ECS640U BIG DATA PROCESSING

Coursework 1 – Twitter analysis with Map Reduce

Efthymios Chatziathanasiadis
150359131

Table of Contents

PART A: Message Length Analysis

Mapping stage

Mapper Input: Each mapper method when called will have as input a line of text which corresponds to a Tweet.

Mapper Output: Each Mapper function when executed will emit as key and value the following:

- Key: The length bin(i.e. category) that the tweet belongs to as an **`IntWritable`** type
- Value: The value of 1 of type **`IntWritable`**

Each map method splits the tweet into an array were each array element contains a tweet component such as tweet id, date etc.
Erroneous tweets are filtered out by checking that tweet components are 4. In addition, tweets that are of length grater than 140 are also filtered out.
As we know the upper bound of tweet size is 140 and that the bin size is 5, we can say that there are 140/5 = 28 possible length groups of 5. The map method divides the tweet length by 5 to get the tweets group which is going to be emitted as a key.

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

private final IntWritable key = new IntWritable(1);
private final IntWritable val = new IntWritable(1);

public void map(Object ob, Text value, Context context) throws
IOException,                InterruptedException {
               String [] fields =value.toString().split(";");
               if(fields.length == 4){
                   if(fields[2].length() <= 140){
                       int length = fields[2].length();
                       float category = (float) length/5;
                       int cat = (int) Math.ceil(category);
                       key.set(cat);
                       context.write(key, val);
                   }
               }
    }
}
```

Figure 1.0: Mapper for part A

Reduce stage

Reducer Input:

- Key: The length bin(i.e. category) as an `IntWritable` type
- Value: A list of 1's of type `IntWritable` that are associated with the bin in the key.

Reducer Output:

- Key: The length bin(i.e. category) that the tweet belongs to as a `Text` converted in the required form (e.g. 1-5 , 6-10 etc.)
- Value: The aggregated frequency of the associated bin.

Each reducer will group by bin and compute the sum of the frequencies of each bin and emit the bin as a key and the aggregated frequency as a value. The input bin will be one of the 1-28 bins, as there are 28 possible bins. Hence, to output the bin in the correct format as stated in the problem definition the bin is multiplied by 5 to get its range (e.g. if the input bin is 28, 28*5=140 etc). This can be viewed in the implementation in figure 1.1.

```
public class partAreducer extends Reducer<IntWritable,
IntWritable, Text, IntWritable> {
  private Text key;
  private final IntWritable val = new IntWritable(1);

  public void reduce(IntWritable k, Iterable<IntWritable>
  values, Context context)
  throws IOException, InterruptedException {
        int sum=0;
        for(IntWritable i : values)sum=sum+i.get();
        val.set(sum);
        key = new Text( ((k.get()*5)-4)+" - "+(k.get()*5));
        context.write(key, val);
  }
}
```

Figure: 1.1: Reducer for Part A.

The output of the Map reduce job can be views in the histogram plot projected in figure 1.2.

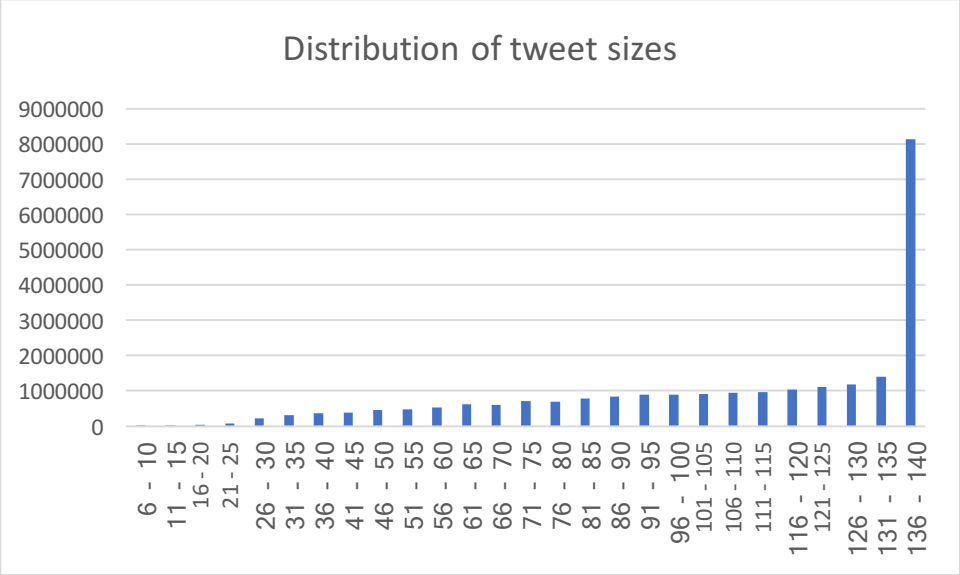

Figure 1.2: Output of Map Reduce in a histogram plot that depicts the distribution of tweet sizes among the Twitter data set.

PART B1: Time Analysis

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of `Text` which corresponds to a Tweet.

Mapper Output: Each Mapper function when executed will emit as key and value the following:

- Key: The hour of the tweet as an `IntWritable` type.
- Value: The value of 1 of type `IntWritable`

Each map method splits the tweet into an array were each array element contains a tweet component such as tweet epoch time etc.
Erroneous tweets are filtered out by checking that tweet components are 4. Further data cleaning is applied by catching format exceptions when parsing epoch time using a try-catch. Epoch time is located is the first component of a tweet. Hence, using the method `LocalDateTime.ofEpochSecond` tweet epoch time gets parsed to a `LocalDateTime` object. In addition, in order to convert epoch time to Rio's time zone, method `ofEpochSecond` allowed the supply of a zone argument (i.e. Rio's time zone is `ZoneOffset.of("-02:00")`). Hence, by now having a `LocalDateTime` object, hour in Rio time zone was easily extracted using `getHour` method. Finally the hour is emitted as a key which can be 0-23(i.e. 24 groups) and value 1. The mapper for this job can be seen in Figure 2.0 below.

