2º Trabalho Prático
Gestão de Grandes Conjuntos de Dados

Grupo 9
Bruno Santos[PG44414], Nelson Estevão[A76434], Rui Reis[A84930] e Susana Marques[A84167]

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal

1 Introdução

Neste trabalho prático é requerida a concretização e avaliação experimental de tarefas de armazenamento e processamento de dados através do uso da ferramenta computacional *Spark*.

Por forma a realizar estas tarefas são utilizados os dados públicos do *IMBD*, que se encontram disponíveis em:

https://www.imdb.com/interfaces/

Ao longo deste documento serão expostas todas as opções tomadas durante a implementação das tarefas de processamento de dados pedidas, incluindo as decisões que o grupo fez a nível de algoritmos e parÂmetros de configuração.

De seguida, são apresentadas todas as instruções que permitem executar corretamente as tarefas desenvolvidas.

No final encontram-se exibidos os objetivos atingidos após a realização das tarefas propostas.

2 1.ª Tarefa

Os dados a utilizar no desenvolvimento deste projeto estão contidos em quatro *datasets*, `name.basics.tsv`, `title.basics.tsv`, `title.principals.tsv` e `title.ratings.tsv`. Os *datasets* contém informação relativa a títulos de filmes e séries televisivas e os actores que participaram nesses mesmos títulos.

O *download* dos dados pode ser feito diretamente dos servidores do IMDb e carregados para o *Hadoop Distributed File System* (HDFS). Obter os datasets pode ser feito utilizando o utilitário `curl`, de seguida descomprimir através do programa `gunzip` e por fim feito o carregamento utilizando o próprio `hdfs`. De seguida é mostrado um exemplo para a estrutura do comando completo para o caso do *dataset* `name.basics.tsv`. Este passo é feito no *script* `data/import_data.sh`.

```
curl https://datasets.imdbws.com/name.basics.tsv.gz \
    | gunzip \
    | hdfs dfs -put - /name_basics/name.basics.tsv
```

Todos os passos são abstraídos em *scripts* para cada tipo de ambiente em que possamos estar a correr o sistema, tal como é apresentado na secção 5.

O passo seguinte foi a criação das tabelas adequadas a cada um dos *datasets* para o Hive Metastore. Este passo é feito no *script* `data/create_tables.sh`. Esta etapa consiste em 3 fases, primeiro criar a tabela representativa de um dado *dataset* referenciando a localização dos dados.

```
create external table name_basics(
    nconst string,
    primaryName string,
    birthYear integer,
    deathYear integer,
    primaryProfession array<string>,
    knowForTitles array<string>)
    row format delimited
    fields terminated by '\t'
    collection items terminated by ','
    lines terminated by '\n'
    stored as textfile
    location 'hdfs:///name_basics'
    tblproperties ("skip.header.line.count"="1");
```

A segunda fase passa por criar uma tabela que contenha apenas os dados necessários para responder às tarefas seguintes num formato eficiente como é o caso do formato *parquet*.

```
create table name_basics_pq(
    nconst string,
    primaryName string,
    birthYear integer,
    deathYear integer,
    primaryProfession array<string>,
    knowForTitles array<string>)
    stored as parquet;
```

Por fim, é necessário passar os dados para este formato a partir do *dataset* original no próprio HDFS.

```
insert overwrite table name_basics_pq select * from name_basics;
```

Este processo é repetido para os 4 *datasets* a utilizar.

3 2.ª Tarefa

Todas as *queries* pedidas tiram partido da projeção mais adequada de dados de tabelas em esquema formato Parquet, de forma a completar a *querie* com a menor informação possível, assim, somos capazes de otimizar as *queries* tirando partido da organização híbrida do armazenamento. Em cada *querie* é explicada a escolha da projeção utilizada usando Spark RDDs.

3.1 Top Genres

Nesta subtarefa é pedido o género mais comum em cada década.

O codigo-fonte da presente *query* encontra-se no package ggcd.tp2.queries e na classe denominada *TopGenresQuery*.

Durante o processamento inicial do ficheiro *title_basics.pq*, ignorou-se o respetivo cabeçalho. Em seguida por cada *title* gerou-se um par onde por cada género que se iterou obteve-se a década dividindo o *StartYear* por 10 e um contador para este par que toma o valor de 1.

