ET4310: SuperComputing For Big Data
Lab Assignment 2

Georgios Dimitropoulos: 4727657
Emmanouil Manousogiannis: 4727517
Group 29

September 2018

Introduction

The objective of this assignment is to process the entire GDelt 2 dataset (i.e. all segments) with 20 c4.8xlarge core nodes, in under half an hour, using our solution from lab assignment 1. In order to be able to do so, it is crucial as an initial step to evaluate whether our application scales well before running on the entire dataset. For this reason we firstly performed some trials in subsets of the entire dataset (e.g. 3000 segments, 30000 segments), and also using smaller and cheaper clusters, to verify if our application scales in a proper way. Based on this, we were able to identify some potential bottlenecks and modify our implementation accordingly in order to achieve the necessary objective. Performing all the configurations (which are analyzed in the following sections of the report) we managed to process the entire dataset in **12 minutes using a 20 node c4.8xlarge cluster** and in **20 minutes using a 10 node c4.8xlarge cluster**, which we believe it was a very promising outcome.

Optimization

After identifying the bottlenecks of our application, in order to achieve the aforementioned performance in the entire dataset, we proceeded in the necessary configurations to optimize our applications efficiency. These configurations can be split into four main categories. Namely, Code optimization, Increasing the level of Parallelism, Cluster Resource Allocation and Data Serialization. Each of these configurations are analyzed below.

1) Code Modifications

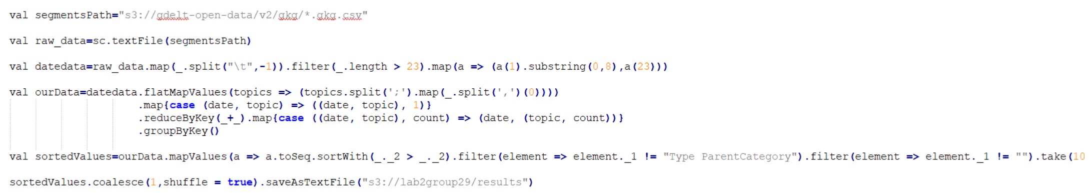
The first step towards improving our application was to modify the code that we used in Lab assignment 1 in order to run on the entire dataset on the AWS cloud. Firstly, we realized that the initial code that we used in the Lab assignment 1 was a poor implementation in terms of efficiency.Apart from finding the top 10 topics for each date dynamically, instead of hard coding the date, which was quite a naive approach and could cot be applied for a dataset of so many dates, we also performed the following changes:

Modification 1: When Using the "*take() method*" after the string filtering, then all of this data is moving from the executors of our cluster to the driver and hence cause a lot of network load.This may also lead o a crush if there is not enough amount of memory available. Therefore, such an implementation does not correspond to a correct Apache Spark implementation, based on the fact that all the data aggregations and reductions should be performed within the Spark API and then only use *take()* to take a subset of the resulted data for debugging.Based on these observations, we only used the "take method" in the very end just to take the 10 most common discussion topics of a date.

Modification 2:As a second change in our Scala code, wee tried to **reduce the number of the data that is shuffled**.As mentioned in [2], shuffles are quite expensive operations, the data to be shuffled must be written to disk and transferred over the network. However, all those operations are not equal so we tried to minimize either the amount of shuffle operations or the amount of data that is to be shuffled.
For this reason, we replaced our $rdd.groupByKey().mapValues(_.sum)$ with $rdd.reduceByKey(_+_)$. Those two operations produce the same outcome however instead of transferring the entire data across the network which is quite slow, we will compute local sums for each key in each partition and then combine those local sums into one after shuffling.

Modification 3: In the final stage of our implementation, we write the results of output to a **textfile**. To do so, we use the command "$rdd.coalesce(1)$ ". This command is equal to repartitioning our data to 1 partition.Otherwise, our relusted rdd would be stored in as many files as the number of partitions. However, when actually executing our code, we monitored our application through YARN application manager and found out that this actual task took an enormous amount of time to execute.This happened because we are creating one partition and hence only one node from our cluster is working. There is no actual parallelism in our computation. To minimize this effect we set "$coalesce(1, shuffle = True)$" ,which allowed us to achieve a very significant boost in the time performance of the application.