```
public class partB1mapper extends Mapper<Object, Text, IntWritable,
IntWritable> {
 private final IntWritable key = new IntWritable(1);
 private final IntWritable val = new IntWritable(1);

 public void map(Object ob, Text value, Context context) throws
IOException, InterruptedException {
     String [] fields = value.toString().split(";");
     if(fields.length == 4){
          try{
                LocalDateTime dateTime = LocalDateTime.ofEpochSecond
(Long.parseLong(fields[0])/1000,0, ZoneOffset.of("-02:00"));
                int hour = dateTime.getHour();
                key.set(hour);
                context.write(key, val);
          }catch(NumberFormatException e){}
      }
    }
}
```

Figure 2.0: Mapper for part B 1

Reduce stage

Reducer Input:

- Key: The hour as an `IntWritable` type
- Value: A list of 1's of type `IntWritable` that are associated with the hour in the key.

Reducer Output:

- Key: The hour as a `Text` type.
- Value: The aggregated frequency of tweets in the corresponding hour as an `IntWritable` type.

Each reducer will group by hour and compute the sum of the frequencies of each hour and emit the hour as a key and the aggregated frequency of all tweets in that hour as a value. The input hour will be one of the group-hour 0-23(i.e. 24 groups). This can be viewed in the implementation below in figure 2.1.

```
public class partB1reducer extends Reducer<IntWritable,
IntWritable, Text, IntWritable> {
  private Text key;
  private final IntWritable val = new IntWritable(1);

