Queen Mary
University of London

Big Data Processing

Coursework 1

Submitted in partial fulfillment of the

requirements for the degree of

Master by Research in Computer Science

Author:

Etiene da Cruz Dalcol

170949419

Lecturer:

Dr. Felix Cuadrado

November 10, 2017

Introduction

This report elaborates the development of the first coursework of the Big Data Processing module. This coursework intends to give students an overview of using Hadoop for different text processing tasks required in the assigned. Besides Hadoop, the tools I used for the assignment were:

- Java - The programming language used to develop the jobs
- Ant - For compiling the code and building the Jar file
- Atom - A text editor
- Unix - Multiple useful commands were used to work with the outputs
- Google Sheets - For transforming tabular data into the charts you will see in the next sections

The dataset for the assignment consisted of a huge compilation of tweets related to the last Olympic Games, which happened in the city of Rio de Janeiro, Brazil, on 2016. The lines of the dataset follow this structure: **epoch_time;tweetId;tweet;device**. During the development, only the first and the third fields were used.

The full source code of this coursework, along with the unmodified outputs of the MapReduce jobs were compressed and delivered with this report.

Part A: Message Length Analysis

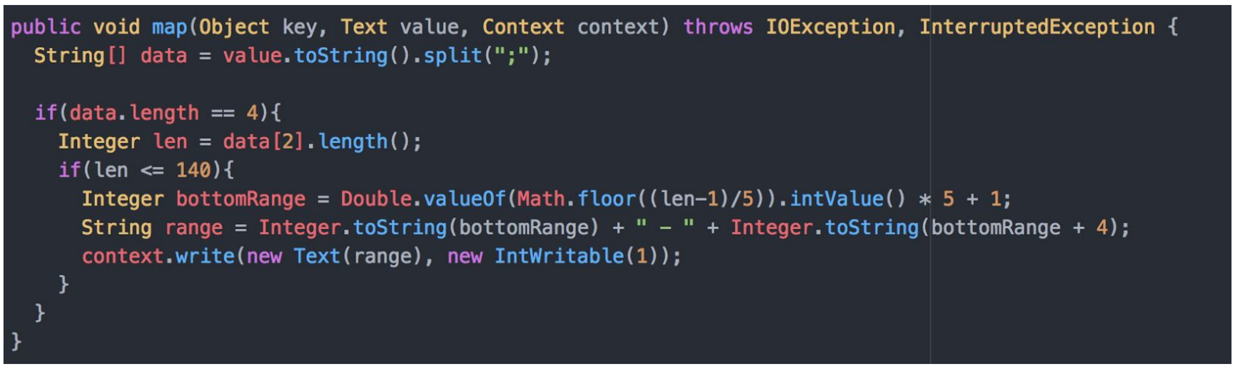
The goal of this part of the coursework, was to obtain a histogram plot of the length of the tweets in the dataset. For this task, 3 Java classes were created: `LengthAnalysis.java`, `LengthMapper.java` and `SimpleSumReducer.java`. The full code of these classes is available compressed along this report.

Run instructions:

```
ant clean dist
hadoop jar dist/EDCD3_BDP_Coursework1.jar LengthAnalysis /data/olympictweets2016rio out
hadoop fs -getmerge out length_analysis.txt
```

The latest command will merge the parts computed by the reducers and copy it to the local disk.

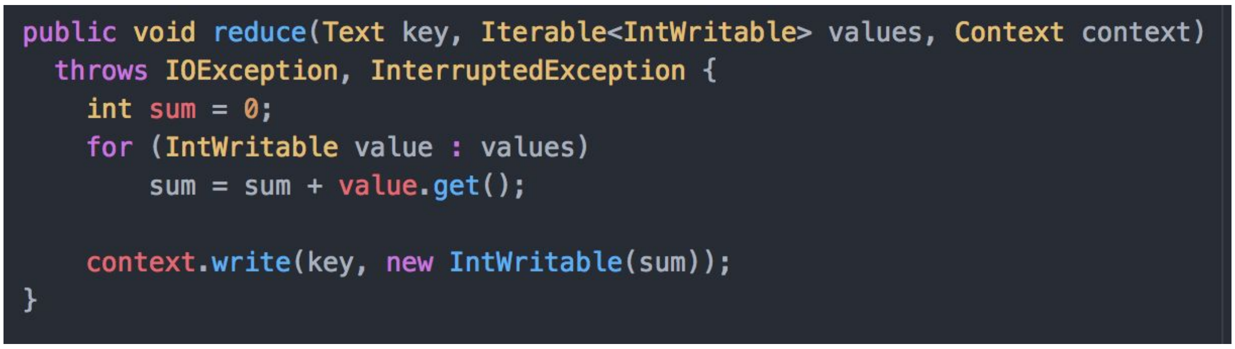
An extract of the mapper is as follows:



```java
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
  String[] data = value.toString().split(";");

  if(data.length == 4){
    Integer len = data[2].length();
    if(len <= 140){
      Integer bottomRange = Double.valueOf(Math.floor((len-1)/5)).intValue() * 5 + 1;
      String range = Integer.toString(bottomRange) + " - " + Integer.toString(bottomRange + 4);
      context.write(new Text(range), new IntWritable(1));
    }
  }
}
```