Below we have attached our main code implementation.



```
val segmentsPath="s3://gdelt-open-data/v2/gkg/*.gkg.csv"

val raw_data=sc.textFile(segmentsPath)

val datedata=raw_data.map(_.split("\t",-1)).filter(_.length > 23).map(a => (a(1).substring(0,8),a(23)))

val ourData=datedata.flatMapValues(topics => (topics.split(';').map(_.split(',')(0))))
                    .map{case (date, topic) => ((date, topic), 1)}
                    .reduceByKey(_+_).map{case ((date, topic), count) => (date, (topic, count))}
                    .groupByKey()

val sortedValues=ourData.mapValues(a => a.toSeq.sortWith(_._2 > _._2).filter(element => element._1 != "Type ParentCategory").filter(element => element._1 != "").take(10

sortedValues.coalesce(1,shuffle = true).saveAsTextFile("s3://lab2group29/results")
```

Figure 1: Scala implementation for processing GDelt dataset

2) Increase the level of parallelism

Even after identifying the basic bottlenecks of our implementation, we only managed to achieve a satisfying scaling performance when the number of segments increases. However, when we tried to process around 30,000 segments within our cluster (1 c4.8xlarge master node and 20 c4.8xlarge nodes), we only managed to achieve this in **10 minutes**. This means that if we assume a relatively linear scaling, we would need around 40 minutes for the whole dataset.

One of the most important aspects of further speeding up the process of the entire dataset , is related to **increasing the level of parallelism of our application.** We would not be able to take advantage of the full cluster resources, unless the level of parallelism for each operation is high enough.

When an RDD is created by reading a file within a Spark application, it is automatically 'split' in a number of partitions. After that, any other RDD that is created from that parent RDD will have the same number of partitions as its 'parent',with a few exceptions like 'coalesce' where *repartionioting* is taking place. This number of partitions, has to be large enough so that parallel processing property of Spark can be actually used. If we have fewer number of partitions than the available CPU cores in the cluster, then obviously some cores will remain idle and the rest will be assigned with more heavy tasks. On the other hand, if we set the number of partitions 2-3 times the number of the available CPU cores in our cluster [5], then all nodes will work in parallel and the procedure will speed up.

In our case, our cluster consists of $21 \times 36 = 756$ cores, so we set the default number of partitions to $2 \times 756 = 1512$. This was set by modifying the **spark.default.parallelism** property in spark-submit. After applying this change the performance difference was surprising. We managed to **process 30,000 segments in 3.4 minutes**.We tried to further increase the level of parallelism but there was not further increase. An overview of the jobs before applying parallelism and after that is attached below from Ganglia.

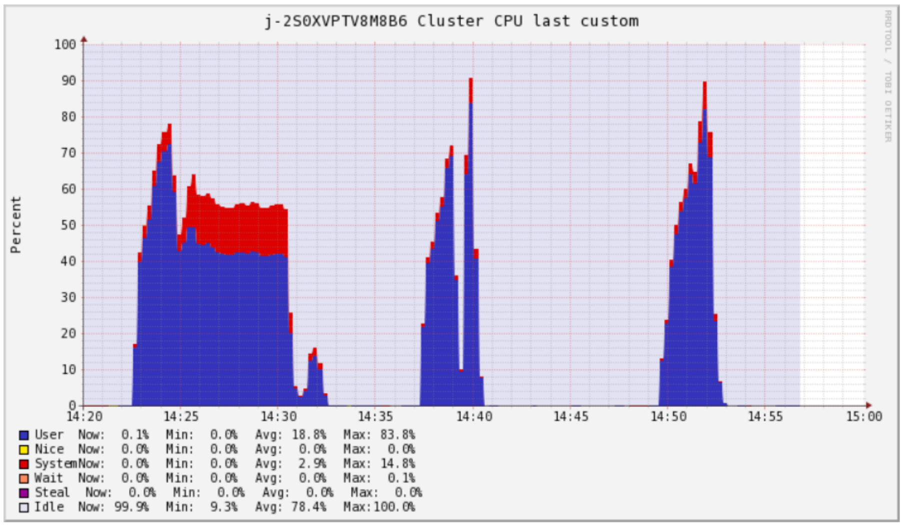


Figure 2: Execution with default parallelism settings (left) and with 2 and 3 X CPUcores parallelism (right)

Default Partition number	**Execution time**
Default	10 minutes
2 X CPU cores	3.4 minutes
3 X CPU cores	3.4 minutes

Table 1: Average Execution Time with different levels of parallelism

3) Cluster Resource Allocation

A.

The next thing we had to do, was to further optimize the way in which our application is using the cluster resources. Without reasonable resource allocation, a spark job can consume entire cluster resources and make other applications starve for resources.

In this section we mainly focus on how to configure the number of executors, memory settings of each executors and the number of cores in our Spark Job. To begin with, let's have a look of spark's architecture.

