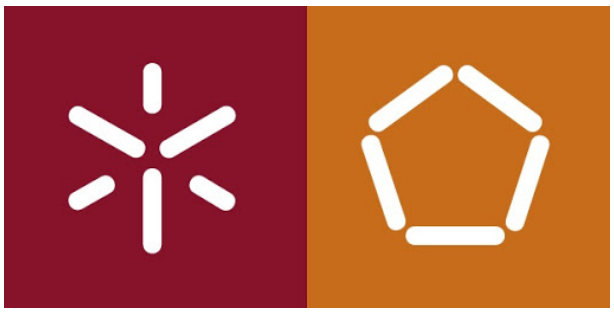

Gestão de Grandes Conjuntos de Dados
Trabalho Prático 1
Grupo 11

Daniel Regado (PG42577)

Diogo Dias (PG42825)
Guilherme Palumbo (PG42832)
João Silva (PG42834)
Nelson Costa (PG44587)

Conteúdo

1 | Introdução

Este trabalho prático tem como objetivo aplicar os conhecimentos adquiridos nas aulas relativos à utilização do Hadoop para processamento de dados, assim como conceitos diretamente relacionados, nomeadamente MapReduce, Avro Parquet, e interações com o HDFS. Além disso, tentamos também enriquecer o conteúdo do trabalho prático, não apenas pela realização do objetivo 3, mas também pelo desenvolvimento de funcionalidades adicionais que serão expostas mais adiante. Algumas dessas funcionalidades estão relacionadas com a Google Cloud Platform, visto que decidimos usar o *Hadoop* recorrendo ao *Dataproc*, disponibilizado pela Google como *PaaS*.

2 | Preparação e Contextualização

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*.

2.1.1 Dependências

Google Cloud Storage Connector

Esta dependência já vem instalada no *Dataproc*, portanto podemos assinalar como *provided*. Irá ser utilizada para aceder a dados que estejam em algum *bucket* do Google Storage.

```
<dependency>
    <groupId>com.google.cloud.bigdataoss</groupId>
    <artifactId>gcs-connector</artifactId>
    <version>hadoop3-2.2.0</version>
    <scope>provided</scope>
</dependency>
```

Hadoop

O Hadoop também já vem instalado no *Dataproc*, portanto também é assinalado como *provided*.

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>3.3.0</version>
    <scope>provided</scope>
</dependency>
```

Parquet-Avro

As seguintes dependências são relativas ao *Parquet* e *Avro*. Estas não estão presentes no *Dataproc*, pelo que temos de as definir aqui, para posteriormente serem incluídas na altura de *packaging*.

```
<dependency>
    <groupId>org.apache.parquet</groupId>
```

```
    <artifactId>parquet-hadoop</artifactId>
    <version>1.12.0</version>
</dependency>
<dependency>
    <groupId>org.apache.parquet</groupId>
    <artifactId>parquet-avro</artifactId>
    <version>1.12.0</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
- schemas

Em *gcloud*, temos um *bash script* para interação com o cluster, e um *ansible playbook* para interação com um Storage Bucket. Estes serão abordados 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 *schemas* temos todos os schemas *Parquet* necessários para os Jobs.

