DS 256 - Scalable systems in Data Science
Assignment - 0

Sheshadri K. R.
Dream Lab (CDS)
06-02-02-10-12-18-1-16336

January 20, 2019

1 Experimental Setup

System Environment of Turing Node:

- **Java Version :** 1.8.0_112 (Oracle Corporation)
- **Spark Version :** 2.1.1.2.6.1.0-129
- **Scala Version :** 2.11.8
- **CPU:** Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- **OS architecture :** amd64
- **Number of Processing Cores :** 12
- **Total installed RAM :** 47GB

HDFS Environment:

- **Block size :** 128MB
- **Replication Factor :** 2

Software Setup

- **Java version**: 1.8.0.144
- **JSON library org.json** , version 20180813

Data description

- **Total Size:** 960GB
- **Total Files:** 3985
- **Average File size:** 245MB
- **Average Number of lines in each file:** 100000
- **Size of each JSON string (one line):** 3024 Bytes

2 Task 1 : FreqTag.java

The task expected us to compute, the ratio of hashtags to tweets, for all the users. Then, plot a frequency distribution of number of users versus hashtags per tweet.

As mentioned in the algorithm below, the input present in the $/user/ds256/twitter/$ is read into a RDD(line 1). All the deleted tweets are filtered out (line 2). Each line which is a JSON is parsed, a pair RDD is created where **UserName** is the key, **UserHashCount** object is the value (which contains hashtag and tweet counts) for the user (line 3). The user tweet and hashtag counts are grouped by *reduceByKey*()(line 4), and each user's hashTag to tweet ratio is computed (line 5) , classified it into buckets and grouped by *reduceByKey*() (line 6)and stored in HDFS.

Algorithm 1 FreqTag (InputFile, OutputFile)

```
1: JavaRDD<String> input ← sc.textFile(inputPath)
2: input ← input.filter(...)                                    ▷ Filter deleted tweets
3: JavaPairRDD<String,UserHashCount> userHashCount ← input.mapPartitionsToPair(...)
4: userHashCount ← userHashCount.reduceByKey(...)
5: JavaPairRDD<Integer,Long> bucketCount ← userHashCount.mapToPair(...)
6: bucketCount = bucketCount.reduceByKey(...)
```

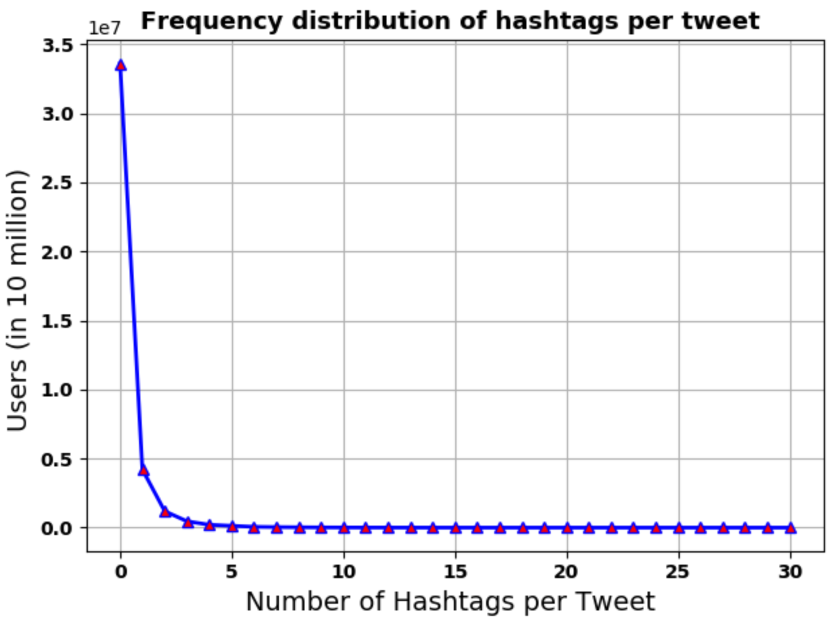

Figure 1: Frequency of Users vs Number of Hashtags per tweet

The users were classified into buckets as follows:

Bukcet 0, holds the count of all users whose hashTag To Tweet ratio is between 0 and 1, $(0, 1)$
Bucket 1, holds the count of all users whose hashTag To Tweet ratio is between 1 and 2, $(1, 2)$
and so on.

If we observe the Figure 1, X-axis is number of buckets, Y-axis is the number of users (in 10million) we can see that most of the users fall in the first 2 buckets.