  public void reduce(IntWritable k, Iterable<IntWritable>
values, Context context)
    throws IOException, InterruptedException {
        int sum=0;
        for(IntWritable i : values)sum=sum+i.get();
        val.set(sum);
        key = new Text(k.get()+"");
        context.write(key, val);
    }
}
```

Figure: 2.1: Reducer for Part B1.

The output of the Map Reduce job can be viewed in the bar plot projected in figure 2.2 below.

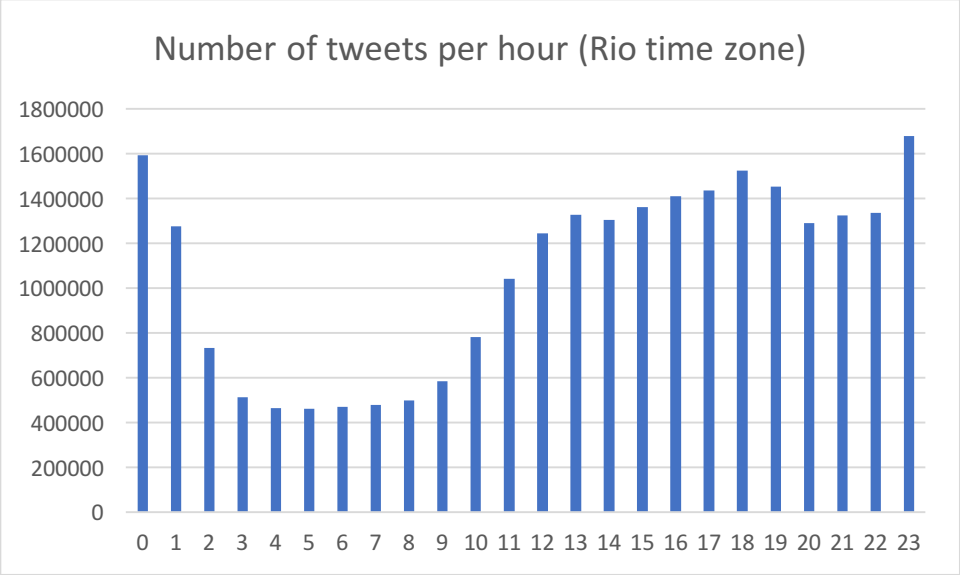

Figure: 2.2: Output of Map Reduce in a bar plot that depicts the distribution of tweet frequencies per hour among the Twitter data set. Clearly hour 23 is the most popular hour with the most number of tweets (to be used in part B2).

PART B2: Time Analysis

Problem definition: For the most popular hour of the games, compute the top 10 hashtags that were emitted during that hour.

The solution to this problem will require the following chained jobs:

1. The first job will be responsible for computing the frequencies of hashtags in the most popular hour (i.e. from part B1 most popular hour was found to be 23).
2. The second job will take the output of job 1 as input (i.e. containing hashtags and their associated frequencies during hour 23) and will compute top 10 hashtags using the top 10 Map Reduce programming pattern seen in lectures.

JOB 1

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of **`Text`** which corresponds to a Tweet.

Mapper Output: Each Mapper function when executed will emit as key and value the following:

- Key: The hashtag of a tweet as an **`Text`** type.
- Value: The value of 1 of type **`IntWritable`**

Each map method splits the tweet into an array were each array element contains a tweet component such as tweet epoch time, tweet content etc. Erroneous tweets are filtered out by checking that tweet components are 4. In addition, tweets that were posted in hours other than 23 are filtered out, as only the tweets in hour 23 must be considered. Hashtags are located in the 3rd component of a tweet. Tweets may include multiple hashtags hence a matching

function with appropriate pattern was defined in the implementation to extract them into an array list. After the hashtags have been obtained, the map method has a loop which goes through the array list and emits all the hashtags in the list as key and as value 1. This means that multiple **writes** happen in a map method.

The implementation of the mapper for this job can be seen in Figure 2.3 below.

```
public class partB21mapper extends Mapper<Object, Text, Text,
IntWritable> {
  private final Text key = new Text();
  private final IntWritable val = new IntWritable(1);
  public void map(Object ob, Text value, Context context) throws
IOException, InterruptedException {
     String [] fields = value.toString().split(";");
     final Pattern extractHashTags = Pattern.compile("#[a-zA-Z0-
9_]+");
     ArrayList<String> list = new ArrayList<String>();
     if(fields.length == 4){
        try{
            LocalDateTime dateTime =
LocalDateTime.ofEpochSecond(Long.parseLong(fields[0])/1000,0,
ZoneOffset.of("-02:00"));
            int hour = dateTime.getHour();
            if(hour == 23){
                Matcher m = extractHashTags.matcher(fields[2]);
                while(m.find()){
                    String tempHashTag = m.group(0);
                    list.add(tempHashTag);
                }
                for(String hashtag : list){
                    key.set(hashtag);
                    context.write(key, val);
                }
            }
        }catch(NumberFormatException e){}
    }
  }
}
```

Figure 2.3: Mapper for Job 1 in Part B2.

Reduce stage

Reducer Input:

- Key: The hashtag as an `Text` type
- Value: A list of 1's of type `IntWritable` that are associated with the hashtag in the key.

Reducer Output:

- Key: The hashtag as a `Text` type.
- Value: The aggregated frequency of the associated hashtag in the key as an `IntWritable` type.

Each reducer will group by hashtag and compute the sum of the frequencies of each hashtag and emit the hashtag as a key and the aggregated frequency of that hashtag in hour 23 as a value. The number of output keys of the reducer will equal the number of unique hashtags in hour 23. The implementation can be seen below in figure 2.1.