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_NAME="ggcd-hadoop"
CLUSTER_NAME="hadoop-cluster"
REGION="europe-west1"
ZONE="europe-west1-b"
MASTER_MACHINE_TYPE="e2-standard-4"
WORKER_MACHINE_TYPE="e2-highmem-4"
DISK_SIZE=50
SSD_DISK_TYPE="pd-ssd"
WORKERS=2
SECONDARY_WORKERS=2
IMAGE_VERSION="2.0-debian10"
MAX_IDLE_SECONDS="7200s"
SCOPE="https://www.googleapis.com/auth/cloud-platform"
```

Desta forma, temos definidos neste script as caraterísticas dos cluster. Optamos por uma arquitetura que excedesse as necessidades mínimas para execução dos Jobs, de forma a tornar estes mais rápidos e a suportar o *dataset* na íntegra. Optamos também por discos SSD de forma a melhorar a performance de I/O do HDFS. Colocamos também um *timeout* de 2 horas, de forma a que não sejam gastos recursos desnecessáriamente.

Decidimos também explorar melhor as funcionalidades oferecidas pela Google Cloud, e acabamos por incorporar *secondary workers*. 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.

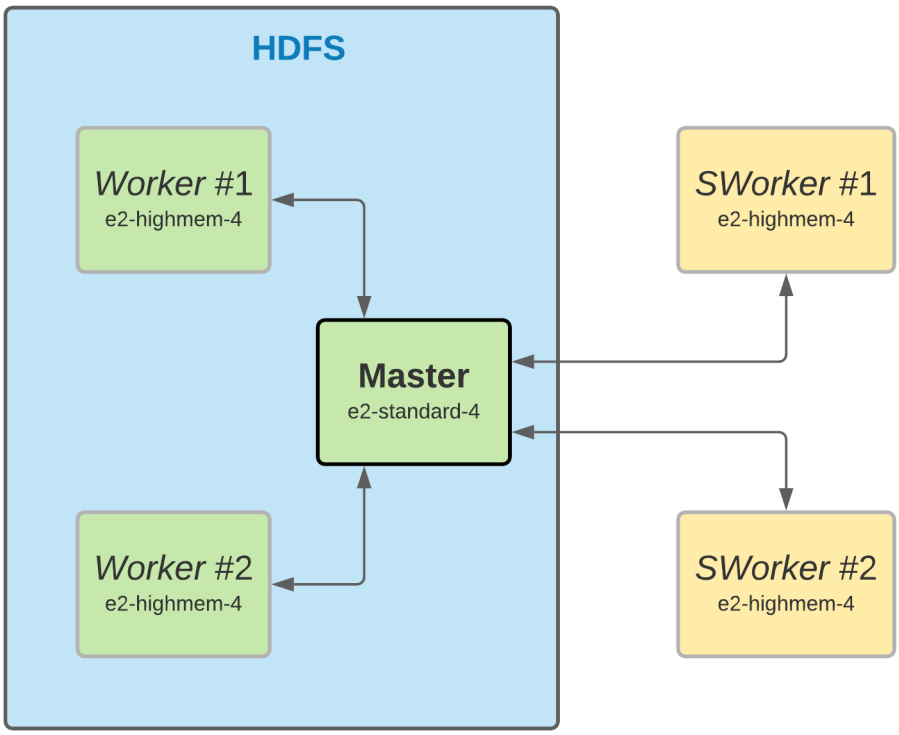

Figura 2.1: Arquitetura do cluster

2.3.2 Ações

Estão implementadas as seguintes ações de interação com o cluster:

Interações com 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 para cluster	$ ./cluster.sh submit `<JAR> <CLASS> [ARGS]`
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 `<SRC>`
Listar diretoria no HDFS	$ ./cluster.sh hdfs_ls `<DIR_PATH>`

2.4 Storage

Para obter maior persistência de dados, a utilização dum *bucket* no Google Storage faz com que dados resultantes da execução de jobs não sejam perdidos quando o cluster é apagado, visto que dados no HDFS também irão desaparecer. Tendo isso em conta, e à existência de uma *collection* no *Ansible Galaxy* dedicada à Google Storage, criamos um pequeno *playbook* para interagir com um dado storage bucket. Será necessário a instalação da *collection*, recorrendo a:

```
$ ansible-galaxy collection install google.cloud
```

Trata-se do ficheiro *storage.yaml*, e neste ficheiro estão contidas informações relativas ao bucket sobre o qual irão incidir as ações:

```
vars:
  gcp_project: "707747151911"
  gcp_auth_file: "auth.json"
  gcp_region: "europe-west1"
  bucket_name: "ggcd-data"
```

Como podemos observar, é necessário declarar a localização de um ficheiro de autenticação, visto que este playbook recorre a uma *service account*.

2.4.1 Ações

Interações com bucket	
Descrição	**Comando**
Cria o bucket	`$ ansible-playbook storage.yaml --tags "create"`
Elimina o cluster	`$ ansible-playbook storage.yaml --tags "destroy"`
Upload de ficheiros para o bucket	`$ ansible-playbook storage.yaml --tags "upload" --extra-vars "src=<SRC> dest=<DEST>"`
Download de ficheiros do bucket	`$ ansible-playbook storage.yaml --tags "download" --extra-vars "src=<SRC> dest=<DEST>"`
Download de pastas do bucket	`$ ansible-playbook storage.yaml --tags "download-folder" --extra-vars "src=<SRC> dest=<DEST>"`
Apagar ficheiros do bucket	`$ ansible-playbook storage.yaml --tags "delete" --extra-vars "src=<SRC>"`

Destas ações, três delas não recorrem aos módulos no *Ansible*. Uma delas é a ação de eliminar o bucket, pois essa ação não é disponibilizada diretamente em nenhum módulo. As outras ações são as de *upload* e *download-folder*. A de *upload* tem problemas ao carregar ficheiros grandes, e o *download* pelo módulo Ansible não suporta cópia recursiva. Para contornar estes problemas, recorre-se diretamente ao *gsutil*.

3 | Jobs

Neste capítulo, iremos abordar os *Jobs* que correspondem aos objetivos a cumprir do enunciado, assim como um job adicional que implementamos, de forma a enriquecer o trabalho prático. Iremos primeiro abordar conceitos que são comuns a todos os jobs, e passaremos depois para uma análise individualizada de cada job.

3.1 Considerações gerais

3.1.1 Inputs

Todos os jobs requerem algum tipo de input para executarem corretamente. No mínimo, devem ser referidos os ficheiros de entrada e a diretoria de saída. Existem também outras opções que deverão ser dada aos jobs, sendo estas opções específicas para um dado job. De forma a facilitar a compreensão e flexibilidade da especificação destas opção, estabelecemos que as opções devem ser introduzidas com a seguinte formatação:

```
--<OPTION>=<VALUE>
```

Desta forma, uma opção deverá ser precedida por `--` e conter o valor após o carater `=`. Isto faz com que seja mais fácil identificar os atributos de input, e que a ordem pela qual estes são introduzidos é indiferente.
Cada job tem uma lista de parâmetros obrigatórios, estando essa lista definida em cada ficheiro do Package *Jobs*. Essa lista terá a seguinte forma (exemplo de um job):

```java
private static final List<String> requiredOptions = Arrays.asList("input", "output");
```

Caso uma destas opções obrigatórias não seja introduzida, o job irá reconhecer as opções em falta, termina o processo e coloca a seguinte mensagem no *sterr* (exemplo de não introduzir `--output`):

```
MISSING OPTIONS: Job not submitted, the following required options are missing: --output
ERROR: (gcloud.dataproc.jobs.submit.hadoop) Job [915ac027b288478dad50ab644aee5bd9] failed with error:
```

Figura 3.1: Mensagem de erro por falta de parâmetros obrigatórios

Existem 2 opções que podem ser introduzidas em todos os jobs, sendo essas:

Overwrite

`--overwrite=true` faz com que o job apague a output directory antes de executar o job.

Reducers

`--reducers=<VALUE>`, faz com o job defina as *reduce tasks* para o valor introduzido (quando aplicável).

3.1.2 Filesystems

Para qualquer um dos Jobs, podem ser especificados ficheiros necessários para a execução, tanto no HDFS como na Google Storage, pelo que foi implementado acesso a estes *filesystems*, mesmo que seja para acesso em cache, como acontece com o ficheiro de *ratings*.

3.2 BasicsRatingsParquet

Este *Job* corresponde à tarefa pedida para resolução do objetivo 1, converter os dados dos ficheiros *title.basics.tsv* e *title.ratings.tsv* para um único ficheiro *Parquet* com um esquema apropriado.

3.2.1 Esquema *Parquet*

O esquema que definimos foi o seguinte, presente no ficheiro *schemas/basicsRatings.parquet*:

```
message basicsRatings {
    required binary ttconst (STRING);
    required binary titleType (STRING);
    required binary primaryTitle (STRING);
    required binary originalTitle (STRING);
    required boolean isAdult;
    optional int32 startYear (UINT_16);
    optional int32 endYear (UINT_16);
    optional int32 runtimeMinutes (UINT_16);
    optional group genres (LIST) {
        repeated binary genre (STRING);
    }
    optional float avgRating;
    optional int32 numVotes (UINT_32);
}
```

A conversão é imediata nos campos do tipo *string* e *boolean*. No entanto, quanto aos campos com valores de números inteiros, ajustamos de forma a que não fosse gasto espaço desnecessariamente. Por exemplo, colocamos com as anotações `UINT` os campos que só podem ter valores inteiros positivos, e limitamos os *bits* de acordo com um valor máximo razoável (`UINT_16` guarda inteiros até $2^{16} - 1$, que é suficiente para valores tipo ano e duração de vídeo, mas insuficiente para número de votos).

3.2.2 Descrição

O *Mapper* utilizado neste job é o BasicsRatingsParquetMapper. Na fase de *setup*, carrega em memória o ficheiro de *ratings* (existente em cache) para posteriormente fazer a correspondência na execução do map. O ficheiro de ratings pode estar comprimido, pois está implementada a leitura recorrendo a um *CompressionInputStream* se necessário, presente na class *IO* do package *Common*. Este também pode estar num bucket, pois também está implementada a verificação para aceder ao *filesystem* correto.

Este mapper apenas cria um record e preenche-o com os dados relativos a cada entrada. Este job não necessita de *reducers*.

3.2.3 Execução

As diretorias introduzidas tanto podem ser *hdfs:///* como *gs://*. Para executar este job, basta executar um comando semelhante a este:

```
$ ./cluster.sh submit <PATH>/ggcd_hadoop-1.0.jar Jobs.BasicsRatings --input=<PATH>
--output=<PATH> --schemas=<PATH> --ratings=<PATH>
```

3.3 YearMovie

Este *Job* corresponde à tarefa pedida para resolução do objetivo 2, para cada ano, obter o número total de filmes, o filme com mais votos, e a lista com o 10 filmes com melhor classificação.

3.3.1 Esquema *Parquet* da projeção

De forma a não importar dados desnecessário, foi definido um esquema apenas com os dados utilizados para o job, definido no ficheiro *schemas/basicsRatingsProjectionForYearMovie.parquet*:

```
message basicsRatings {
    required binary ttconst (STRING);
    required binary titleType (STRING);
    required binary primaryTitle (STRING);
    optional int32 startYear (UINT_16);
    optional float avgRating;
    optional int32 numVotes (UINT_32);
}
```

3.3.2 Esquema *Parquet* de output

Definimos o seguinte esquema para guardar o output, presente no ficheiro *schemas/yearMovie.parquet*:

```
message yearMovie {
    required int32 startYear (UINT_16);
    required int32 totalMovies (UINT_32);
    optional group mostVotedMovie {
        required binary ttconst (STRING);
        required binary primaryTitle (STRING);
        required int32 numVotes (UINT_32);
    }
    required group top10RatedMovies (MAP) {
        repeated group topRankedMovie {
            required binary rank (UTF8);
            optional group movieRatingInfo {
                required binary ttconst (STRING);
                required binary primaryTitle (STRING);
                required float avgRating;
            }
        }
```