The spark job was submitted with the following configuration:

- Number of executors = 5
- Number of cores = 4
- Memory = 8GB

- Number of Partitions of data = 9406

The algorithm ran into completion taking around 1.7 hours.

3 Task 2 : TopCoOccurrence.java

In task 2, we were expected to find the top 100 co-occurring pair of hashtags.In the algorithm shown below, The input data is read into RDD(line 1). Deleted tweets were removed(line 2), and for each JSON string, $flatMapToPair()$ was applied to create a pair RDD(line 3). In the pairRDD, the key is the **co-occuring hash tag** and the **value is initialized to 1**, since it can only appear once in a single tweet. Repeat pairs are ignored. (eg: #AusOpen #AusOpen was not considered as a co-occuring hashtag).

The pairRDD keys were grouped by $reduceByKey()$ (line 4), so that we have co-occuring hashtags and their counts. The hashtags and counts were swapped (line 5) and sorted (line 6) to get the top 100 co-occuring hastags. The results were stored in hdfs.

Algorithm 2 TopCoOccurrence (InputFile, OutputFile)

```
1: JavaRDD<String> input ← sc.$textFile(inputPath)$
2: input ← input.*filter(...)*                                    ▷ Filter deleted tweets
3: JavaPairRDD<String,Integer> pairHashTag ← input.$flatMapToPair(...)$
4: pairHashTag ← pairHashTag.$reduceByKey(...)$            ▷ Groups all the co-occuring hashtags
5: JavaPairRDD<Integer,String> pairHashTagCount ← pairHashTag.$mapToPair(...)$   ▷ swap the
   key-value pair
6: pairHashTagCount ← pairHashTagCount.$sortByKey(...)$   ▷ Sorts the counts of pair hashtags in
   decreasing order
```

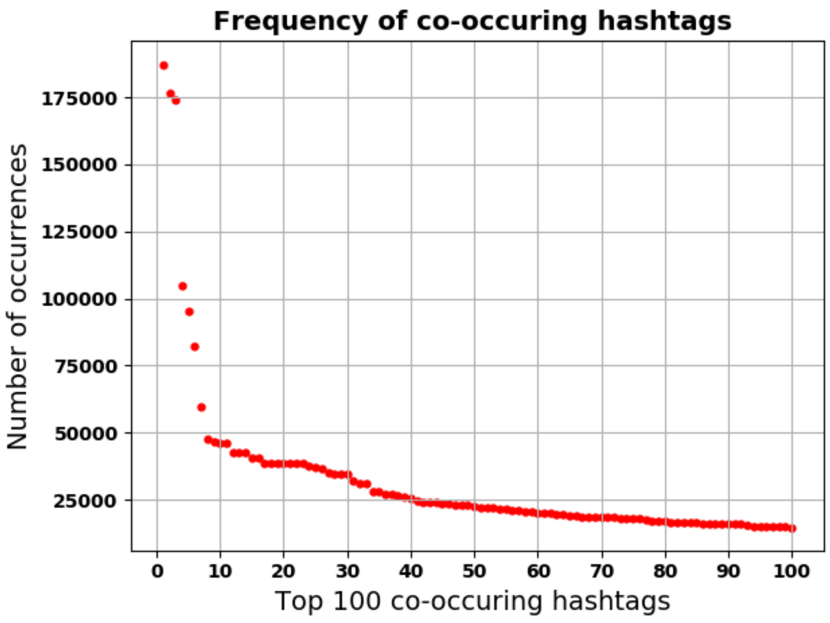

Figure 2: Frequency of the Top 100 co-occuring hashtags

In Figure 2 X-axis represents the top-100 co-occuring hashtags and Y-axis represents their corresponding frequencey , popular co-occuring hashtags show a high frequency for nearly top 5 values and show an steep drop when we approach the last 10 co-occuring hashtags.

The spark job was submitted with the following configuration:

- Number of executors = 5
- Number of cores = 4
- Memory = 8GB
- Number of Partitions of data = 9406

The algorithm ran into completion taking around 4.9 hours. The reason it took around 5 hours is because of 2 count() statements in the program, which alone around 3 hours.

4 Task 3 : InterGraph.java

In this task, we were expected to created a vertex file and an edge file. In vertex file, each line represented a vertex, which had a fixed part (user Id and created at) and a temporal part (timestamp of the tweet, follower count and friends count).