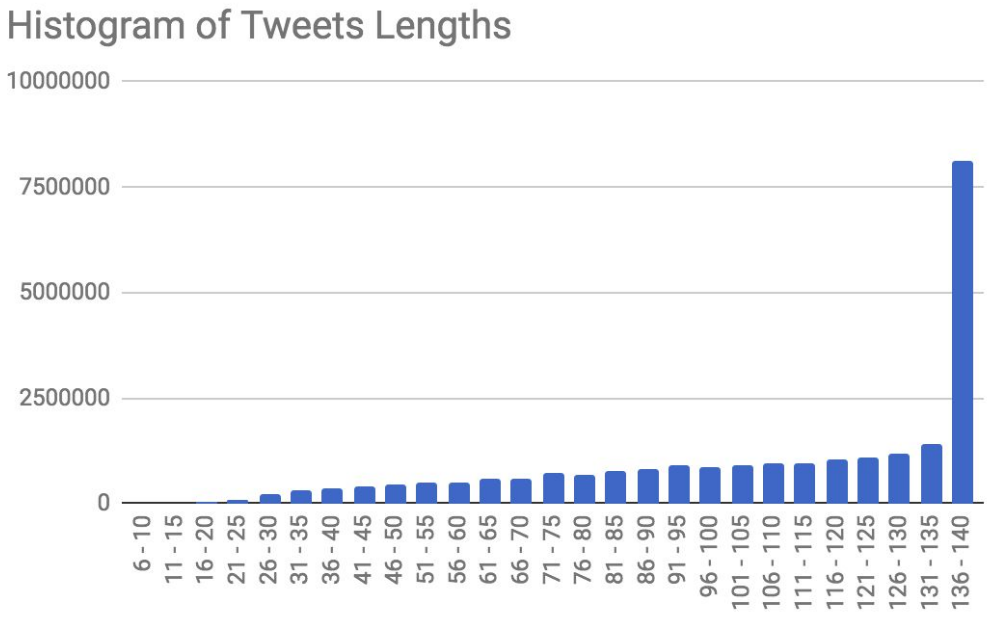
As shown on the program above, a simple technique was used to filter the input and discard tweets that were considered invalid. Two main criteria were used for the selection: if the line is comprised of 4 fields and if the field that corresponds to the tweet message contains up to 140 characters. The reason for that was that some lines either contain slightly broken data, which would cause the job to fail. **Similar filtering processes were used on all subsequent parts of the assignment.**

The length of the tweet was simplified to fit in a particular range for the histogram by calculating the bottom and the upper limits of the range. The range was used as the key and 1 was used as the value for the output of the mapper. The Reducer merely receives the the range and an array of 1's, which it them sums to output the range again but this time with the total count values as seen below:

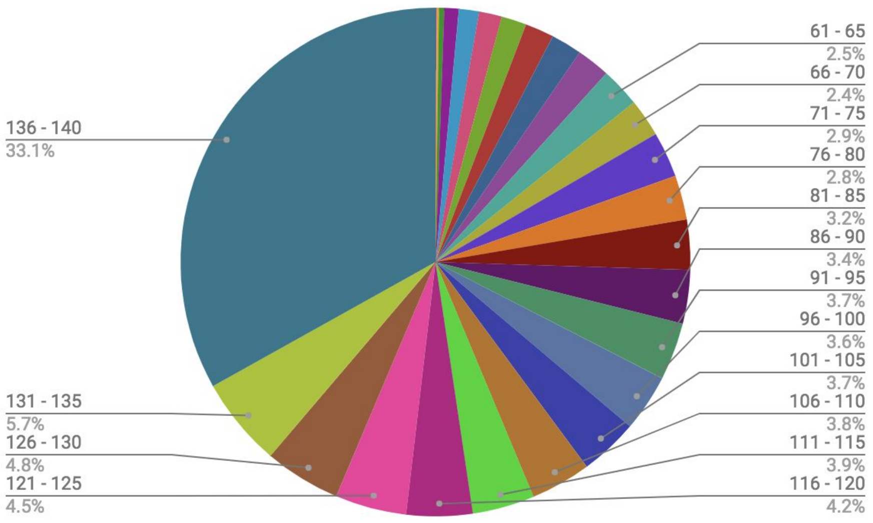


```java
public void reduce(Text key, Iterable<IntWritable> values, Context context)
  throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable value : values)
        sum = sum + value.get();

    context.write(key, new IntWritable(sum));
}
```

The unmodified output of the MapReduce job can be seen on the file `histogram_analysis.txt`, compressed with this report and the source code. This output was subsequently copied to a spreadsheet on Google Drive and manually sorted to obtain the graphic chart included below.



For my own amusement, I decided to also plot this as a pie chart. The results are very interesting. Over 30% of the tweets are “lengthy”, as in they either reach the limit of 140 characters or are very close, which might partially explain the recent controversial decision of Twitter to expand the character limit from 140 to 280.



Part B: Time Analysis

This part of the assignment is divided into two tasks:

1 - Create a bar plot showing the number of Tweets that were posted each hour of the event.
2 - For the most popular hour of the games, compute the top 10 hashtags that were emitted during that hour.