```
public class partB21reducer extends Reducer<Text,
IntWritable, Text, IntWritable> {
  private Text key;
  private final IntWritable val = new IntWritable(1);
  public void reduce(Text k, Iterable<IntWritable> values,
Context context)
      throws IOException, InterruptedException {
    int sum=0;
    for(IntWritable i : values)sum=sum+i.get();
    val.set(sum);
    key = k;
    context.write(key, val);
  }
}
```

Figure: 2.4: Reducer for Job 1 in Part B2.

The intermediate output of Job1 will be a list of hashtags and their frequencies. To set the input of job 1 to job 2 the following configuration was performed in the configuration class in the main method. A part of the configuration can be seen in figure 2.5 below.

```
public static void main(String[] args) throws Exception {
  runJob1(Arrays.copyOfRange(args, 0, args.length - 1),
args[args.length - 1]);
  String outputJob1 = args[args.length - 1];
  String inputJob2 [] = {outputJob1};
  runJob2(inputJob2, "Top10HashTags");
}
```

Figure: 2.5: Configuration for setting the output of job 1 as an input to job 2.

JOB 2 – The Top Ten Pattern

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of **`Text`** containing a hashtag and its frequency.

Mapper Output: Each Mapper function when executed will emit as key and value the following:

- Key: A **`NullWritable`** object
- Value: A **`TextIntPair`** customized writable object containing two instance variables. One **`Text`** variable storing the hashtag and one **`Intwritable`** storing the hashtag's frequency.

Default number of mappers is going to be used i.e. one per input split. **`Nullwritable`** is emitted as a key because we want all of the outputs of all the mappers to be grouped into a single key in the reducer.
The implementation of the mapper for job2 can be seen in Figure 2.6 below.

```
public class partB22mapper extends Mapper<Object, Text,
NullWritable, TextIntPair> {

  private  Text key = new Text();
  TextIntPair pair = new TextIntPair();

  public void map(Object ob, Text value, Context context)
throws IOException, InterruptedException {
    String fields [] =value.toString().split("\t");
    pair.set(fields[0], Integer.parseInt(fields[1]));
    context.write(NullWritable.get(), pair);
  }
}
```

Figure 2.6: Mapper for Job 2 in Part B2.

Reduce stage

Reducer Input:

- Key: `NullWritable`
- Value: A list of `TextIntPair` writable objects, each containing hashtag and its associated frequency.

Reducer Output:

- Key: `NullWritable`
- Value: Hashtag in `Text` type.

We specifically want one reducer because there will only be one key in all of the data (i.e. `NullWritable`). All of the data from the mappers will be collected together. To make sure only one reducer will run in the configuration class **`job.setNumReduceTasks(1);`** was set. The mapper will have a `TreeMap` to store the top 10, then finally output the top 10 all at once by calling 10 writes outputting the top 10 hashtags in descending order.

The implementation can be seen below in figure 2.7.

```
public class partB2reducer extends Reducer<NullWritable,
TextIntPair, NullWritable, Text> {
  private Text key ;
  public void reduce(NullWritable k, Iterable<TextIntPair> values,
Context context)
      throws IOException, InterruptedException {
      TreeMap<Integer, String> tweetFreqs =new TreeMap<Integer,
String>();
      for(TextIntPair i : values){
        String tag = i.getLeft();
        Integer freq = i.getRight();
        tag = tag + " " + freq;
        tweetFreqs.put(freq, tag);
        if(tweetFreqs.size() > 10)
tweetFreqs.remove(tweetFreqs.firstKey());
      }
      for(String st : tweetFreqs.descendingMap().values()){
        key = new Text(st);
        context.write(NullWritable.get(), key);
      }
  }
}
```

Figure: 2.7: Reducer for Job 2 in Part B2.

The final output with the top 10 hashtags of the 2 chained Map Reduce jobs can be viewed in figure 2.8 below.

Rank	Top 10 hash tags in hour 23	Mentions
1	#Rio2016	1361113
2	#Olympics	79574
3	#rio2016	74988
4	#Futebol	41568
5	#USA	40366
6	#Gold	39301
7	#BRA	37868
8	#OpeningCeremony	34570
9	#CerimoniaDeAbertura	33944
10	#TeamUSA	31198

Figure: 2.8: Output of Part B2 chained Job with the top 10 hashtags during the hour 23. From these results one could say that a USA football match was taking place as #USA and #football hashtags were one of the most popular ones during that hour.

PART C1: Support Analysis

Problem definition: compute the top 30 medalist athletes that received the highest support, according to the Twitter messages of our dataset.

The solution to this problem will require the following chained jobs:

1. The first job will be responsible for computing the mentions of each medalist in the original dataset. Part of the computation of this job will require a repartition join with the small medalist dataset provided using the **`setup`** method in the mapper.
2. The second job will take the output of job 1 as input (i.e. containing medalists and their associated mentions) and will compute top 30 hashtags using the top 10 Map Reduce programming pattern seen in lectures.

JOB 1

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of **`Text`** which corresponds to a Tweet.

Mapper Output: Each mapper when executed will emit as key and value the following:

- Key: The medalist's full name as a **`Text`** type.
- Value: The value of 1 of type **`IntWritable`**

The mapper class contains the setup method which assigns to a hashtable all the medalist names using the medalist data set provided. In order to access the medalist dataset appropriate configuration has been applied in the configuration class using the **`job.addCacheFile`** method. Having the hashtable initialized with all the medalist full names as key to the hashtable the mapper checks for full names in each tweet in constant time exploiting the big-oh(1) of hashtables using the **`containsKey()`** method. Full names are found using a simple algorithm which concatenates each word in the tweet with its previous as full names are two words back to back. It also covers full names that contain middle name where each word in a tweet is concatenated with its previous and next. Finally it handles cases were full names are just a single name, for example Neymar which has no surnames or middle names. Then each concatenation of each word is checked in constant time in the hashtable using **`containsKey`** method. Finally if **`containsKey`** returns true the mapper emits the full name as key and one as value.

The algorithm is composed of 3 methods for checking the following:

1) Single names

