Lab 2 Report

Super Computing for Big Data

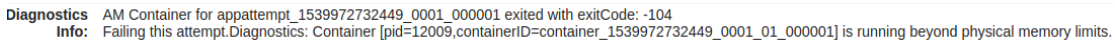
Group 4

Kyriakos Psarakis (4909437)

Rafail Skoulos (4847482)

Initial Run

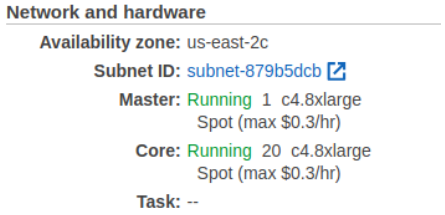
To start with, in this lab we had to chose between our two implementations from Lab1. The first thing we did was to try and run our code on AWS, starting with data from 1 month and then one year. All went well and the RDD implementation seemed to be the optimal choice but, when we tried to run our application with the 4TBs of the entire dataset we had the following error on our resource manager.



So, our first thought was to check the cluster's configuration and see why we had that memory limit error. What we observed was that if we did not specify the configuration to the cluster, it takes the defaults which are insufficient for our big data problem. Consequently, we searched on the AWS EMR documentation [1] and found that if we set the maximizeResourceAllocation setting to true the cluster will automatically configure Spark to run with optimized settings based on our cluster instances. Creating one executor per Node giving him most of the Nodes resources.

Framework choice

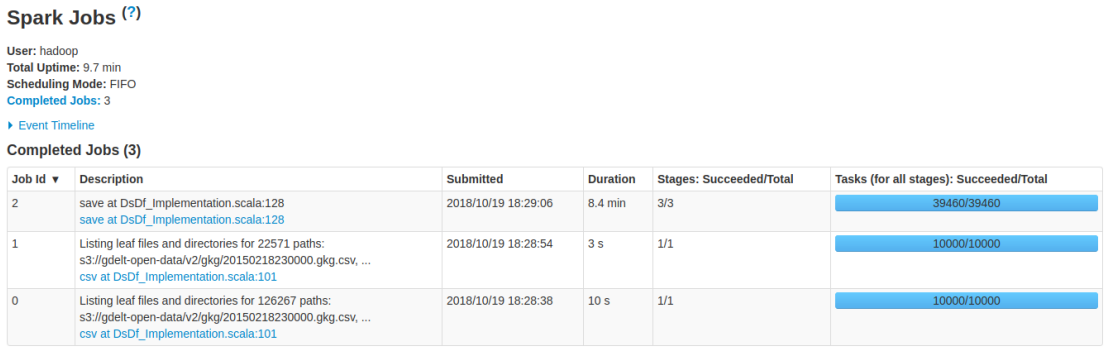
Now that both of our codes run on the cluster for the entire dataset we need to chose between one of our two lab1 implementations. We executed the RDD implementation in the following setup but, the runtime exceeded the minimum of 30 minutes.



In the same setup we executed the Dataframe implementation and in the initial run we had a runtime of 8.5 minutes. That led us to chose to improve upon the Dataframe implementation for lab2. Before showing graphs for our initial Dataframe run we have to note that the only changes we did to the code form lab1 was to remove the line that printed the results on the driver node and replace it with the following line: ds2.write.format("json").save(args(1)) that saves each nodes results in a distributed fashion into Amazons S3. Additionally, in our lab1 discussion with the TAs we were told to double check if the way that we solved the problem worked as expected in the cluster. Reason being that we used the rank Window function to get the top10 topics per day. We observed that the results were as expected furthermore, in [2] we read that the window functions do indeed work in a cluster.

Initial Measurements

In the following screenshots from the Spark History server we observe that we run our analysis on the entire dataset in 8.6 minutes. Moreover, in the executor's tab we see that we have 20 executors, 1 per Node. A drawback with having executors with so much RAM is that the garbage collection time is bigger than the normal of 10%, in our case 19,4%. Also, we get that the Shuffle read/write are 61.7 GB 1.5% of our 4TB dataset which in our opinion is good for our use case that requires 2 wide dependencies in order to reach our desirable outcome.