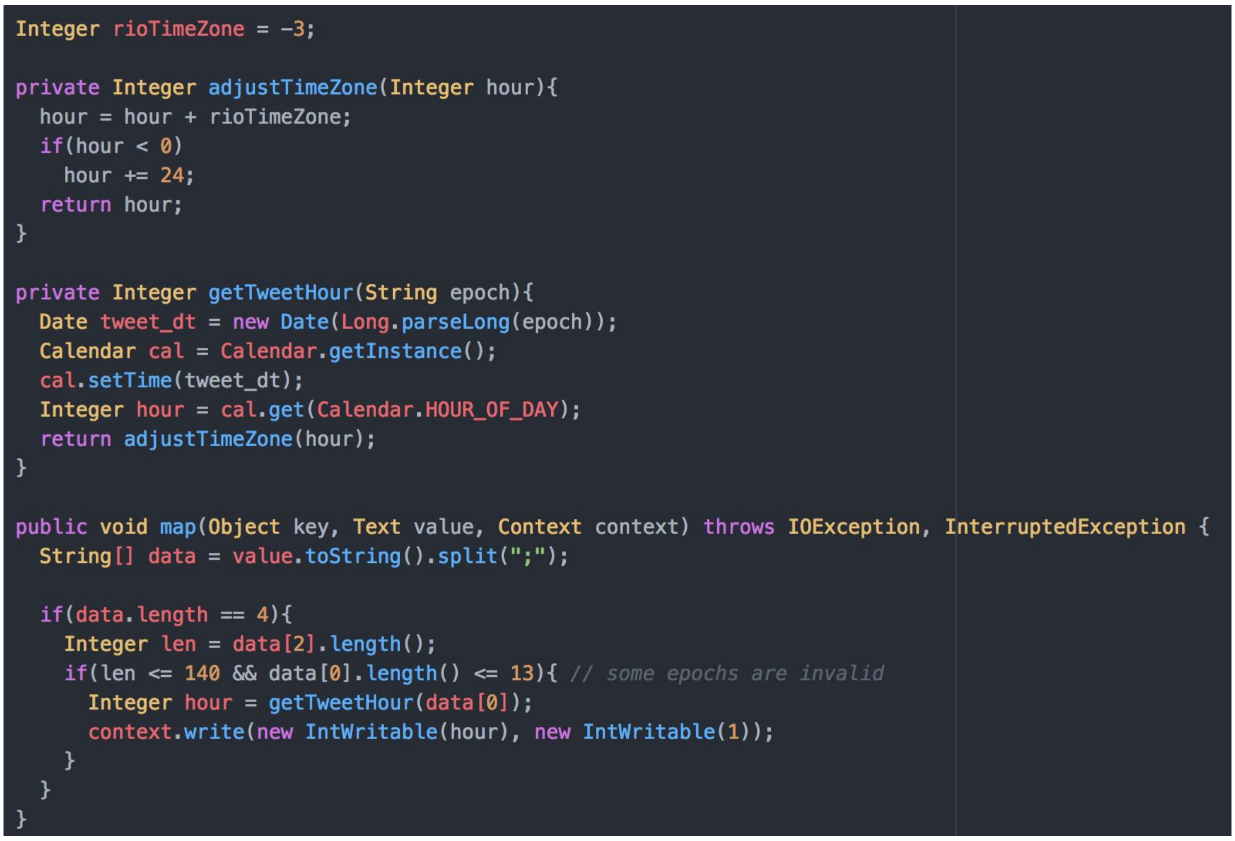
B.1 - Time

The MapReduce job `TimeAnalysis` used for Part 1 runs a reducer that is almost identical to `SimpleSumReducer.java`, the only difference being that the IntIntSumReducer.java produces an output where, like its value, the key is also a number, instead of a text.

The `TimeMapper` class contains two helper functions:

- `private Integer getTweetHour(String epoch)`
  - Parses the epoch time field to obtain the hour when the tweet was posted.
- `private Integer adjustTimeZone(Integer hour)`
  - Since the epoch time is in UTC, this function adjusts the obtained hour to match the local time in Rio de Janeiro. It is important to notice that while the offset from UTC in Rio is of two hours right now, the correct offset is, in fact, three hours, since the Southeast region of Brazil is currently on summer and has a Daylight Saving Time of +1. While normally the Olympic Games happen on Summer, this was not true for the games of 2016. The candid temperatures of the Brazilian winter allowed the games to happen normally in the middle of the year even though that is winter in the South Hemisphere. Although it is not necessarily true that all the tweets collected were posted in the timeframe of the games, for the purpose of this assignment, I am making this assumption.

Apart from using the two helper functions to obtain the hour, which is used as the output key for this Mapper, another difference between this and the previous mapper is in the filtering. While running the job, I encountered some invalid epochs, which was causing the job to fail. This was solved by discarding lines with an epoch with more than 13 digits as seen in the code extract below. Similar to the previous part of the assignment, the output value of the mapper is always 1, which is then summed by the reducer on the appropriate keys.