A seguir efetuou-se a soma destes contadores unitários e agrupou-se os resultados pelo número de géneros e tendo em conta que se pretendia obter um resultado ordenado, a lista dos resultados com os valores (década e género) foi clonada e efetuada uma operação de ordenação.

```
public void run(final SparkSession sparkSession) {
        List<Tuple2<Integer, Tuple2<String, Integer>>> c =
            sparkSession
                .table(Configuration.TITLE_BASICS_PQ)
                .toJavaRDD()
                .filter(r -> !r.isNullAt(5) && !r.isNullAt(8))
                .flatMapToPair(
                    r -> r.getList(8).stream()
                    .map( p ->  new Tuple2<>(
                        new Tuple2<>(
                        r.getInt(5) / 10,
                        (String) p), 1)) .iterator())
                .reduceByKey(Integer::sum)
                .mapToPair(a -> new Tuple2<>(a._1._1, new
                ↪  Tuple2<>(a._1._2, a._2)))
                .groupByKey()
                .mapToPair(
                     a -> {
                    List<Tuple2<String, Integer>> ls =
                    ↪  Lists.newArrayList(a._2);
                    ls.sort((b, dc) -> (-1) * Long.compare(b._2,
                    ↪  dc._2));
                    return new Tuple2<>(a._1, ls.get(0));
                                })
                .sortByKey()
                .collect();
```

3.2 Season Hits

Nesta subtarefa é pedido o título mais bem classificado em cada ano.

O codigo-fonte da presente *query* encontra-se no package ggcd.tp2.queries e na classe denominada *SeasonHitsQuery*.

Usou-se a tabela *title_ratings.pq* para obter-se os resultados pretendidos.

Em primeiro lugar, durante o processamento inicial da tabela, ignorou-se o respetivo cabeçalho. Posteriormente gerou-se um par onde para cada título correspondia o ano em que começou.

A seguir fez-se um *join* para unir as classificações (*ratings*) e um *swap* colocando as classificações como chave e o valor com o título e o ano correspondente.

Agrupou-se os resultados pela classificação e tendo em conta que se pretendia obter um resultado ordenado como na alínea anterior, a lista dos resultados foi clonada e efetuada uma operação de ordenação.

```
public void run(final SparkSession sparkSession) {
    // Get the ratings in RDD, for simplicity
    JavaPairRDD<String, Double> ratings =
        sparkSession
            .table(Configuration.TITLE_RATINGS_PQ)
            .toJavaRDD()
            .filter(r -> !r.isNullAt(0) && !r.isNullAt(1))
            .mapToPair(
                r -> new Tuple2<>(
                r.getString(0),
                Double.parseDouble(r.getString(1))));

    List<Tuple2<Integer, Tuple2<String, Double>>> c =
            sparkSession
                .table(Configuration.TITLE_BASICS_PQ)
                .toJavaRDD()
                // TCONST (0) and Startyear (5)
                .filter(r -> !r.isNullAt(0) && !r.isNullAt(5))
                .mapToPair(p -> new Tuple2<>(p.getString(0),
                ↪  p.getInt(5)))
                .join(ratings)
                .mapToPair(p -> new Tuple2<>(p._2._1, new
                ↪  Tuple2<>(p._1, p._2._2)))
                .groupByKey()
                .mapToPair(
                    a -> {
                        List<Tuple2<String, Double>> ls =
                        ↪  Lists.newArrayList(a._2);
                        ls.sort((b, dc) -> (-1) *
                        ↪  Double.compare(b._2, dc._2));
                        return new Tuple2<>(a._1, ls.get(0));
                            })
                .sortByKey()
                .collect();
```

3.3 Top 10

Nesta subtarefa é pedido o cálculo dos 10 atores que participaram em mais filmes distintos.

O codigo-fonte da presente *query* encontra-se no package ggcd.tp2.queries e na classe denominada *Top10ActorsQuery.*

Usou-se a tabela *title_principals.pq* para obter-se os resultados pretendidos.

Durante o processamento inicial desta tabela, é, tal como seria de esperar, ignorado o respetivo cabeçalho. Em seguida filtrou-se os registos que realmente correspondiam a atores e atrizes.

Dito isto, por cada ator encontrado é gerado um par com um contador de filmes, sendo que este toma o valor de 1 por cada filme **distinto** em que o ator participa. É efetuado a soma destes contadores unitários e depois ordenados pelo seu valor (número de filmes).