```
    }
}
```

Escolhemos guardar o top 10 de filmes com melhor classificação num estrutura `MAP`, dessa forma conseguimos numerar o ranking destes, e caso não existam 10 filmes com classificação, aparece o valor a *null* para um dado rank, de 1 a 10. Assim da mesma forma que aparece a *null* o campo do filme com mais votos, se não existir nenhum filme com votos.

3.3.3 Descrição

O mapper correspondente a este job é o YearMovieMapper, que lê os ficheiro proveniente do ficheiro parquet, e escreve como key uma classe composta por startYear e avgRating. Como value, utiliza uma classe que implementa a classe *Writable*, com os campos necessários para o reducer.

De forma a aproveitar o *shuffle* antes do reduce, implementamos um *secondary sort* de forma a que os filmes fossem ordenados pelo avgRating, descendente. Para tal, implementamos as seguintes classes:

YearRatingPair

Guarda o valor da *natural key*, que é o startYear, e junta a esse valor o atributo sobre o qual queremos fazer a ordenação, o avgRating.

YearMovieData

Guarda os atributos necessários para a etapa do reducer, que são os atributos presentes no esquema da projeção *parquet*, excluindo os atributos usados na key, sendo esses *startYear* e *avgRating*.

YearRatingPartitioner

Utilizado para a repartição das entradas proviniente dos Mappers para os Reducers. Implementa o *Rehash-Partitioner*, que de acordo com a documentação, é sugerido para keys do tipo *Integer* (corresponde ao nosso caso, startYear) com padrões de distribuição simples.

YearRatingGroupingComparator

Faz a comparação das keys tendo em conta apenas o atributo startYear.

A etapa do Mapper passa por criar a *composite key*, representada por uma instância de *YearRatingPair*, e criar o *value*, que será representado por uma instância de *YearMovieData*. Tendo implementado o secondary sort, na etapa do Reducer basta apenas percorrer o *iterable* de *YearMovieData* uma vez, para encontrar o filme com mais votos, e fazer a contagem simultaneamente. Desta forma, evitamos fazer a ordenação dos filmes no Reducer para obter o top 10.

3.3.4 Reducers

Testamos a execução deste job recorrendo a 1 Reducer (comportamento por omissão), recorrendo a 4 Reducers (total de workers segundo a arquitetura demonstrada em Cluster, e a 8 Reducers (para analisar comportamento).

Em termos de tempo de execução, obtivemos os seguintes valores:

Influência de Reducers	
Reducers	**CPU time spent (ms)**
1	137880
4	72740
8	92050

Tendo em conta que não foi especificada a necessidade de ordenação do output final, obtemos uma vantagem significativa ao escolher a abordagem com vários Reducers. No entanto, obtivemos melhor resultado ao distribuir uma tarefa de Reduce por cada worker, aumentar este número para além desse *ratio* acaba por introduzir *overhead* desnecessário.

3.3.5 Execução

Para executar este job, basta executar um comando semelhante a este:

```
$ ./cluster.sh submit <PATH>/ggcd_hadoop-1.0.jar Jobs.YearMovie --input=<PATH>
--output=<PATH> --schemas=<PATH>
```

3.4 MovieSuggestion

Este *Job* corresponde à tarefa pedida para resolução do objetivo 3, para cada filme, recomendar o outro do mesmo género que tenha a melhor classificação.

3.4.1 Esquema *Parquet* da projeção

Mais uma vez, de forma a não importar dados desnecessário, foi definido um esquema apenas com os dados utilizados para o job, definido no ficheiro *schemas/basicsRatingsProjectionForMovieSuggestion.parquet*:

```
message basicsRatings {
    required binary ttconst (STRING);
    required binary titleType (STRING);
    required binary primaryTitle (STRING);
    optional group genres (LIST) {
        repeated binary genre (STRING);
    }
    optional float avgRating;
}
```

3.4.2 Estrutura do ficheiro de output

Como pedido no enunciado, este job tem como output um ficheiro de texto. Tentamos seguir a formatação do *imdb* para que fosse uma estrutura familiar. Dessa forma, os ficheiros de output colocam cada filme numa linha, com vários campos separados por tabs. São estes os campos, por esta ordem:

- ttconst

- primaryTitle
- genre
- suggestedTtconst
- suggestedPrimaryTitle
- suggestedAvgRating

3.4.3 Descrição

Este job tem uma fluxo semelhante ao job anterior, pelo que também implementa uma classe *Writable* para passagem de dados entre Mapper e Reducer, assim como também implementa *secondary sort* pelo atributo *avgRating*. A principal diferença é o formato de output.

Neste caso, a *natural key* utilizada é o género principal do filme, sendo depois complementada com o *avgRating* para a ordenação. Foram então implementadas as seguintes classes:

GenreRatingPair

Guarda o valor da *natural key*, que é o *genre*, e junta a esse valor o atributo sobre o qual queremos fazer a ordenação, o *avgRating*.

MovieSuggestionData

Guarda os atributos necessários para a etapa do reducer, que são os atributos presentes no esquema da projeção *parquet*, excluindo os atributos usados na key, sendo esses *genre* e *avgRating*.

MovieSuggestionPartitioner

Utilizado para a repartição das entradas proviniente dos Mappers para os Reducers. Implementa o *KeyFieldBasedPartitioner*, que foi o Partitioner com o qual conseguimos uma distribuição mais equilibrada entre reducers. Nunca será uma repartição totalmente equilibrada, pois há géneros muito mais populares do que outros, que acabam por ter muitos mais filmes do que outros géneros, como é o exemplo da comédia ou ação.

MovieSuggestionGroupingComparator

Faz a comparação das keys tendo em conta apenas o atributo *genre*.

A etapa do Mapper passa por simplesmente criar a *composite key*, representada por uma instância de *GenreRatingPair*, e criar o *value*, que será reprensentado por uma instância de *MovieSuggestionData*. Posteriormente, o Reducer terá já o *iterable* ordenado, pelo que para todos os filmes irá recomendar o primeiro, à exceção dele próprio, pois o primeiro filme irá recomendar o segundo. Caso não haja filmes com classificação, os três campos relativos ao filme sugerido irão conter `\N`, à semelhança de campos vazios no *dataset* do *imdb*.

3.4.4 Reducers

Mais uma vez, testamos a execução deste job recorrendo a 1 Reducer (comportamento por omissão), recorrendo a 4 Reducers (total de workers segundo a arquitetura demonstrada em Cluster, e a 8 Reducers (para analisar comportamento).
Em termos de tempo de execução, obtivemos os seguintes valores:

Influência de Reducers	
Reducers	**CPU time spent (ms)**
1	113710
4	82750
8	88320

Como no caso anterior, não foi especificada a necessidade de ordenação do output final. Obtemos melhores resultados com o *ratio* de 1:1 entre workers e Reducers.

3.4.5 Execução

Para executar este job, basta executar um comando semelhante a este:

```
$ ./cluster.sh submit <PATH>/ggcd_hadoop-1.0.jar Jobs.MovieSuggestion --input=<PATH>
--output=<PATH> --schemas=<PATH>
```

3.4.6 Exemplo de output

Temos abaixo algumas linhas do resultado da execução deste Job:

```
tt10445280   Uncivilized Adventure tt13587614 Surveillance Camera Man 1-8 9.6
tt13587614 Surveillance Camera Man 1-8 Adventure tt10445280 Uncivilized 9.7
tt2091864 Black Hearts Adventure tt10445280 Uncivilized 9.7
...
tt0215851 Hockey: Canada's National Game Short \N \N \N
tt10564884 Escape Short tt0215851 Hockey: Canada's National Game 5.2
```

3.5 ParquetToJson

Este Job é um extra que implementamos que faz a conversão de um ficheiro *parquet* para um equivalente em JSON. Da forma que foi implementado, está preparado para ter como únicos argumentos uma diretoria de input e outra de output. Não está dependente de nenhum *schema* específico, porque quisemos que esta abordagem fosse geral, assim pode converter qualquer ficheiro e *schema*. A motivação por detrás desta Job foi a necessidade de verificação do output dos Jobs BasicsRatingsParquet e YearMovie de forma fácil, e aprofundar conhecimentos, pelo que foi necessário investigar melhor implementações de classes adicionais que não foram desenvolvidas nas aulas, nomeadamente RecordWriters e FileOuputFormatters.

3.5.1 Descrição

Tal como acontece nos Jobs YearMovie e MovieSuggestion, recebe como input uma diretoria a conter toda a informação relativa a um dado *schema* e *dados*.

Na fase do Mapper, este recorre ao método *fieldToNode* da classe *IO*, presente no package *Common*. Este método, no caso de encontrar tipos compostos (por exemplo `MAP` ou `LIST`), faz chamadas recursivas até chegar a tipos elementares (por exemplo `INT` ou `STRING`).

No entanto, este Job tem dois Mappers possiveis, que fazem a escrita dos dados de forma diferente, o que resulta em output de ficheiros JSON estruturalmente diferentes. O Mapper mais geral é o ParquetToJsonEntriesMap, que faz a conversão dos *records* para JSON, e coloca todos os *records* no mesmo array, identificado pela key *entries*. O Mapper ParquetToJsonIDsMap assume que o primeiro *field* do *schema* irá ser utilizado como key para representar o *record*. Por omissão, é utilizado o primeiro Mapper referido, por ser mais geral. No entanto, a utilização da opção `--firstAsId=true` faz uso do segundo Mapper. No fim desta secção iremos exemplificar ambas as abordagens.

Relativamente às classes implementadas par obter o output em JSON, são as seguints:

JsonOutputFormat

Esta classe trata das interações com o *filesystem* e retorna o RecordWriter que contém o OutputStream para escrita nos ficheros de output.

JsonRecordWriter

Esta classe trata da escrita dos dados passados pelo Mapper. No entanto, a escrita só acontece no fim, na invocação do método *close*. Isto acontece de forma a que seja possível atualizar valores do JSON resultante, ao contrário de escrever os valores assim que são recebidos. Desta forma, a utilização do método *merge* presente nessa classe, faz com seja possível concatenar arrays ou juntar objetos que tenham a mesma key.

3.5.2 Execução

Para executar este job, basta executar um comando semelhante a este:

```
$ ./cluster.sh submit <PATH>/ggcd_hadoop-1.0.jar Jobs.ParquetToJson --input=<PATH>
--output=<PATH>
```

3.5.3 Exemplo de execução - Entries

Para este exemplo, foi utilizado como input o output do Job BasicsRatingsParquet:


```
{
   "entries":[
      {
         "ttconst":"tt0000001",
         "titleType":"short",
         "primaryTitle":"Carmencita",
         "originalTitle":"Carmencita",
         "isAdult":false,
         "startYear":1894,
         "endYear":null,
         "runtimeMinutes":1,
         "genres":[
            "Documentary",
            "Short"
         ],
         "avgRating":5.6,
         "numVotes":1695
      }
   ]
}
```


Foi colocado apenas um *record* convertido para demonstração, no ficheiro real estarão todos os *records* no array de *entries*.

3.5.4 Exemplo de execução - IDs

Para este exemplo, foi utilizado como input o output do Job YearMovie. Vamos apresentar apenas uma entrada, as restantes terão o mesmo aspeto:


```
{
   "1917":{
      "totalMovies":23,
      "mostVotedMovie":{
         "ttconst":"tt0007778",
         "primaryTitle":"Castles for Two",
         "numVotes":15
      },
      "top10RatedMovies":{
         "1":{
            "ttconst":"tt0007778",
            "primaryTitle":"Castles for Two",
            "avgRating":6.9
         },
         "2":{
            "ttconst":"tt1167871",
            "primaryTitle":"Venchal ikh satana",
            "avgRating":5.6
         },
         "3":{
            "ttconst":"tt0302667",
            "primaryTitle":"The Irish Girl",
            "avgRating":2.2
         },
         "4":null,
         "5":null,
         "6":null,
         "7":null,
         "8":null,
         "9":null,
         "10":null
      }
   }
}
```

4 | 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 primeiro argumento a localização do ficheiro *title.basics.tsv*, como segundo argumento a localização do ficheiro *title.ratings.tsv*, e como terceiro argumento a diretoria para onde será copiado o output.

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

- Compilar o JAR
- Criar o cluster
- Importar dados para HDFS
- Executar os jobs
  - BasicsRatingsParquet
  - YearMovie
  - MovieSuggestion
  - ParquetToJson com input BasicsRatingsParquet
  - ParquetToJson com input YearMovie e `--firstAsId`
- Copiar os outputs para a diretoria indicada
- Eliminar o cluster

Execução

O script verifica se os argumentos dados são ficheiros existentes, assim como verifica se a pasta para output existe. 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. Para executar este script, basta executar um comando semelhante a este:

```
$ ./run.sh <BASICS> <RATINGS> <OUTPUT>
```

5 | Conclusão

Com este trabalho prático, conseguimos aplicar os conhecimentos obtidos nas aulas, e ainda aprofundar estes conhecimentos, com a implementação do objetivo 3 e das funcionalidades extra que foram aqui descritas. Com este trabalho, conseguimos interagir a vários nível em termos de processamento de dados, desde a criação do cluster, implementação das tarefas de Map e Reduce, e ainda com a implementação de classes adicionais necessárias para adaptar o processamento a um caso específico, neste caso ao *dataset* proveniente do *imdb*. Para lém disso, conseguimos também implementar persistência de dados, mesmo quando o cluster for eliminado, recorrendo à *Google Cloud Storage*.