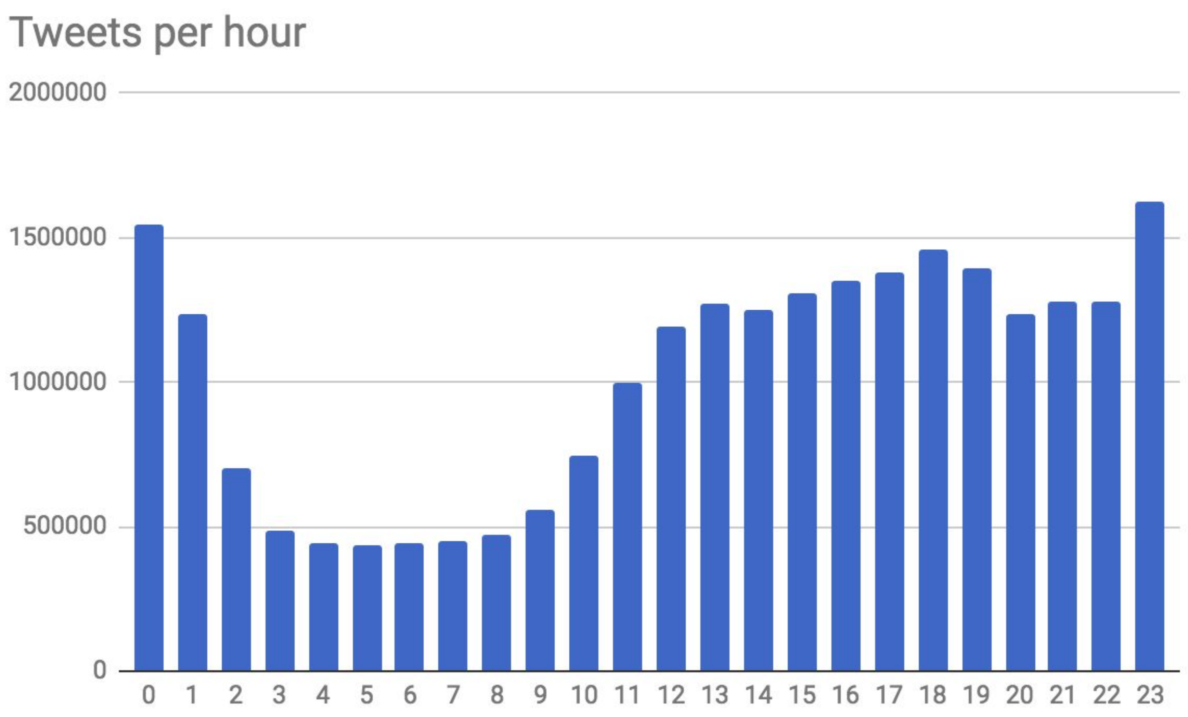
```java
Integer rioTimeZone = -3;

private Integer adjustTimeZone(Integer hour){
  hour = hour + rioTimeZone;
  if(hour < 0)
    hour += 24;
  return hour;
}

private Integer getTweetHour(String epoch){
  Date tweet_dt = new Date(Long.parseLong(epoch));
  Calendar cal = Calendar.getInstance();
  cal.setTime(tweet_dt);
  Integer hour = cal.get(Calendar.HOUR_OF_DAY);
  return adjustTimeZone(hour);
}

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
  String[] data = value.toString().split(";");

  if(data.length == 4){
    Integer len = data[2].length();
    if(len <= 140 && data[0].length() <= 13){ // some epochs are invalid
      Integer hour = getTweetHour(data[0]);
      context.write(new IntWritable(hour), new IntWritable(1));
    }
  }
}
```

Run instructions:

```
hadoop jar dist/EDCD3_BDP_Coursework1.jar TimeAnalysis /data/olympictweets2016rio out
hadoop fs -getmerge out time_analysis.txt
```

The unmodified output of the MapReduce job can be seen on the file `time_analysis.txt`, compressed with this report and the source code. To produce the chart below, the output was sorted using the unix command sort with a numerical flag then copied to a spreadsheet on Google Drive to produce the subsequent graph.

```
sort -n time_analysis.txt
```



According to my results, the busiest hour was between 23:00 and midnight. These results are interesting because many of the football games ran pretty late. Perhaps they were popular? Or perhaps people were tweeting while celebrating at bar or parties after the games with friends!