```
private void singleName(Context context, String tweet)throws
IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(tweet.toString(),
"-\t\n\r\f,.:;?![]'\"");
    while(itr.hasMoreTokens()){
      String current = itr.nextToken();
      if(athleteSports.containsKey(current)){
        key = new Text(current);
        context.write(key, val);
      }
    }
  }
```

Figure 3.0: Method implementation for searching single names e.g. Neymar.

2) Full names that are composed of a name and surname.

```
private void fullName(Context context, String tweet)throws
IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(tweet.toString(), "--
\t\n\r\f,.:;?![]'\"");
    String previous = null;
    while(itr.hasMoreTokens()){
      String current = itr.nextToken();
      if(previous != null){
        String fullName = previous+" "+current;
        if(athleteSports.containsKey(fullName)){
          key = new Text(fullName);
          context.write(key, val);
        }
      }
      previous = current;
    }
  }
```

Figure 3.1: Method implementation for searching full names e.g. Michael Phelps

3) Full names that are contain name surname and middle name.

```
private void fullName_withMiddle( Context context, String
tweet)throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(tweet.toString(),
"-- \t\n\r\f,.:;?![]'\"");
      String first = null;
      String middle = null;
      if(itr.hasMoreTokens()){
        first = itr.nextToken();
        if(itr.hasMoreTokens()){
          middle = itr.nextToken();
        }
      }
      if(first != null && middle != null){
        while(itr.hasMoreTokens()){
          String last = itr.nextToken();
          String fullName_middle = first+" " +middle+" "+last;
          if(athleteSports.containsKey(fullName_middle)){
            key = new Text(fullName_middle);
            context.write(key, val);
          }
          first = middle;
          middle = last;
        }
       }

  }
```

Figure 3.2: Method implementation for searching full names with midlle name e.g. Wayde van Niekerk

The implementation of the mapper for this job can be seen in Figure 3.3 below which calls the above methods for matching full names from the tweets to the medalist data set using the `containsKey()` method of Hashtable.

```
public class partC11mapper extends Mapper<Object, Text, Text,
IntWritable> {
  private  Text key ;
  private final IntWritable val = new IntWritable(1);
  private Hashtable<String, String> athleteSports;
  public void map(Object ob, Text value, Context context) throws
IOException, InterruptedException {
    String [] fields =value.toString().split(";");
    if(fields.length == 4){
      String tweet = fields[2];
      singleName(context, tweet);
      fullName(context, tweet);
      fullName_withMiddle(context, tweet);
    }
  }
```

Figure 3.3: Mapper for Job 1 in Part C1. In the mapper class there is also the setup method which intialises the hashtable with the medalists from the small dataset provided. The code for the setup method can be seen in the source files.

Reduce stage

Reducer Input:

- Key: Medalist full name as a `Text` type
- Value: A list of 1's of type `IntWritable` that are associated with the Medalist full name in the key.

Reducer Output:

- Key: The full name of the medalist as a `Text` type.
- Value: The aggregated sum of the mentions of the medalist in the key as an `IntWritable` type.

Each reducer will group by medalist full name and compute the sum of the frequencies of each full name and emit the full name as a key and the aggregated sum of mentions of that full name as a value. The number of output keys of the reducer will equal the number of unique medalist full names present in the original data set. The implementation can be seen below in figure 3.4.