In Edge file, similarly, we had fixed part (tweeter's User ID, retweeter's User ID) and a temporal part (timestamp of the retweet, tweet ID, retweet ID, hashtags).

After processing the JSON we were expected to give the count of vertex file and edge file.

In the algorithm shown below, The input data is read into RDD(line 1). Deleted tweets were removed(line 2), *mapPartitionsToPair* is applied to each JSON string from the file, to created a PairRDD (line 3). In it, key is **(userId, createdAt)** and the value is **(timestamp, followers, friends)**. The pairRDD is applied a *reduceByKey*() to group all the values per key. (line 4)

A similar operation is applied to edge too, which can be seen in lines 5 and 6. The results in RDD are saved into HDFS.

Algorithm 3 InterGraph (InputFile, VertexFile, EdgeFile)

```
1: JavaRDD<String> input ← sc.textFile(inputPath)
2: input ← input.filter(...)                                        ▷ Filter deleted tweets
3: JavaPairRDD<String,String> vertexRDD ← input.mapPartitionsToPair(...)    ▷ key is the
   constant part userId, createdAt, for the vertex file
4: vertexRDD ← vertexRDD.reduceByKey(...)
5: JavaPairRDD<String,String> edgeRDD ← input.mapPartitionsToPair(...)      ▷ key is the
   constant part which is sourceId, sinkId, for the edge file)
6: edgeRDD ← edgeRDD.reduceByKey(...)
```

The spark job was submitted with the following configuration:

- Number of executors = 5
- Number of cores = 4
- Memory = 8GB

- Number of Partitions of data = 9406

The algorithm ran into completion taking around 3.8 hours. The reason it took around 4 hours is because of the *mapToParitions*() was applied twice , separately once for edge and once for vertex.

The total number of vertices are **216379932**. The total number of edges are **81555593**.

5 Task 4 : Scaling the inputs

In this task, we were expected to run the inputs for different sizes 1%, 5%,10%, 25%, and 100% for any of the above task and report the time taken for execution for the same. I have chosen and ran **FreqTag.java**.

Initially it was ran for 1% using the $sample(withReplacement, fraction)$ api in the RDD, but it would go through the entire 1TB dataset to give a RDD of the specified fraction.

Hence, I examined the size of files in the $/user/ds256/twitter/$ and chose enough files whose size together made upto 1%(10GB), 5%(55GB), 10%(106GB), 25%(260GB).

The spark job was submitted with the following configuration:

- Number of executors = 5
- Number of cores = 4
- Memory = 8GB
- Number of Partitions of data = 83, 448, 878, 2168 respectively for 1%(10GB), 5%(55GB), 10%(106GB), 25%(260GB).

The running time is as shown in the Figure 3 below(next page).

Expectation: The algorithms should take less time (i.e run faster) with increase in number of cores.

Observation: It behaves as expected, it can be seen that the algorithm scales almost linearly with the increase in input, with the above mentioned configuration. It was an expected behavior.

6 Additional Experiments

6.1 Varying the number of executors

I ran the experiment varying the number of executors, for the 1% data size (10GB).

The spark job was submitted with the following configuration:

- Number of executors = 2, 3, 4, 5
- Number of cores = 4
- Memory = 8GB
- Number of Paritions of data = 83

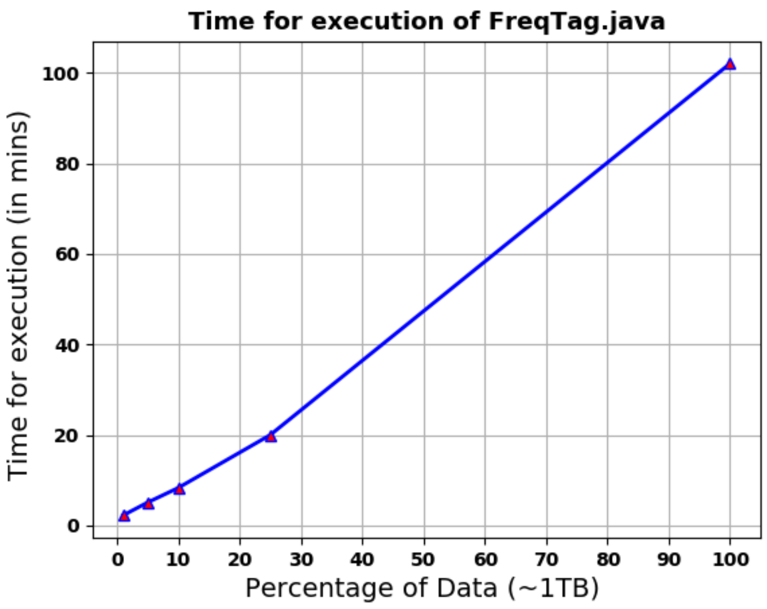

Figure 3: Running time of various data sets

Expectation: As we increase the number of executors, the jobs should start running faster. **Observation:** The results of experiments run as expected. If we see the Figure 4. The job submitted with 5 executors take the least time and 2 executor takes the highest amount of time.