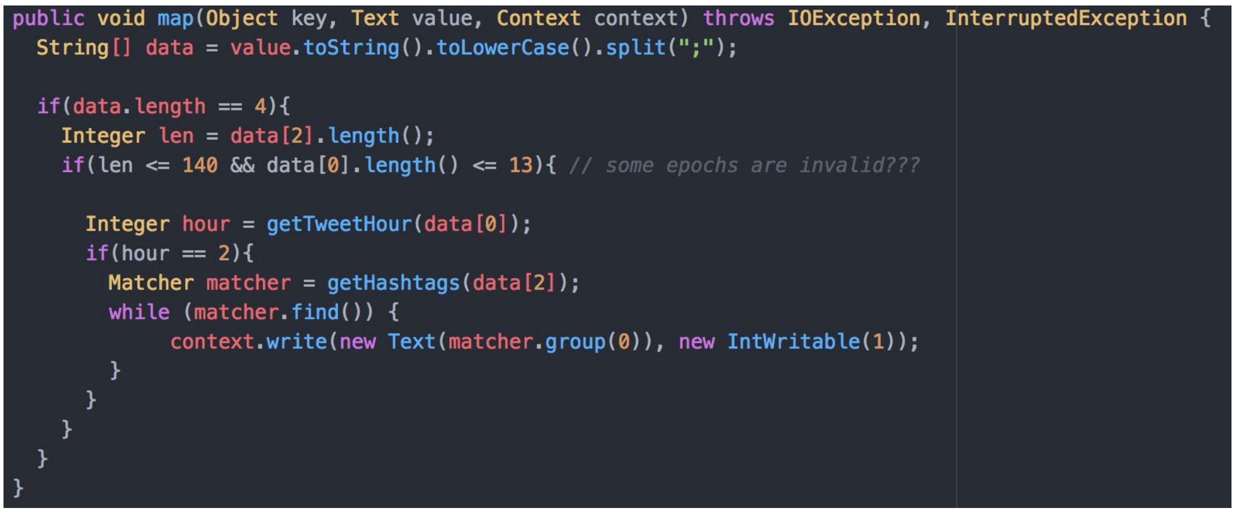
B.2 - Hashtags

The `HashTag` Analysis job runs the same reducer as Part A, since our key is a text containing the hashtag and the value is our final sum count.

The `HashtagMapper` contains two helper functions:

- `private Integer getTweetHour(String epoch)`
  - The same as getTweetHour from the previous part of the assignment
- `private Matcher getHashtags(String tweet)`
  - This function prepares a regular expression `"\\B(#\\w*[a-zA-Z]+\\w*)"` to match against the tweet message with the purpose of finding the present hashtag in it. After reading the most common definition for a twitter hashtag, for the purpose of this assignment, the regular expression above will compute as follows: Find substrings beginning with a blank space followed by a '#' then one or more alphanumeric characters as long as at least one of these characters is alphabetic (a hashtag can't be only numbers), then capture everything except the blank space. This function compiles the pattern and returns a Matcher, which is an iterator that will provide the captures.

For a more accurate lookup of the hashtags, the input line was converted to lowercase on the mapper. This was done because hashtags are case insensitive. The mapper also filtered the lines by hour, using the getTweetHour function and comparing it with the hardcoded value of `2`, which was picked based on the results of assignment B1, without the timezone adjustment (23:00 in Rio is 2:00 in UTC). After the line is filtered, the tweet message is matched against the regular expression with the `getHashtags` function. For each match, the hashtag captured by the RegEx is used as key, with 1 as value for the output of the Mapper as seen below:



```java
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
  String[] data = value.toString().toLowerCase().split(";");

  if(data.length == 4){
    Integer len = data[2].length();
    if(len <= 140 && data[0].length() <= 13){ // some epochs are invalid???

      Integer hour = getTweetHour(data[0]);
      if(hour == 2){
        Matcher matcher = getHashtags(data[2]);
        while (matcher.find()) {
             context.write(new Text(matcher.group(0)), new IntWritable(1));
        }
      }
    }
  }
}
```

Run instructions:

```
hadoop jar dist/EDCD3_BDP_Coursework1.jar HashtagAnalysis /data/olympictweets2016rio out
hadoop fs -getmerge out hashtag_analysis.txt
```

After running the MapReduce job and getting the result output, the following unix command was used to sort and truncate the data. The unmodified output of the MapReduce job can be seen on the file `hashtag_analysis.txt`, compressed with this report and the source code.

```
sort -t$'\t' -k2 -nr hashtag_analysis.txt | head -n10 | awk -F'\t' '{print $1}'
```

Sort is defining tabs as a column delimiter, choosing the second column as sorting key in a numerical reverse order. The output is then piped to the head command, which truncates it to 10 lines. The output is then piped again into the awk command that prints the column before the tab, giving us the following result with the ordered top 10 hashtags:

#rio2016

#olympics
#gold
#bra
#futebol
#usa
#oro
#cerimoniadeabertura
#swimming
#openingceremony

As expected, #rio2016 and #olympics are quite common hashtags, as for people celebrating gold medals. Brazil, United States and Football are other very frequent ones. This is also quite expected considering the olympics were happening in Brazil, a country of football lovers, where the brazilian masculine team and the american feminine team were in the dispute for favourites! Another interesting fact is that the opening ceremony appears twice in this ranking, once in Portuguese, once in English, and similar for the gold hashtag (English and Spanish). A future improvement for such analysis would be to associate words that mean the same thing in different languages, in order to obtain a more accurate description of the common topics.

Another possible future improvement could be to extract functions such as `getTweetHour` into a separate class, so it can be imported and reused by different mappers. In addition to that, instead of computing the top 10 using unix commands, this could also have been achieved by adding a combiner class. By using a Text and Integer pair, containing the hashtag and its count used as values with some unique or null key as output, this would allow for an additional reducer to access the whole range of hashtags and counts, so it could sort it and crop it. For the moment I chose not to do that, due to time constraints in delivering the coursework, especially since the external computation was allowed by the lecturer on the module forums.

