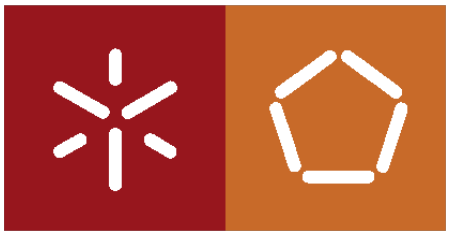


Universidade do Minho
MIEI - 4º ano - 2º semestre

Gestão de Grandes Conjuntos de Dados

Apache Spark e Hadoop HDFS

Grupo 2:

A76516 - João Vieira
A74357 - António Lopes

5 de junho, 2020

Conteúdo

1 Introdução **2**

2 Configurações Iniciais **2**
2.1 Configuração Docker Swarm 2
2.2 Hadoop HDFS 3
2.3 Dockerfile e Estrutura 4

3 Desenvolvimento das Tarefas **4**
3.1 Processamento de *Streams* 4
3.1.1 Log 5
3.1.2 Top3 5
3.1.3 Trending 5
3.2 Processamento em *Batch* 6
3.2.1 Top10 6
3.2.2 Friends 7
3.2.3 Ratings 7

4 Configurações de Execução e Resultados **8**
4.1 Processamento de *Streams* 8
4.2 Processamento em *Batch* 9

5 Conclusão **12**

1 Introdução

O segundo trabalho prático da U.C. de Gestão de Grandes Conjuntos de Dados, consiste na concretização e avaliação de tarefas de armazenamento e processamento de grandes quantidades de dados. Este processamento é dividido em duas componentes, processamento em *batch* e *streaming*. Como tal, as tarefas serão implementadas com a *framework* do **Apache Spark**, e com o auxílio do **Hadoop HDFS** para disponibilizar um sistema de ficheiros distribuído.

Os dados a utilizar são do *dataset* público do **IMDB**, e o objetivo será realizar todas as computações necessárias a estes dados, de modo a dar resposta às tarefas do enunciado, procurando também sempre o melhor desempenho em termos de implementação.

Este relatório documenta todo o processo de trabalho, desde o *setup* do ambiente a utilizar, o desenvolvimento efetivo das tarefas, passando pela escolha da melhor configuração de execução, até aos resultados finais obtidos.

2 Configurações Iniciais

Nesta secção, está explicitado o ambiente e configurações iniciais para a execução das tarefas necessárias, que será feita através da **Google Cloud Platform**. Para tal, é usada uma instalação *Docker* para obter o ambiente que permite realizar o projeto.

Este ambiente permite configurar a *stack* de execução através de *docker-machines*, assim como criar o sistema de ficheiros distribuídos do **Hadoop HDFS**, que irá armazenar os ficheiros de *input* e outros que se ache necessário.

2.1 Configuração Docker Swarm

Primeiramente é necessário criar três máquinas na **GCP** que irão fazer parte da nossa *stack* de execução, em que uma será a *master* e as outras duas serão máquinas *worker*. Estas máquinas terão todas a mesma configuração, utilizando uma imagem de **CentOS-7** e um disco SSD com tamanho de 100GB. Através do seguinte comando que irá ser executado três vezes, apenas alterando o nome da instância no final, criam-se as três máquinas:

```
$ docker-machine create \
> --driver google --google-project PROJECT-ID \
> --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 \
> INSTANCE-NAME
```

A seguir, é necessário descarregar a configuração *docker* do **swarm-spark** disponibilizada pelo docente. Esta configuração vai permitir criar o *swarm* das máquinas e ligar as duas máquinas *worker* à *master*, através de *docker-machine*. Com o comando seguinte, dá-se o *setup* do *swarm* no master:

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

Este último comando irá retornar um *token*, que será necessário para ligar as duas restantes máquinas *worker* ao *swarm*, com os comandos a seguir:

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

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

Com o *swarm* devidamente configurado, ativa-se o ambiente *master* no terminal:

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

Antes de proceder ao *deployment* da *stack* de configuração no *swarm*, é necessário adicionar o nosso gerador de eventos, **streamgen**, aos serviços do ficheiro docker-compose.yml, dentro da diretoria do **swarm-spark**:

```
streamgen:
    image: jopereira/streamgen
    command: hdfs://namenode:9000/input/title.ratings.tsv 120
    env_file:
      - ./hadoop.env
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - "node.role==manager"
```

Após este passo, podemos finalmente dar *deploy* da configuração através do comando:

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

No final, é só esperar que todos os serviços da stack fiquem ativos e a configuração estará pronta para executar tarefas. Para verificar se estes se encontram em execução, correr o comando:

```
$ docker service ls
```

Para posteriormente remover a configuração do *swarm* executar:

```
$ docker stack rm mystack
```

2.2 Hadoop HDFS

Tendo já o nosso sistema de ficheiros distribuído do **HDFS** ativo nos serviços da *stack* configurada anteriormente, resta apenas aceder a este e carregar os respetivos ficheiros de dados, que servirão de *input* às tarefas do projeto. Para acedermos à *bash* do sistema de ficheiros, executa-se o comando seguinte, dentro do ambiente *master*:

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

Na *bash* do **HDFS** criou-se uma pasta de nome *input*, que será dedicada a armazenar exclusivamente os ficheiros do IMDB que as tarefas irão receber. Para criar a pasta, e carregar os três ficheiros necessários às tarefas, **title.principals.tsv**, **title.basics.tsv** e **title.ratings.tsv**, executámos respetivamente os comandos:

```
// Criar pasta
# hdfs dfs -mkdir /input

// Carregar os 3 ficheiros
# 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

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

Temos portanto, tanto a configuração de *swarm* como o nosso sistema de ficheiros prontos para executar as tarefas do projeto.

2.3 Dockerfile e Estrutura

O projeto como é sabido, encontra-se dividido em duas componentes, *streaming* e *batch*. Com isto em conta, foram criadas duas *main* separadas de execução, uma com o nome **Ex1** que executa as tarefas de *streaming* em simultâneo, e outra com o nome **Ex2** que executa as tarefas de *batch* sequencialmente.

O **Dockerfile** responsável pela execução das tarefas na *cloud* tem a seguinte estrutura base:

```
FROM bde2020/spark-base:2.4.5-hadoop2.7
COPY target/spark-tp2-1.0-SNAPSHOT.jar /
ENTRYPOINT ["/spark/bin/spark-submit", "--class", "streaming.Ex1",
"--master", "spark://spark-master:7077", "/spark-tp2-1.0-SNAPSHOT.jar"]
```

Também é necessário configurar as definições de execução do *Dockerfile*, que são as apresentadas a seguir:

```
-p 4040:4040 --network mystack_default
--env-file /...PATH-TO.../swarm-spark/hadoop.env
```

Por fim, é importante não esquecer de conectar o *daemon* do Docker à *docker-machine* do *master*. Temos portanto o ficheiro pronto a executar as tarefas pretendidas.

Recomenda-se a execução primeiro das tarefas de *streaming*, através da classe *streaming.Ex1* pois o Ex2 irá utilizar ficheiros criados por esta. De modo a executar depois as tarefas de *batch*, basta trocar o argumento a seguir à *flag* −**class** para *batch.Ex2* no **Entrypoint**,

3 Desenvolvimento das Tarefas

Esta secção irá abordar os objetivos propostos, assim como explicitar a implementação realizada para dar resposta a cada um deles, dividida nas duas componentes deste projeto *streaming* e *batch*.

Para uma melhor organização e leitura do código, o projeto tem uma pasta de nome *batch* relativa às tarefas deste processamento, e outra pasta de nome *streaming* com as tarefas deste tipo de processamento.

3.1 Processamento de *Streams*

Nesta componente do projeto, temos um serviço **streamgen** que irá produzir eventos de texto com um identificador de filme arbitrário, dos que estão presentes no *dataset*, e um voto em decimal respetivo para esse filme.

As seguintes tarefas têm o objetivo de processar essa *stream* de dados e dar resposta ao que é pedido no enunciado, sendo executadas todas ao mesmo tempo.

3.1.1 Log

Esta primeira tarefa pedia que fossem armazenados todos os votos individuais, em lotes de 10 minutos, e que esses lotes fossem guardados em ficheiro.

Como tal, primeiramente foi definida a *window* desta tarefa com duração de 10 minutos e com o *slide* também para 10 minutos. Dentro desta *window*, os RDD's foram percorridos e guardados todos para um mesmo ficheiro, com o nome **part-00000**. Este ficheiro é guardado numa diretoria em que o nome é do tipo **Hora_Minutos**, de modo a identificar os diferentes lotes. Por sua vez, todos estes lotes e respetivas diretorias, estarão guardadas no **HDFS** na diretoria **/logs**.

Output:

```
root@4235a0462434:/# hdfs dfs -head /logs/21_52/part-00000
2020-06-04 21:53:10,705 INFO sasl.SaslDataTransferClient: S
ostTrusted = false, remoteHostTrusted = false
(tt0114370,0.0)
(tt4686862,8.0)
(tt0050084,4.0)
(tt0395975,10.0)
```

3.1.2 Top3

Para esta tarefa, era pedido que fosse realizada uma computação sobre a *stream* de dados, de modo a apresentar o top 3 de filmes, a cada minuto, que obtiveram melhor classificação média nos últimos 10 minutos.

Primeiro, foi definida a janela desta tarefa com uma duração de 10 minutos, e um *slide* de 1 minuto de modo a imprimir para cada *slide*. De seguida, os dados foram agrupados por *key*, para se obter todos os votos individuais para cada filme num *iterable*. Para termos a média de votos para cada filme, mapearam-se os dados para um novo tuplo, que irá conter a *key* do filme e a respetiva média, calculada através de uma *stream* sobre o iterador de votos.

Com os dados no formato pretendido, para cada RDD foi feito um *join* com um *JavaPairRDD*, guardado em *cache*, que contém para cada *id* de filme, o seu respetivo título. Após isto, os dados foram mapeados de modo a ter como tuplo a média e o respetivo título, e ordenados por ordem decrescente da média. Por fim, foi feito um *take* das 3 primeiras entradas, retornando esta informação no formato de lista e imprimindo assim para o ecrã o top 3 de títulos nos últimos 10 minutos.

Output:

```
20/06/04 22:05:39 INFO DAGScheduler: Job 10 finished: take at Ex1.java:63, took 0.073129 s