```
public class partC11reducer extends Reducer<Text,
IntWritable, Text, IntWritable> {
  private Text key;
  private final IntWritable val = new IntWritable(1);
  public void reduce(Text k, Iterable<IntWritable> values,
Context context)
  throws IOException, InterruptedException {
    int sum=0;
    for(IntWritable i : values)sum=sum+i.get();
    val.set(sum);
    key = new Text(k);
    context.write(key, val);
  }
}
```

Figure: 3.4: Reducer for Job 1 in Part C1.

The intermediate output of Job1 will be a list of medalist full names and their mentions in the twitter data set. To set the input of job 1 to job 2 the following configuration was performed in the configuration class in the main method. A part of the configuration can be seen in figure 3.5 below.

```
public static void main(String[] args) throws Exception {
  runJob1(Arrays.copyOfRange(args, 0, args.length - 1),
args[args.length - 1]);
  String outputJob1 = args[args.length - 1];
  String inputJob2 [] = {outputJob1};
  runJob2(inputJob2, " Top30Medalists ");
}
```

Figure: 3.5: Configuration for setting the output of job 1 as an input to job 2.

JOB 2 – The Top Ten Pattern for computing Top Thirty Athletes

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of **`Text`** containing medalist full name and its mentions in the twitter data set.

Mapper Output: Each Mapper function when executed will emit as key and value the following:

- Key: A **`NullWritable`** object
- Value: A **`TextIntPair`** customized writable object containing two instance variables. One **`Text`** variable storing the full name and one **`Intwritable`** storing the full name's mention frequency.

Default number of mappers is going to be used i.e. one per input split. **`Nullwritable`** is emitted as a key because we want all of the outputs of all the mappers to be grouped into a single key in the reducer.
The implementation of the mapper for job2 can be seen in Figure 3.6 below.

```
public class partC12mapper extends Mapper<Object, Text,
NullWritable, TextIntPair> {

  private  Text key = new Text();
  private  IntWritable val = new IntWritable(1);

  public void map(Object ob, Text value, Context context)
throws IOException, InterruptedException {

   String fields [] =value.toString().split("\t");
   TextIntPair pair = new TextIntPair();
   pair.set(fields[0], Integer.parseInt(fields[1]));
   context.write(NullWritable.get(), pair);
  }
}
```

Figure 3.6: Mapper for Job 2 in Part C1.

Reduce stage

Reducer Input:

- Key: `NullWritable`
- Value: A list of `TextIntPair` writable objects, each containing medalist full name and its associated mentions in the twitter data set.

Reducer Output:

- Key: `NullWritable`
- Value: Medalist full name in `Text` type.

We specifically want one reducer because there will only be one key in all of the data (i.e. `NullWritable`). All of the data from the mappers will be collected together. To make sure only one reducer will run in the configuration class `job.setNumReduceTasks(1);` was set.

The mapper will have a `TreeMap` to store the top 30 medalists ordered by mention frequency, then finally output the top 30 all at once by calling 30 writes outputting the top 30 medalists in descending order, hence first output medalist will be the most mentioned medalist, second medalist will be the second most mentioned and so on.

The implementation of the reducer can be seen below in figure 3.7 below.