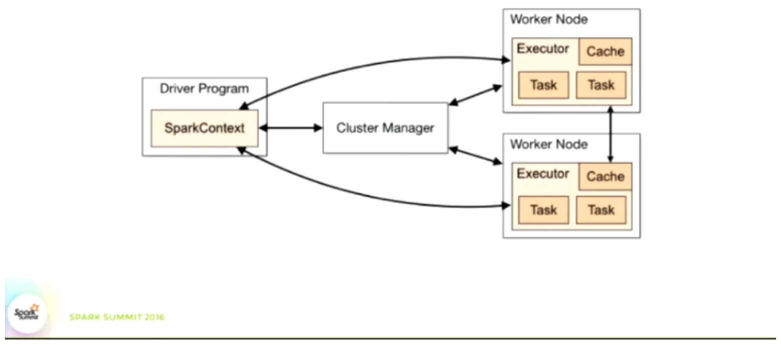


Figure 3: Architecture of Spark

Every Spark application corresponds to an instance of the SparkContext class. An application can correspond to one or multiple jobs to be performed and has some processes, called Executors, running on the cluster on its behalf continuously. In order to determine the optimal number of executors, as well as the resources that they will be assigned with (number of cores and memory), we were based on [6]. After defining the optimal resource allocation in our cluster, we aim for further performance improvement.

Our cluster consists of:
-21 nodes (c4.8xlarge)
-36 cores per node
-60 GiB memory per node

Step 1: We will assign **5 cores per executor**. This is the maximum value we can set as HDFS client is reported to have problems if this number is higher.

Step 2: Our cluster will have a total of :$21 * (36 - 1) = 735$ available cores. We subtract 1 core per node for several daemons that'll run in the background like NameNode, Secondary NameNode, DataNode, JobTracker and TaskTracker.

Now we need to determine the amount of memory assigned to each executor. We have to mention here that we have to consider at least 1024MB per node and 1 executor (in total) for the Application Master in order to be able to negotiate resources and perform other kinds of tasks. So instead of 60 GiB per node, we will consider 59 GiB per node. In addition, we have to mention the concept of memory overhead. When we request a specific amount of memory from YARN, it will always request an additional small overhead from the Resource manager. This is usually the $max(384mB, 0.7 * requestedMemory)$. The image below is indicative.

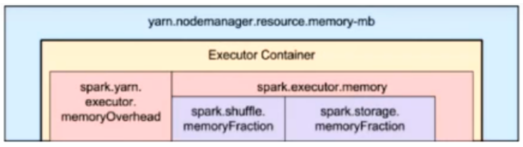

yarn.nodemanager.resource.memory-mb
Executor Container
spark.yarn.executor.memoryOverhead
spark.executor.memory
spark.shuffle.memoryFraction
spark.storage.memoryFraction

Figure 4: Memory requested to Resource Manager

Having the above in mind we can now calculate the optimal amount of memory per executor.

Step 3: Based on step 1 and 2, the total number of executors will be: $735/5 = 147$ **executors**.This means that each node will have $147/21 = 7$ executors per node. So $59/7 = 8$GB and $8 \times (1 - 0.07) = 7.83$GiB.

So finally we will have: **734 executors, with 5 cores each and 7 GB of memory each**. This will be configured through the spark-submit arguments when we start our Spark project.

After applying those updates our results became even better, as we were able to process **30,000 segments in 3.1 minutes**.

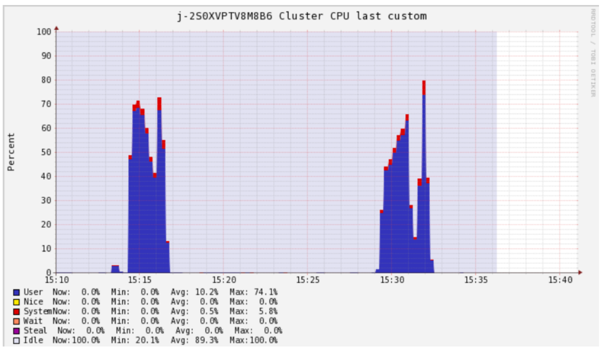


Figure 5: Left:Optimal resource allocation. Right: Initial resource allocation

Since we have managed to achieve such a good execution time for almost one third of the dataset, we can safely assume that our application will scale sufficiently enough for the whole dataset. Indeed we managed to process all the **GDelt dataset in 14 minutes**. This already satisfies the objective of this assignment.

B.