De modo a obter o *top 10*, é efetuado um *take* de 10 unidades ao *RDD* previamente armazenado em *cache.*

```
public void run(final SparkSession sparkSession) {
 JavaRDD<Row> mainData
 ↪  =sparkSession.table(Configuration.TITLE_PRINCIPALS_PQ).toJavaRDD();
      List<Tuple2<String, Integer>> c =
         mainData.filter(r -> !r.isNullAt(0) && !r.isNullAt(2)
          && !r.isNullAt(3))
             // only maintain actors or actresses
             .filter(
              r -> r.getString(3).equals("actor")
                   || r.getString(3).equals("actress"))
                  .mapToPair(
                       r -> new Tuple2<>(new Tuple2<>(r.getString(2),
                       ↪  r.getString(0)), 1))
                  .distinct()
                  .mapToPair(p -> new Tuple2<>(p._1._1, p._2))
                  .reduceByKey(Integer::sum)
                  .map(t -> t)
                  .sortBy(a -> a._2, false,
                  ↪  mainData.partitions().size())
                  .take(10);
```

4 3.ª Tarefa

A terceira tarefa consiste em desenvolver um conjunto de operações em Spark RDD ou SQL e que consubstanciem para um único único ficheiro.

Decidimos resolver as operações usando SQL, posto que SQL é mais intuitivo do que spark RDD e deste modo poderíamos contrastar o desenvolvimento usando SQL e spark RDD.

A presente tarefa tem como objetivo armazenar o resultado das subsequentes operações num único ficheiro. Decidimos que o ficheiro final fosse armazenado em formato parquet em detrimento de .csv, uma vez que é um formato mais compacto e permite uma eficiência maior na consulta dos dados. Para que o armazenamento final fosse possível, foi concebida uma tabela e inserida no Hive metasore. A tabela é denominada de actor_pages_pq e contém as seguintes colunas:

- **nconst** Identificação do ator/atriz
- **name** Nome do ator/atriz
- **age** Idade do ator/atriz
- **titlesNumber** Número de títulos do ator/atriz
- **activityYears** Anos de atovidade do ator/atriz
- **averageRating** Média de classificações nos títulos em que o ator/atriz participou
- **top10titles** Top 10 dos títulos com melhor classificação do ator/atriz
- **top10generation** Top 10 dos atores que nasceram na mesma década
- **friends** Lista de atores que contracenaram com o presente ator/atriz

Depois de o resultado final ter sido inserido na tabela, esta é escrita num ficheiro e armazenado no HDFS.

4.1 Segmento - Base

O codigo-fonte do presente segmento encontra-se no package ggcd.tp2.segments e na classe denominada BaseSegment.

O ficheiro final, páginas de ator, consiste em nove colunas. Este é responsável por obter os dados das primeiras 5 colunas: nconst, name, age, titlesNumber e activityYears. Para tal foi necessário fazer um *inner join* entre as quatro tabelas: name_basics_pq; title_basics_pq; title_ratings_pq; title_principals_pq.

As colunas nconst, name e age foram obtidas da tabela name_basics_pq. A coluna age resultou da diferença entre o ano de falecimento da passoa (deathYear) se não nulo, se nulo considera-se o ano atual, e do ano de nascimento (birthYear).

```
nvl(first(nb.deathYear),
    year(current_date())) - first(nb.birthYear)
```

Na presente query, ao realizar o *group by* por nconst foi possível obter as colunas titlesNumber, activityYears e averageRating usando funções de agregação. Para a coluna titlesNumber usou-se a função *count(distinct tp.tconst)*, para a coluna averaverageRating a função *avg(tr.averageRating)* e para a coluna activityYears, anos de atividade da pessoa em questão, calculou-se a diferença entre o maior valor entre o valor máximo da coluna endYear e o valor máximo da coluna startYear e o valor mínimo da coluna startYear para a pessoa presente no group by.

```
greatest(max(tb.endYear),
    max(tb.startYear)) - min(tb.startYear) as activityYears
```

4.2 Segmento - Hits

O codigo-fonte do presente segmento encontra-se no package ggcd.tp2.segments e na classe denominada HitsSegment.