6.2 Varying the number of cores

I ran the experiment varying the number of cores, for the 1% data size (10GB).

The spark job was submitted with the following configuration:

- Number of executors = 5
- Number of cores = 1,2,3
- Memory = 8GB
- Number of Partitions of data = 83

Expectation: As we decrease the number of executors, the job should take more time.

Observation: The results of experiments didn't meet the hypothesis, if we see Figure 5, the 1 core ran faster than 3 cores.

The 1 executor cores took 5s in GC, 2 executor cores took 7s in GC, 3 executor cores took 14s. The saveToTextFile completed in 2s for 1 executor core, 11s in 2 executor cores, 1.5mins in 3 executor cores.

The read time for 10GB in 1 executor took 2.3 mins, 3 executor cores took 4.1 mins. All this contributed for 1 core to execute faster than 3 cores.

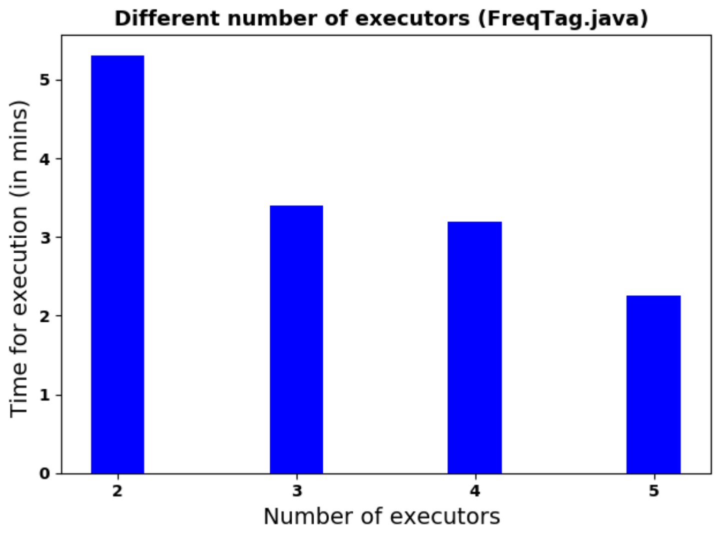

Figure 4: Varying the Number of Executors

The write time for 1 core took 2s, and 1.5 mins for 3 cores. This was at the time when there was high cluster utilization, where it witnessed a low read and write throughput.

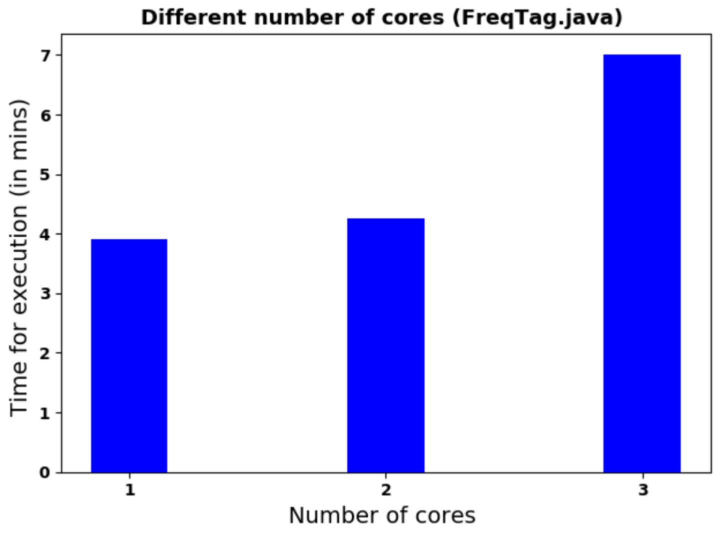

Figure 5: Varying the number of cores

7 Acknowledgements

I would like to thank Swapnil for quick responses and the spark primer to help kick-start the assignment.