Part C: Support Analysis.

This part of the assignment is divided into two tasks:

1 - Draw a table with the top 30 athletes in number of mentions across the dataset.
2 - Draw a table with the top 20 sports according to the mentions of olympic athletes captured.

C.1 - Athletes

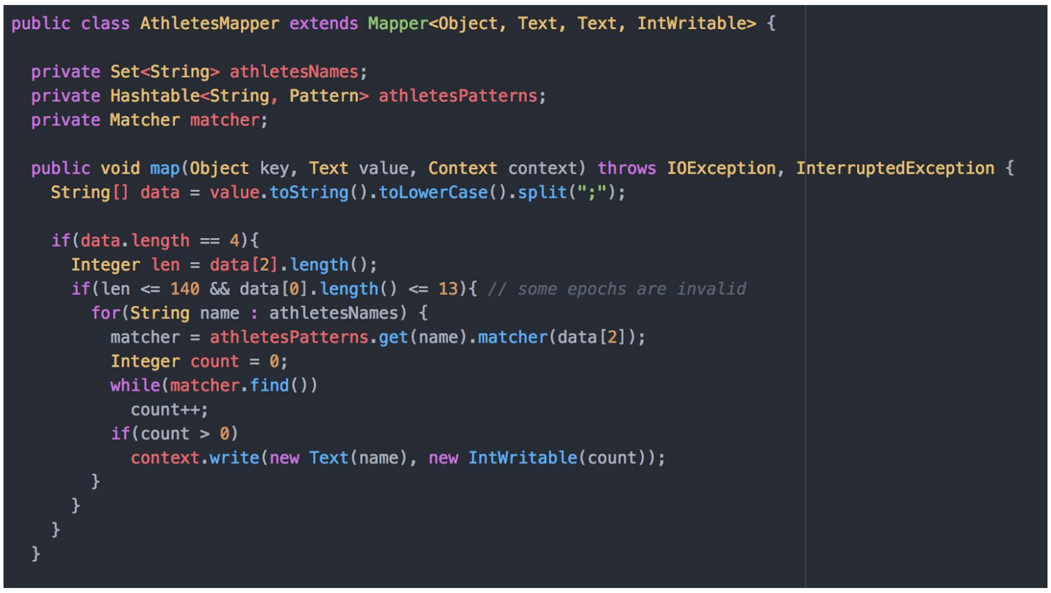
The `AthletesAnalysis` job runs the same reducer as Part A, since our key is a text containing the athlete name and the value is our final sum count. To be able to compute the athletes' names, a new, smaller, dataset `"/data/medalistsrio.csv"` was joined by being added into the cache.

Similarly to the exercise done in the lab 4 of this module, this dataset is accessed during the setup of the mapper. The setup creates a hashtable with the athletes as keys, and a regular expression pattern with the name as value. I chose to create the pattern here, instead of in the map function because this part of the assignment was a very computationally heavy. The reason is it is heavy is because it needs to match every single tweet to each of the thousands of athletes registered. The job was taking too long to compute and running approximately 45 containers. Eventually it was getting killed without being completed. Initially I used the smaller test dataset provided at QMULPlus, since I was worried about finishing the coursework. However, after doing some code cleanup and slight modifications, as the one I mentioned above, the job still ran slowly but it was able to finish in time without getting killed using the complete dataset.