TOP3: [(10.0,Four Weddings and a Funeral), (10.0,Trollsyn), (10.0,Clown)]
```

3.1.3 Trending

Nesta última tarefa de *streaming*, pedia-se que a cada 15 minutos, se apresentasse apenas os títulos em que o número de votos recolhido nesse período fossem superiores aos votos obtidos no período anterior.

Para a resolução desta tarefa, foi necessário fazer uma implementação *Map with State*, em que para cada período temos um estado que é atualizado e guardado de modo a ser utilizado com o *input* do período seguinte.

Sendo assim, foi definida a *window* da tarefa com ambos a duração e *slide* de 15 minutos. Foram agrupados novamente todos os votos num iterador, e estes dados foram mapeados para um tuplo com

o *id* do filme e a respetiva contagem do nº de votos no iterador. Tendo assim a informação necessária à tarefa, resta-nos implementar o mapeamento com estado que é feito sobre os dados para cada janela.

Este tipo de mapeamento é feito através de uma função que recebe a *key* do filme, um *Optional* com a contagem dos votos no período atual, caso exista, e um *State* com a contagem dos votos para a mesma *key* mas no período anterior, novamente caso exista essa informação. Como tal, foram definidos três casos possíveis nesta computação:

- O **primeiro** caso acontece quando existe informação da contagem de votos para um filme, tanto no período atual como no período anterior. Com isto, guarda-se numa variável *noVotes* a contagem dos votos no período atual, **apenas** quando esta contagem é maior do que na contagem no estado anterior. Caso contrário, é guardado na variável o valor 0. No final é realizado um *update* ao estado com o novo valor, de modo a que este seja guardado para o período seguinte.

- O **segundo** caso acontece quando existe uma contagem de votos do filme para o período atual, mas não existe essa informação no estado anterior. Sendo assim, como temos apenas a contagem do período atual, esta é guardada diretamente na variável *noVotes* e é feito o *update* do estado com este novo valor, para o período seguinte.

- O **terceiro** e último caso, apenas acontece quando nenhum dos dois casos acima se verifica. Ou seja, apenas se verifica quando não existe nem uma contagem atual, nem uma contagem do período anterior para um mesmo filme. Com isto, é guardada na variável *noVotes* o valor 0, e removido a instância de estado deste filme para o período seguinte.

No final desta computação de casos, a função do *MapWithState* retorna um tuplo com o *id* do filme e a respetiva variável com o nº de votos, atualizada anteriormente.

De seguida, é feito um *filter* de modo a que sejam filtradas as entradas cujo nº de votos tenham valor 0, sendo assim descartadas na impressão do resultado. Ao finalizar a tarefa, são percorridos os *RDD's* e à semelhança da tarefa do **top3**, é feito um *join* com o *JavaPairRDD* que contém os títulos de filmes para cada *id*, e os dados são mapeados de forma a termos a informação no formato de tuplo **(Título,Contagem)**. É portanto feito um *collect* e imprimido para o ecrã a lista dos filmes cuja contagem de votos seja superior à do período anterior.

Output:

```
20/06/04 22:06:26 INFO DAGScheduler: Job 11 finished: collect at Ex1.java:95, took 46.810358 s


TRENDING: [(Pueblo Terror,1), (Melo,1), (Shanghai Moon,1), (A Summer You Will Never Forget,2), (The Happening,1), (Gonshchiki
,1), (Je t'aime moi non plus,1), (Starship Invasions,1), (Blood Link,1), (Unfair Exchanges,1), (Breaking All the Rules,1), (N
oble House,1), (Saturday Night Special,1), (Archangel,1), (Nitrate Kisses,1), (Philadelphia Experiment II,1), (Across the Moo
n,1), (Intimate with a Stranger,1), (Friend of the Family,1), (The Search for Christmas,1), (Lassie,1), (Every Dog Has Its Da
```

3.2 Processamento em *Batch*

Nesta componente de processamento, as tarefas correspondentes têm o objetivo de processar os ficheiros por completo do *dataset* e realizar as devidas computações sobre os dados. Cada uma destas tarefas é executada sequencialmente no projeto.

3.2.1 Top10

Para esta alínea, era pedido o top 10 dos atores que participaram em mais filmes diferentes. Esta informação está contida no ficheiro **title.principals.tsv** em que um dado *id* de filme tem várias entradas, e cada uma dessas entradas contém informação sobre uma pessoa que teve uma determinada função nesse filme.

Como tal, em primeiro lugar é dado como *input* o ficheiro de texto e separados os diversos campos, para cada linha de texto. De seguida, são filtradas apenas as entradas cujo atributo *category* equivale à palavra *actor* ou *actress*, tendo assim apenas as entradas relativas a atores, como era pedido. Por conseguinte, é mapeado para um tuplo o *id* do ator na primeira posição, e inicializado com um inteiro 1 na segunda posição. É feito um *foldByKey* que vai agrupar por *id* de ator as entradas, e realizar a computação de soma de todos os inteiros presentes nessas entradas, obtendo assim o nº de filmes por ator.

No final, é mapeado um tuplo de forma a ter o nº de filmes como *key* e o *id* de ator como *value*, isto servirá para realizar um *sortByKey* que vai ordenar as entradas por ordem decrescente. É novamente realizado um mapeamento final para voltar a colocar o *id* de ator como *key* do tuplo e o respetivo nº de filmes como *value*, e por fim é realizado um *take* das 10 primeiras entradas, de modo a termos o top 10 como desejado. Esta informação é imprimida para o ecrã na forma de lista.

Output:

```
20/06/04 22:12:35 INFO DAGScheduler: Job 1 finished: take at Top10.java:19, took 3.451520 s


TOP10: [(nm10120013,7447), (nm0151534,5304), (nm0887808,4454), (nm1518349,4432), (nm0839741,4431), (nm1944549,4422), (nm0708
802,4138), (nm6427507,4032), (nm0192504,3990), (nm0555829,3917)]
```

3.2.2 Friends

Esta tarefa pedia que fosse calculado o conjunto de colaboradores de um ator, ou seja, atores que participaram nos mesmos filmes. Esta informação está novamente contida no ficheiro **title.principals.tsv** que servirá como *input*.

Como tal foram filtradas as entradas correspondentes a atores, mapeadas para um RDD contendo o *id* do filme e *id* de um ator, e agrupados por *id* de filme. De seguida a lista de atores de cada filme, foi convertida para um *Set* para evitar repetições, e em que cada entrada deste *Set* era um tuplo com dois atores, evidenciando uma relação. De seguida, cada par do *Set* foi transformado num RDD e esses RDD's foram agrupados por *key* ou seja pelo *id* de cada ator. A lista dos colaboradores é novamente convertida para um *Set* e no final é filtrada a entrada do ator que está como *key* do seu *Set* de colaboradores. A informação é imprimida para o ecrã na forma de lista, em que cada entrada é um tuplo do tipo **(Ator,[Colaboradores])**.

Apesar desta tarefa executar com sucesso em ambiente local, o mesmo não acontece quando executado na *cloud* da **GCP**. A execução é abortada por um erro de falta de memória, não chegando a computar os resultados pretendidos. Tendo isto em conta e não querendo prejudicar a execução das outras duas tarefas relativas a este exercício 2, foi comentada a chamada do método que executa esta tarefa na respetiva *main*.

Output:

```
20/06/04 22:32:47 INFO BlockManagerInfo: Removed broadcast_2_piece0 on a27f4d91af31:43895 in memory (size: 4.6 KB, free: 366.
3 MB)