O desígnio do segmento Hits consiste em obter o top 10 dos títulos mais bem classificados em que cada pessoa participou. Para tal foi necessário as tabelas title_principals_pq para obter todos os títulos de uma dada pessoa e title_ratings_pq para obter os ratings dos respetivos títulos.

É necessário obter as 10 primeiras linhas de cada grupo, cujo agrupamento é feito por nconst. Não existe sintaxe nativa no SQL para realizar este procedimento, posto isso, inicialmente atribuímos número a linhas iniciando no número 1 para cada grupo e posteriormente filtramos as 10 primeiras linhas de cada grupo.

Assim sendo, depois de realizado o join entre as tabelas, atribuímos números a linhas agrupadas por nconst e ordenadas por averageRating usando a função dense_rank() e as clausulas over, partition by e order by.

```
select nconst, tconst, averageRating, dense_rank()
over(partition by nconst order by averageRating desc) as rank
```

Dispondo das linhas ordenadas dentro do grupo, foi realizado um filtro para recolher apenas as 10 primeiras linhas. Uma vez que na tabela final a coluna top10titles contém uma lista de strings, e usando a clausula group by e a função collect_list(t2.tconst), para cada pessoa(nconst) foi colocado numa lista o top 10 anteriormente calculado. Deste modo o retorno do presente segmento contém as colunas nconst e top10titles.

4.3 Segmento - Generation

O codigo-fonte do presente segmento encontra-se no package ggcd.tp2.segments e na classe denominada GenerationSegment.

O desígnio deste segmento consiste em obter o top 10 dos atores/atrizes de cada geração (que nasceram na mesma década). O procedimento foi semelhante ao descrito para o segmento anterior, mas neste caso envolve um maior número de tabelas.

Inicialmente foi calculado para cada ator o número de filmes que participou, usando a tabela title_principals_pq, e armazenado o resultado na tabela temporária tfilmes. De seguida foi realizado um inner join com a tabela name_basics_pq para obter, para cada ator, o número de filmes e a sua decada de nascimento. Posteriormente foi feito um particionamento por década e por cada década foi colocado os 10 atores com maior número de filmes, sendo assim, foi concebida uma tabela temporária cujos registos são a decada e uma lista com os 10 atores com mais filmes da presente década. Por fim, foi realizado um inner join com a tabela name_basics_pq e obtido uma tabela em que os registos são aos atores e uma lista dos 10 atores da mesma geração com mais filmes, tenso a tabela resultante as colunas nconst e top10generation.

4.4 Segmento - Friends

O codigo-fonte do presente segmento encontra-se no package ggcd.tp2.segments e na classe denominada FriendsSegment. O seu desígnio consiste em obter, para cada ator, os outros atores que participaram nos mesmos títulos.

Inicialmente foram selecionadas as colunas tconst e nconst da tabela title_principals_pq para, de seguida, fazer um inner join com a tabela title_principals_pq. Usando a claúsula group by pela coluna nconst e usando a função collect_set(tf.nconst), a tabela resultante contem, para cada ator, um conjunto (elimina os valores repetidos) dos atores que contracenaram nos mesmos filmes. As colunas da tabela final possuiem os camapos nconst e friends, sendo friends um *alias* de collect_set(tf.nconst).

4.5 Segmento - PagesAtor

O codigo-fonte do presente segmento encontra-se no package ggcd.tp2.segments e na classe denominada PagesAtorSegment.

Este segmento tem como objetivo realizar o join entre os segmentos anteriores (base, hits, generation e friends) e, deste modo, construir a tabela final com as nove colunas. O join é realizado através da coluna em comum entre as quatro tabelas anteriores, denominada por nconst.

4.6 Query - PagesAtorBuild

O código-fonte da presente query encontra-se no package ggcd.tp2.query e na classe denominada PageActorBuildQuery.

O objetivo da presente query é, depois de o segmento anterior (PagesAtor) ter sido executado, e com isso a tabela final com as nove colunas ter sido construída, inserir o resultado na tabela actor_pages_pq (construída e inserida no hive metastore) e posteriormente a tabela ser escrita em formato parquet no HDFS, de forma a ser guardada de forma persistente.

Após estas fases, o resultado final é um ficheiro, em formato parquet, denominado *actor_pages_pq* e armazenado no HDFS cujos registos são as informações pedidas no enunciado para para ator/atriz.