```
public class partC12reducer extends Reducer<NullWritable,
TextIntPair, NullWritable, Text> {
  private Text key ;
  public void reduce(NullWritable k, Iterable<TextIntPair> values,
Context context)
  throws IOException, InterruptedException {
    TreeMap<Integer, String> tweetFreqs =new TreeMap<Integer,
String>();
    for(TextIntPair i : values){
     String name = i.getLeft();
     Integer val = i.getRight();
     name = name+" "+val;
     tweetFreqs.put(val, name);
     if(tweetFreqs.size() > 30)
tweetFreqs.remove(tweetFreqs.firstKey());
    }
    for(String st : tweetFreqs.descendingMap().values()){
     key = new Text(st);
     context.write(NullWritable.get(), key);
    }
  }
}
```

Figure: 3.7: Reducer for Job 2 in Part C1.

The final output with the top 30 Medalists of the 2 chained Map Reduce jobs can be viewed in figure 3.8 below.

Rank	Medallist	Mentions
1	Michael Phelps	180258
2	Usain Bolt	167323
3	Neymar	86352
4	Simone Biles	78907
5	Ryan Lochte	40203
6	Katie Ledecky	37970
7	Yulimar Rojas	33608
8	Simone Manuel	26909
9	Joseph Schooling	26243
10	Sakshi Malik	24543
11	Rafaela Silva	22290
12	Andy Murray	21328
13	Kevin Durant	21222
14	Wayde van Niekerk	18103
15	Penny Oleksiak	17552
16	Monica Puig	16469
17	Laura Trott	15939
18	Rafael Nadal	15788
19	Teddy Riner	13970
20	Ruth Beitia	13638
21	Lilly King	13273
22	Jason Kenny	12018
23	Elaine Thompson	11990
24	Shaunae Miller	11966
25	Caster Semenya	11492
26	Almaz Ayana	11068
27	Hidilyn Diaz	11005
28	Allyson Felix	10582
29	Carmelo Anthony	10406
30	Adam Peaty	9814

Figure 3.8: *Output of Part C1 chained Job with the top 30 medalists and their associated mentions across the twitter data set.*

PART C2: Support Analysis

Problem definition: Compute top 20 sports according to the mentions of Olympic athletes captured.

The solution to this problem will require the following chained jobs:

1. The first job will be responsible for resolving each Medalist's full name to the corresponding sport using the medalist's data set containing the sport of each medalist. It will also compute the frequency of each sport. Part of the computation of this job will require a repartition join with the small medalist dataset provided using the `setup` method in the mapper.
2. The second job will take the output of job 1 as input (i.e. containing sports and their associated frequencies) and will compute top 20 sports among medalists using the top 10 Map Reduce programming pattern seen in lectures.

JOB 1

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of `Text` which contains an athlete medalist and his mentions across the data set. Essentially the input to the mapper is the intermediate output of Job 1 in Part C1 containing a list of all the mentioned medalists in the twitter data set.

Mapper Output: Each mapper when executed will emit as key and value the following:

- Key: The sport of each medalist as a `Text` type resolved using the hashtable i.e. repartition join
- Value: The frequency of each medalist of type `IntWritable`

The mapper class contains the setup method which assigns to a hashtable all the medalist names as key using the medalist data set provided. As value to the hashtable the corresponding sport of each medalist is assigned. In order to access the medalist dataset appropriate configuration has been applied in the configuration class using the `job.addCacheFile` method. Having the hashtable initialized with all the medalist full names, and sports the mapper checks each medalist using `containsKey()` in constant time exploiting the big-oh(1) efficiency of hashtables. Then, gets the sport of each medalist using the hashtable containing sports and writes the sport as key and value the mention frequency of each medalist obtained by the input to the mapper i.e. to be aggregated.

The implementation of the mapper for this job can be seen in Figure 3.9 below.

```
public class partC21mapper extends Mapper<Object, Text, Text,
IntWritable> {
  private  Text key ;
  private final IntWritable val = new IntWritable(1);
  private Hashtable<String, String> athleteSports;
  public void map(Object ob, Text value, Context context) throws
IOException, InterruptedException {
   String [] fields =value.toString().split("\t");
   if(fields.length == 2){
    if(athleteSports.containsKey(fields[0])){
     key = new Text(athleteSports.get(fields[0]));
     val.set(Integer.parseInt(fields[1]));
     context.write(key, val);
    }
   }
  }
```

Figure 3.9: Mapper for Job 1 in Part C2. In the mapper class there is also the setup method which initializes the hashtable with the medalists from the small dataset provided as a key and as a value to the hashtable the sport of the associated medalist is set. The code for the setup method can be seen in the source files.

Reduce stage

Reducer Input:

- Key: Sport as a **`Text`** type
- Value: A list of frequencies of type **`IntWritable`** that are associated with the sport in the key.

Reducer Output:

- Key: The sport as a **`Text`** type.
- Value: The aggregated sum of the sport frequencies as an **`IntWritable`** type.

Each reducer will group by sport and compute the sum of the frequencies of each sport and emit the sport as a key and the aggregated sum of frequencies of that sport as a value. The number of output keys of the reducer will equal the number of unique sports.

The implementation can be seen below in figure 4.0.