Entry: (nm4177192,[nm4935937, nm3863077, nm2700522, nm5996169, nm9413182])
```

3.2.3 Ratings

Nesta última tarefa, era pedido que o ficheiro **title.ratings.tsv** fosse atualizado tendo em conta os votos recebidos e guardados no HDFS, aquando da execução da primeira tarefa de *streaming* **Log**. Este ficheiro contém para cada filme, a média de votos e o respetivo nº de votos.

Como tal, sabemos que os ficheiros de votos estão guardados na diretoria **/logs** do HDFS, organizados por várias pastas, em que cada uma dessas pastas contém um ficheiro **part-00000** com vários votos individuais para aquele espaço de tempo. A implementação itera todas as pastas dentro de **/logs** e procura os ficheiros pretendidos. Sempre que for encontrado um ficheiro **part-00000**, o ficheiro original é processado e são retirados para um RDD os campos *id* de filme, média e nº de votos.

De seguida, é necessário processar o ficheiro de votos, retirando cada um dos votos por filme, e agrupando-os por filme. É feito um *fullOuterJoin* com o RDD do ficheiro original, de modo a termos para um *id* de filme, informação sobre a média e contagem de votos atual, assim como a informação sobre os novos votos, caso esses existam. O RDD resultante do *join* é processado tendo em conta os dois seguintes casos:

- O **primeiro** caso acontece quando existe informação sobre novos votos. Para cada filme, a média e contagem atual são guardados, e cada um dos novos votos são adicionados à média através do método *addToAverage*, atualizando assim a média e incrementando a contagem do nº de votos. Depois de iterar todos os novos votos, é retornado um tuplo com o *id* do filme, e as respetivas novas média e contagem.
- O **segundo** caso ocorre quando não foram encontrados novos votos para um dado filme. Quando isto acontece, é simplesmente retornado um tuplo com a informação da média e contagem de votos original, por filme.

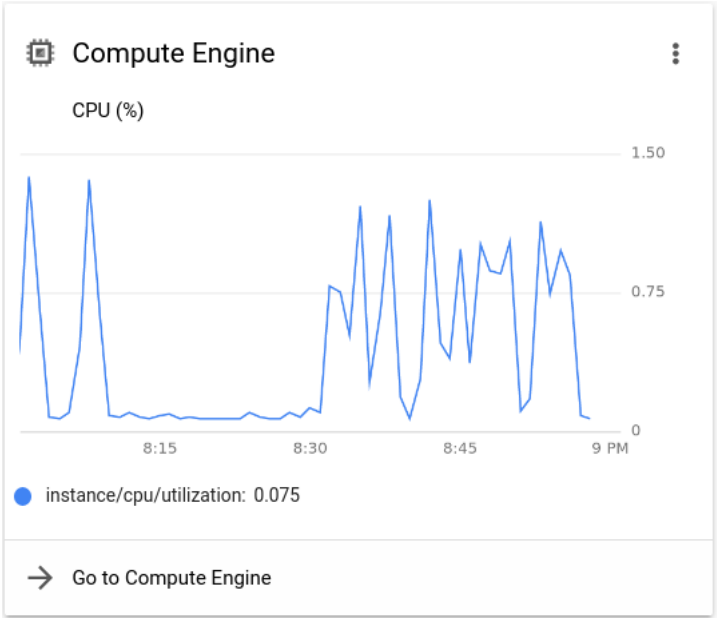
Depois de computados todos os votos de um determinado ficheiro, os campos *id* de filme, média e nº de votos são mapeados da mesma forma do ficheiro original, separados por um *tab*. Esta informação está contida num RDD que é guardado em ficheiro no HDFS, na diretoria **/tmp**, com o nome **part-00000**. O último passo trata de apagar o ficheiro original da diretoria **/input** e move o ficheiro atualizado novamente para o **/input** com o devido nome **title.ratings.tsv**. A diretoria **/tmp** é assim eliminada do HDFS.

4 Configurações de Execução e Resultados

4.1 Processamento de *Streams*

Nesta secção, são testadas diferentes métricas de eventos/min gerados pelo **streamgen** e realizada a execução das tarefas para cada uma das métricas.

Como tal, vamos ter atenção ao índice de utilização do CPU (aproximadamente) na **GCP**, disponível através da sua interface no *browser*:



As métricas de eventos/min que serão testadas são: 120, 240, 480 e 960.

- **120 eventos/min**
  Índice CPU: 1.650
- **240 eventos/min**
  Índice CPU: 1.700
- **480 eventos/min**
  Índice CPU: 1.750
- **960 eventos/min**
  Índice CPU: 1.800

Como se pode analisar, a diferença do índice para as diferentes métricas é mínima, logo conclui-se que um maior nº de eventos gerados não afeta muito tanto em termos de *performance* como de *software*.

4.2 Processamento em *Batch*

Nesta secção, serão testadas diferentes configurações de execução do **spark**, combinando opções de execução como **executor-memory**, **driver-memory** e **total-executor-cores**. O desempenho será aferido calculando o tempo de execução de cada configuração (em milissegundos), e analisando a que dá melhor resultados.

Para isso, apenas iremos testar executando uma das tarefas de *batch*, neste caso o **top10**, para tornar mais fácil a análise.

- **1GB de memória + 2 cores de execução**

```
ENTRYPOINT ["/spark/bin/spark-submit",
"--executor-memory","1g","--driver-memory","1g",
"--total-executor-cores","2", "--class", "batch.Ex2",
```

```
"--master", "spark://spark-master:7077",
"/spark-tp2-1.0-SNAPSHOT.jar"]
```

Tempo: 58094ms

- **1GB de memória + 4 cores de execução**