We have already achieved a nice performance , however we also wanted to try the **Maximize Resource Allocation** property offered by Spark. As per Spark documentation [7], maximize resource allocation property enables the cluster executors to utilize the maximum resources possible on each node. To achieve this,it calculates the maximum compute and memory resources available for an executor on an instance and then sets some default properties. Those properties are mentioned below:

1)spark.default.parallelism (equal to 2 X CPU cores)
2)spark.driver.memory
3)spark.executor.memory
4)spark.executor.cores
5)spark.executor.instances

Based on the above, we can state the hypothesis , that when enabling the maximizeResourceAllocation property at the initial configuration of our cluster, we will have a similar performance with our execution in section A above. This is because we have also set the default number of parallelism to 2 times the number of CPUs on the cluster, and we have also calculated the optimal number of executors and memory based on the available resources.

After running our application on the whole dataset, we resulted in completing this spark job in **12 minutes** using again 1c4.8xlarge master node and 20 c4.8xlarge nodes. This is very close to the performance we had in A, which ensures our hypothesis that our calculations were close to optimal. Below we present a figure of the whole dataset execution as well as a table with the execution times in each case.

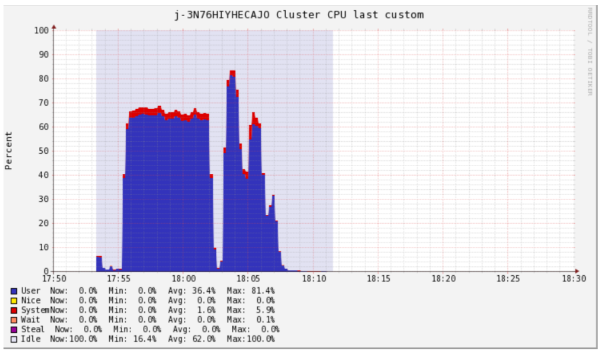


Figure 6: CPU usage when processing the entire GDelt dataset

Execution	**Execution time**
Manual resource allocation	14
MaximizeResourceAllocation enabled	12

Table 2: Average Execution Time with different way of resource allocation

4) Data Serialization

In this section we will also describe another attempt to optimize our application, despite the fact that it did not seem to actually improve our performance Based on [1] and taking into account that serialization plays an important role in the performance of any distributed application we decided to tune the data serialization in order to optimize our Spark application. More specifically, we used the Kryo serialization which is a library that Spark can use in order to serialize objects in a more quick way. According to [1] Kryo has a significant difference in speed and in compactness in comparison to the the default Java serialization.However, practically we saw a small difference in the execution time but we can not safely assume that this will make a big difference in a larger scale. In the table below we have included the execution time difference in the two cases. The test was performed on an 11 c4.8xlarge node cluster,with maximumResourceAllocation enabled and

30,000 segments.

Kryo Serialization	**Java Serialization**
Execution Time Experiment 2: 4 minutes	Execution Time Experiment 2: 4.2 minutes

Table 3: Average Execution Time with Kryo and Java Serialization

Metric Optimization

Finally, in the previous Lab assignment we thought that the most important metric for a company would be to reduce the cost. Hence, taking into account that we were able to process the entire dataset in under 30 minutes, we believe that it makes sense to try to to use a smaller cluster. Doing so, we try to identify which would be more efficient in terms of both cost and execution time. More specifically, we use a weighted penalty term p and we define this penalty of each cluster to be: $p = 0.9 * cost/12dollars + 0.1 * time/30minutes$.
(12 dollars and 30 minutes are the limits set from the assignment and hence we use them for normalization).Therefore, we calculate the cost of the big cluster that has 21 nodes and took 12 minutes to run and the cost of the small cluster that has 11 cores and took 20 minutes to run with maximizeResourceAllocation enabled. The cost for each node is 0.3 dollars/hour. Finally, we choose the cluster that results to the smaller penalty term.

10 node cluster: $p_{10} = 0.9(\frac{0.3\times11\times0.25}{12}) + 0.1(\frac{20}{30}) = 0.128$

20 node cluster: $p_{20} = 0.9(\frac{0.3\times21\times0.2}{12}) + 0.1(\frac{12}{30}) = 0.1345$

Based on the above results, we can conclude that is would be more appropriate for a company to use a cluster of 11 c4.8xlarge cluster (including master node) that would need 20 minutes to produce results, than using a 21 node cluster despite the fact that it is faster.

References

[1] *https://spark.apache.org/docs/latest/tuning.html*

[2] *http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/*

[3] *http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/*

[4] *https://jaceklaskowski.gitbooks.io/mastering-apache-spark/*

[5] *https://spark.apache.org/docs/latest/tuning.htmllevel-of-parallelism*

[6] *https://databricks.com/session/top-5-mistakes-when-writing-spark-applications*

[7] *https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.htmlemr-spark-maximizeresourceallocation*