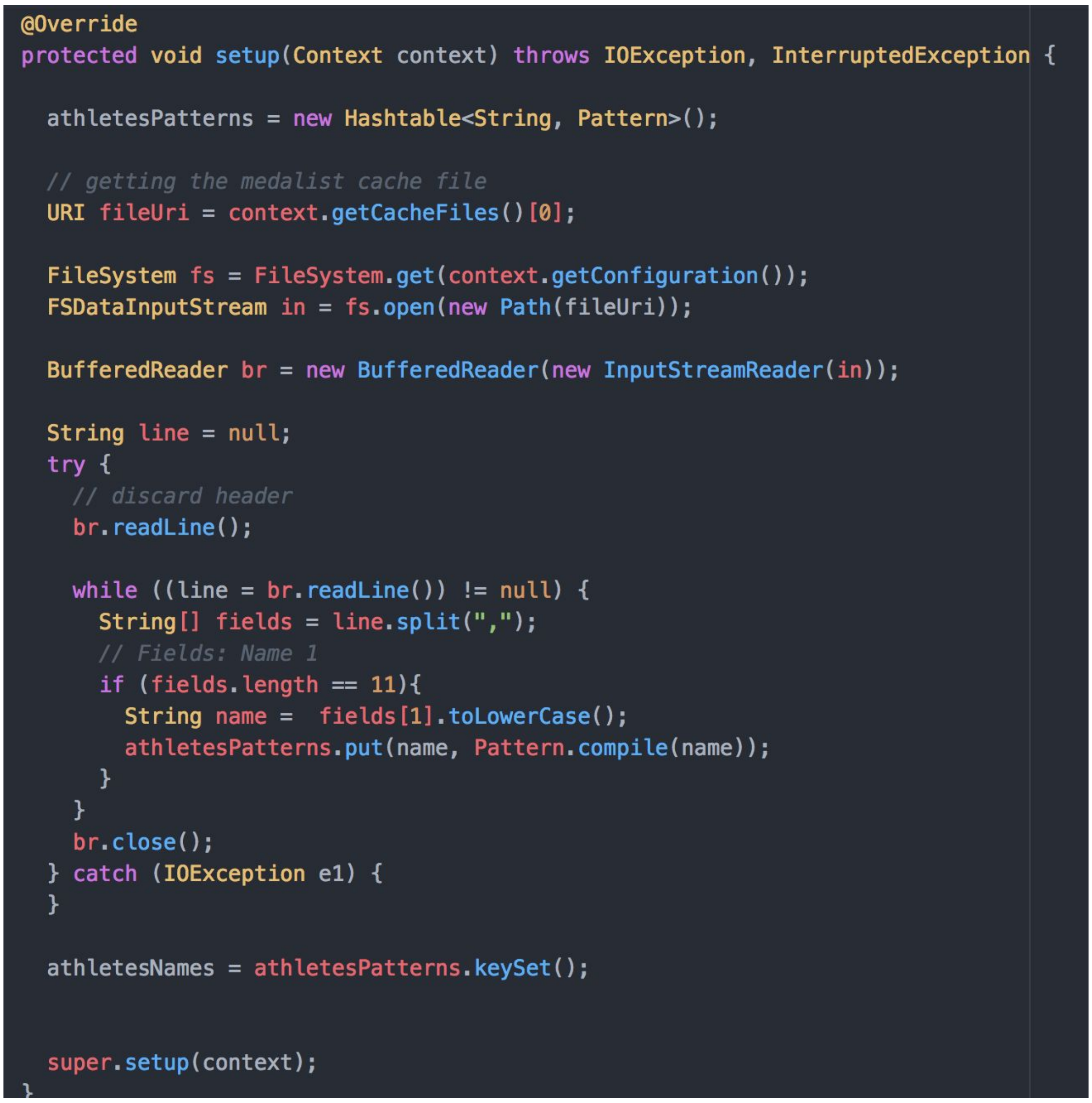
For the purpose of this assignment, I used every single mention of the athletes name as a symbol of support, even if it was mentioned more than once in the same tweet. Perhaps we could give some bonus points for enthusiasm? For this reason the mapper outputs the sum of all matches of the regular expression instead of just 1, which would also be a valid possibility. Perhaps it would even be the sanest alternative, but I decided to leave the job as is since it took long to compute.

Another thing I took into consideration was that in informal medias such as twitter, people often don't capitalise properly, so again, for the purpose of the assignment, to be sure that I was finding all occurrences of an athlete's name, the input from both the tweets dataset and the medalists dataset were converted to lowercase.

Code extracts from the map and the setup function can be seen below:



```java
public class AthletesMapper extends Mapper<Object, Text, Text, IntWritable> {

  private Set<String> athletesNames;
  private Hashtable<String, Pattern> athletesPatterns;
  private Matcher matcher;

  public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String[] data = value.toString().toLowerCase().split(";");

    if(data.length == 4){
      Integer len = data[2].length();
      if(len <= 140 && data[0].length() <= 13){ // some epochs are invalid
        for(String name : athletesNames) {
          matcher = athletesPatterns.get(name).matcher(data[2]);
          Integer count = 0;
          while(matcher.find())
            count++;
          if(count > 0)
            context.write(new Text(name), new IntWritable(count));
        }
      }
    }
  }
```



```java
@Override
protected void setup(Context context) throws IOException, InterruptedException {

  athletesPatterns = new Hashtable<String, Pattern>();

  // getting the medalist cache file
  URI fileUri = context.getCacheFiles()[0];

  FileSystem fs = FileSystem.get(context.getConfiguration());
  FSDataInputStream in = fs.open(new Path(fileUri));

  BufferedReader br = new BufferedReader(new InputStreamReader(in));

  String line = null;
  try {
    // discard header
    br.readLine();

    while ((line = br.readLine()) != null) {
      String[] fields = line.split(",");
      // Fields: Name 1
      if (fields.length == 11){
        String name =  fields[1].toLowerCase();
        athletesPatterns.put(name, Pattern.compile(name));
      }
    }
    br.close();
  } catch (IOException e1) {
  }

  athletesNames = athletesPatterns.keySet();


  super.setup(context);
}
```