```
ENTRYPOINT ["/spark/bin/spark-submit",
"--executor-memory","1g","--driver-memory","1g",
"--total-executor-cores","4", "--class", "batch.Ex2",
"--master", "spark://spark-master:7077",
"/spark-tp2-1.0-SNAPSHOT.jar"]
```

Tempo: 52071ms

- **2GB de memória + 2 cores de execução**

```
ENTRYPOINT ["/spark/bin/spark-submit",
"--executor-memory","2g","--driver-memory","2g",
"--total-executor-cores","2", "--class", "batch.Ex2",
"--master", "spark://spark-master:7077",
"/spark-tp2-1.0-SNAPSHOT.jar"]
```

Tempo: 62615ms

- **2GB de memória + 4 cores de execução**

```
ENTRYPOINT ["/spark/bin/spark-submit",
"--executor-memory","2g","--driver-memory","2g",
"--total-executor-cores","4", "--class", "batch.Ex2",
"--master", "spark://spark-master:7077",
"/spark-tp2-1.0-SNAPSHOT.jar"]
```

Tempo: 52848ms

- **4GB de memória + 2 cores de execução**

```
ENTRYPOINT ["/spark/bin/spark-submit",
"--executor-memory","4g","--driver-memory","4g",
"--total-executor-cores","2", "--class", "batch.Ex2",
"--master", "spark://spark-master:7077",
"/spark-tp2-1.0-SNAPSHOT.jar"]
```

Tempo: 60157ms

- **4GB de memória + 4 cores de execução**

```
ENTRYPOINT ["/spark/bin/spark-submit",
"--executor-memory","4g","--driver-memory","4g",
"--total-executor-cores","4", "--class", "batch.Ex2",
"--master", "spark://spark-master:7077",
"/spark-tp2-1.0-SNAPSHOT.jar"]
```

Tempo: 51313ms

Como podemos averiguar, a configuração que teve melhor tempo de execução foi a última, **4GB de memória + 4 cores de execução**, apesar de não ser uma grande diferença para outras configurações testadas.

5 Conclusão

Dado por finalizado o segundo e último trabalho prático da U.C., é da opinião do grupo que, de um modo geral, a resolução do enunciado foi cumprida com sucesso. Contudo, é claro que a implementação das tarefas poderia ser melhorada, tanto a nível do algoritmo como do seu desempenho.

A principal dificuldade com que nos deparamos foi sem dúvida com todo o *setup* do ambiente de computação na **GCP**, e respetivamente, a passagem da execução local das tarefas para execução na *cloud*. Felizmente, conseguimos ultrapassar estes problemas e realizar com sucesso a execução das tarefas no ambiente distribuído que era requerido no enunciado.

Outra das dificuldades foi a habituação e implementação de tarefas na *framework* do **Spark**, pois tal como no primeiro trabalho prático, são ambientes de programação nunca antes enfrentados no nosso percurso académico. Este tipo de dificuldade foi evidenciado a tentar perceber como funcionava a execução de tarefas em *streaming*, mas também em *batch*, nomeadamente a resolução da segunda alínea do exercício 2 que nos causou mais obstáculos, como foi mencionado neste relatório. Contudo, as restantes soluções encontradas dão uma boa resposta ao que são os objetivos de cada tarefa.

Em suma, o projeto foi muito enriquecedor para mais uma vez aprendermos uma nova forma de processamento de um grande conjunto de dados, e as dificuldades e fatores inerentes a este tipo de processamento. Realizando um comparativo com o primeiro trabalho, em que era feito um processamento com base em tarefas de **Map/Reduce**, o grupo tem a opinião que foi preferível usar o **Spark** para processar este tipo de dados. Tanto a nível de implementação, facilitada por uma programação com base em *streams* de **Java 8** a que estamos mais à vontade, como a nível de desempenho pois a execução das tarefas era efetivamente mais rápida no **Spark**.