```
public class partC21reducer extends Reducer<Text,
IntWritable, Text, IntWritable> {
  private Text key;
  private final IntWritable val = new IntWritable(1);
  public void reduce(Text k, Iterable<IntWritable> values,
Context context)
  throws IOException, InterruptedException {
    int sum=0;
    for(IntWritable i : values)sum=sum+i.get();
     val.set(sum);
     key = new Text(k);
     context.write(key, val);
  }
}
```

Figure: 4.0: Reducer for Job 1 in Part C2.

The intermediate output of Job1 will be a list of sports their frequencies. To set the output of job 1 as input to job 2 the following configuration was performed in the configuration class in the main method. A part of the configuration can be seen in figure 4.1 below.

```
public static void main(String[] args) throws Exception {
  runJob1(Arrays.copyOfRange(args, 0, args.length - 1),
args[args.length - 1]);
  String outputJob1 = args[args.length - 1];
  String inputJob2 [] = {outputJob1};
  runJob2(inputJob2, "To20Sports");
}
```

Figure: 4.1: Configuration for setting the output of job 1 as an input to job 2.

JOB 2 – The Top Ten Pattern for computing Top Twenty sports

Mapping stage

Mapper Input: Each mapper method when called will have as input value a line of `Text` containing a sport and its frequency.

Mapper Output: Each Mapper function when executed will emit as key and value the following:

- Key: A `NullWritable` object
- Value: A `TextIntPair` customized writable object containing two instance variables. One `Text` variable storing the sport and one `Intwritable` storing the sport frequency.

Default number of mappers is going to be used i.e. one per input split. `Nullwritable` is emitted as a key because we want all of the outputs of all the mappers to be grouped into a single key in the reducer.
The implementation of the mapper for job2 can be seen in Figure 4.2 below.

```
public class partC22mapper extends Mapper<Object, Text,
NullWritable, TextIntPair> {
  private  Text key = new Text();
  private  IntWritable val = new IntWritable(1);
  public void map(Object ob, Text value, Context context)
throws IOException, InterruptedException {
   String fields [] = value.toString().split("\t");
   TextIntPair pair = new TextIntPair();
   pair.set(fields[0], Integer.parseInt(fields[1]));
   context.write(NullWritable.get(), pair);
  }
}
```

Figure 4.2: Mapper for Job 2 in Part C2.

Reduce stage

Reducer Input:

- Key: `NullWritable`
- Value: A list of `TextIntPair` writable objects, each containing sport and its associated frequency.

Reducer Output:

- Key: `NullWritable`
- Value: Sport with frequency in `Text` type.

We specifically want one reducer because there will only be one key in all of the data (i.e. `NullWritable`). All of the data from the mappers will be collected together. To make sure only one reducer will run in the configuration class `job.setNumReduceTasks(1);` was set.

The mapper will have a `TreeMap` to store the top 20 sports ordered by frequency, then finally output the top 20 all at once by calling 20 writes outputting the top 20 sports in descending order, hence first output sport will be the most popular among medalists, second sport will be the second most popular and so on.

The implementation of the reducer can be seen below in figure 4.3 below.

```
public class partC22reducer extends Reducer<NullWritable,
TextIntPair, NullWritable, Text> {
  private final IntWritable val = new IntWritable(1);
  private Text key ;
  public void reduce(NullWritable k, Iterable<TextIntPair> values,
Context context)
  throws IOException, InterruptedException {
  TreeMap<Integer, String> tweetFreqs =new TreeMap<Integer,
String>();
  for(TextIntPair i : values){
   String sport = i.getLeft();
   Integer val = i.getRight();
   sport = sport + " " + val;
   tweetFreqs.put(val, sport);
   if(tweetFreqs.size() > 20)
tweetFreqs.remove(tweetFreqs.firstKey());
  }
  for(String st : tweetFreqs.descendingMap().values()){
   key = new Text(st);
   context.write(NullWritable.get(), key);
  }
  }
```

Figure: 4.3: Reducer for Job 2 in Part C1.

The final output with the top 20 sports, obtained by the 2 chained Map Reduce jobs can be viewed in figure 4.4 below.

Rank	Sport	Frequency
1	aquatics	443833
2	athletics	442755
3	football	142873
4	gymnastics	127024
5	judo	95562
6	tennis	75622
7	basketball	72940
8	cycling	64848
9	wrestling	33800
10	sailing	23244
11	weightlifting	23147
12	shooting	22990
13	canoe	22955
14	equestrian	22950
15	boxing	22733
16	badminton	19779
17	volleyball	17827
18	taekwondo	15850
19	rowing	15344
20	fencing	12333

Figure: 4.4: Output of Part C2 chained Job with the top sports 20 along with their frequencies.