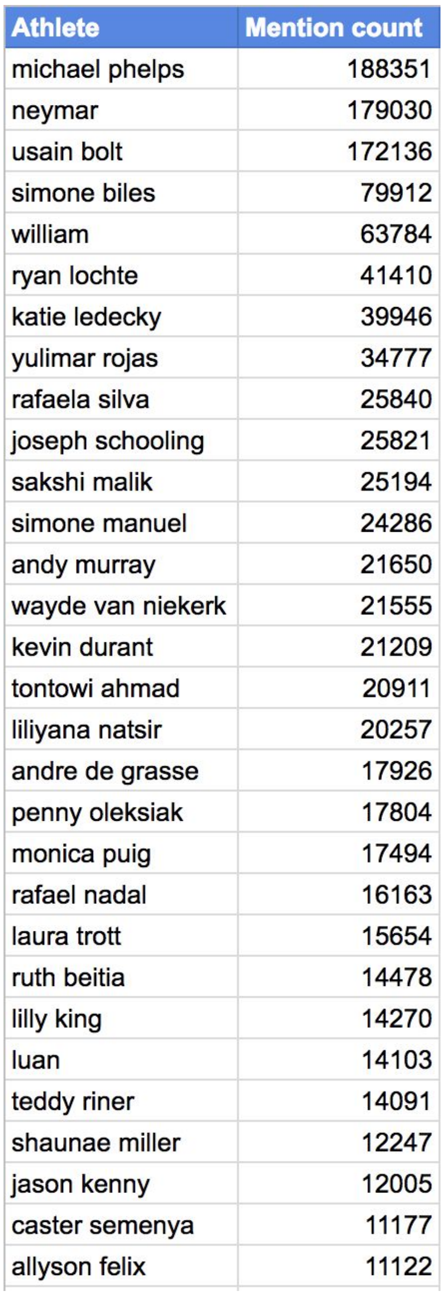
Run instructions:

```
hadoop jar dist/EDCD3_BDP_Coursework1.jar AthletesAnalysis /data/olympictweets2016rio out
hadoop fs -getmerge out athletes_mentions.txt
```

The unmodified output of the MapReduce job can be seen on the file `athletes_mentions.txt`, compressed with this report and the source code. The output of the reducers was sorted and truncated with the following unix command, using the same logic as in the previous part of the assignment:

```
sort -t$'\t' -k2 -nr athletes_mentions.txt | head -n30
```

The output was copied into a spreadsheet in Google Drive, producing the table below:



Athlete	Mention count
michael phelps	188351
neymar	179030
usain bolt	172136
simone biles	79912
william	63784
ryan lochte	41410
katie ledecky	39946
yulimar rojas	34777
rafaela silva	25840
joseph schooling	25821
sakshi malik	25194
simone manuel	24286
andy murray	21650
wayde van niekerk	21555
kevin durant	21209
tontowi ahmad	20911
liliyana natsir	20257
andre de grasse	17926
penny oleksiak	17804
monica puig	17494
rafael nadal	16163
laura trott	15654
ruth beitia	14478
lilly king	14270
luan	14103
teddy riner	14091
shaunae miller	12247
jason kenny	12005
caster semenya	11177
allyson felix	11122

C.2 - Sports

This part of the assignment is almost identical to the previous part. The main difference was that in the setup of the `SportMapper`, instead of adding the compiled pattern of the athletes' names as values in the hashtable for caching, the associated sport of the athlete was used, so it could be used as output key for the map function.

Although I was able to run this job with the whole dataset, there was a big increase in the time of computation in comparison with the previous part of the assignment, leading me to believe that the caching of the pattern into a hashtable was a significant performance boost. Refactoring this mapper to do the same would be an interesting improvement to consider for the future. I also wonder if it would be possible to reuse one same mapper with two different reducers to accomplish C.1 and C.2.

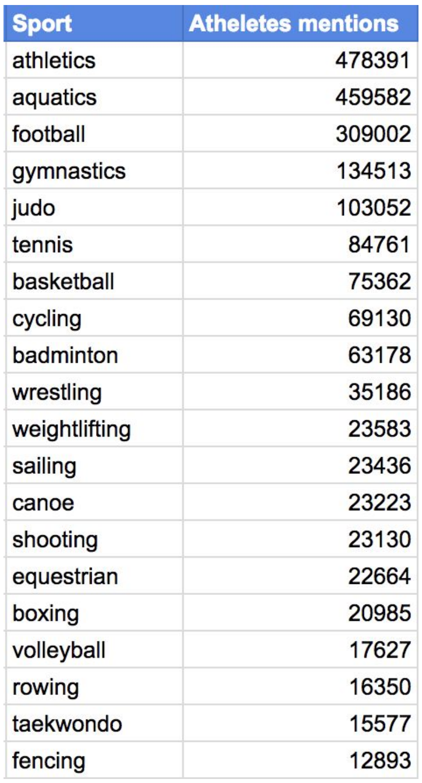
Run instructions:

```
hadoop jar dist/EDCD3_BDP_Coursework1.jar SportAnalysis /data/olympictweets2016rio out
hadoop fs -getmerge out sport_analysis.txt
```

The unmodified output of the MapReduce job can be seen on the file `sports_analysis.txt`, compressed with this report and the source code. This merged output was sorted and truncated with the following unix command, using the same logic as in the previous parts of the coursework:

```
sort -t$'\t' -k2 -nr sports_analysis.txt | head -n20
```

The output was copied into a spreadsheet in Google Drive, producing the table below:



Sport	Atheletes mentions
athletics	478391
aquatics	459582
football	309002
gymnastics	134513
judo	103052
tennis	84761
basketball	75362
cycling	69130
badminton	63178
wrestling	35186
weightlifting	23583
sailing	23436
canoe	23223
shooting	23130
equestrian	22664
boxing	20985
volleyball	17627
rowing	16350
taekwondo	15577
fencing	12893