5 Guia de Utilização

A utilização do programa desenvolvido pode ser feito em dois ambientes, através de docker *containers* ou então utilizando um serviço de *Platform as a Service* (PaaS) como é o caso do Google Cloud através do serviço *Dataproc*.

5.1 Configuração com Docker

A criação dos containers de docker necessários são abstraídos em dois `docker-compose.yml` baseados nos repositórios da Big Data Europe que são o docker-hive e o docker-spark. A criação de todos os containers e a ligação entre eles está simplificada através do *script* `bin/setup`.

A criação dos *containers* e coloca-los a correr é apenas o primeiro passo. Além disso, é necessário o *download* dos dados a utilizar para o (HDFS). Essa tarefa pode ser executada com o *script* `docker-hive/setup`, onde também são criadas as tabelas e feito a referênciaa aos dados assim como a inserção dos mesmos (no caso dos dados em formato *parquet*).

A configuração através de docker *containers* exige presente no sistema o docker e o docker compose.

5.2 Configuração do Cluster no Google Cloud

O *cluster* utilizando o serviço do *Dataproc* fornecido pelo Google Cloud foi criado através do comando apresentado de seguida.

```
gcloud beta dataproc clusters create ggcd-g9-tp2 \
  --region us-central1 --zone us-central1-c \
  --master-machine-type n1-standard-4 \
  --num-workers 2 \
  --master-boot-disk-size 500 \
  --worker-machine-type n1-standard-4 \
  --worker-boot-disk-size 500 \
  --image-version 2.0-debian10 \
  --project ggcd-grupo-9
```

À semelhança do caso anterior, é necessário importar os dados para o Hive Metastore é possível ser feito através do script `glcoud/setup` onde é feito o upload dos *datasets* necessários e criadas as tabelas necessárias. Desta forma, é possível ter uma configuração do sistema simples de utilizar.

A configuração do *cluster* através da Google Cloud Plataform existe a instalação e correta configuração do gcloud cli.

5.3 Execução de Tarefas

As tarefas podem ser executadas através do script `bin/run` que tem a seguir interface de utilização.

```
USAGE
  bin/run <environment> <args>
  bin/run [options]

ENVIRONMENTS
  --local           To run in the docker container environment.
  --gcloud          To run in the google cloud environment.

OPTIONS
  -h --help         Show this screen.
  -v --version      Show version.
```

As *flags* `--local` e `--gcloud` permitem optar por correr a tarefa localmente (através de docker *containers* ou remotamente, no *cluster* do serviço *Dataproc*.

O argumento `<args>` consiste nos argumentos do próprio programa desenvolvido e são apresentados de seguida.

- `top-genre` - Género mais comum em cada década.
- `season-hist` - Título mais bem classificado em cada ano.
- `top-actors` - Top 10 dos atores que participaram em mais títulos diferentes.
- page-actor-build - Capaz de construir a tabela de páginas de actor com várias informações relacionadas com o mesmo.
- page-actor - Igual ao anterior, mas sem a criação da tabela e apenas uma pequena visualização da mesma.

A documentação sobre a utilização do programa pode ser pedida executando `bin/run help`.

6 Conclusão

Numa fase de análise de resultados, observou-se que a solução implementada cumpriu todos os objetivos propostos, bem como solidificou os benefícios de computação distribuída para tratamento de grande conjunto de dados. 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*.

Podemos constatar também que a utilização de Spark SQL providencia mais informação sobre a estrutura dos dados e da computação que está a ser feita do que Spark RDD. Assim Spark SQL consegue fazer otimizações extra com essa maior quantidade de informação. Para além disso, é bastante mais intuitivo o uso de SQL ao uso de RDDs em *Java*.

Em comparação com o primeiro trabalho prático, onde se abordava o paradigma *MapReduce*, esta ferramenta apresenta diversas vantagens, nomeadamente a maior flexibilidade nas operações efetuadas sobre os dados.

Foi ainda possível observar maior eficiência a nível do tempo de resposta a eventos do que na arquitetura *MapReduce*. Outros dos grandes benefícios na utilização deste utensílio é a capacidade do mesmo em processar grandes quantidades de informação sem os mesmos estarem armazenados localmente, e ainda, o facto desta estratégia admitir o processamento de uma sequência de eventos infinita.