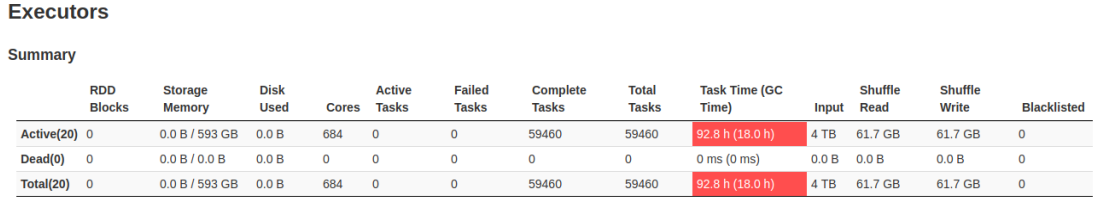
Spark Jobs (?)

User: hadoop
Total Uptime: 9.7 min
Scheduling Mode: FIFO
Completed Jobs: 3

▸ Event Timeline

Completed Jobs (3)

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	save at DsDf_Implementation.scala:128 save at DsDf_Implementation.scala:128	2018/10/19 18:29:06	8.4 min	3/3	39460/39460
1	Listing leaf files and directories for 22571 paths: s3://gdelt-open-data/v2/gkg/20150218230000.gkg.csv, ... csv at DsDf_Implementation.scala:101	2018/10/19 18:28:54	3 s	1/1	10000/10000
0	Listing leaf files and directories for 126267 paths: s3://gdelt-open-data/v2/gkg/20150218230000.gkg.csv, ... csv at DsDf_Implementation.scala:101	2018/10/19 18:28:38	10 s	1/1	10000/10000



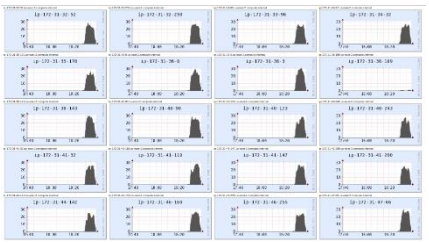
Executors

Summary

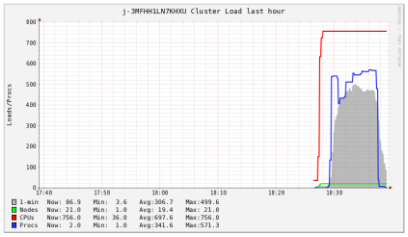
	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(20)	0	0.0 B / 593 GB	0.0 B	684	0	0	59460	59460	92.8 h (18.0 h)	4 TB	61.7 GB	61.7 GB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(20)	0	0.0 B / 593 GB	0.0 B	684	0	0	59460	59460	92.8 h (18.0 h)	4 TB	61.7 GB	61.7 GB	0

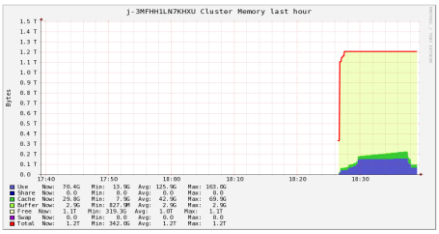
The next step is to check how our cluster's resources get utilized in order to detect possible bottlenecks and try to find improvements to tackle them.

To start with we check in the two following screenshots the CPU utilization of our Nodes. Firstly, we see that we don't fully utilize our cluster's power (running at 70%) but the load is evenly distributed across all the nodes. This could improve if we increase the number of tasks that run on each Node but, it is not that simple due to the case that the bottleneck could be on another component of the cluster for example, the network connection could be unable to feed data to our executors fast enough leading to lower than maximum CPU usage percentages.

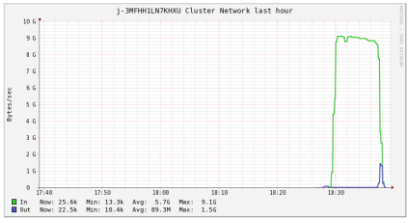


The next two screenshots show the load of the cluster and RAM usage. Those two are as expected, due to the 70% CPU usage and the RAM is related to the task's input size.





Now, for the last screenshot from Ganglia, we have the component with the higher probability of being the bottleneck, we say probability because Amazon does not state their cluster's network speed. Additionally, we see it sit around the 10Gigabit mark and it is safe to assume that their connection is the standard 10Gigabit being the main bottleneck to our application.



The overall maximum cost of this run is:

(0.3 (cost per node) *20(#nodes)) ($/h) *(8.5(runtime mins)/60(mins/h)) (h) = 0.85 $

Code Optimizations

In this section we tried to find some improvements to our code.

Kryo Serializer

The first thing that we have found when we where searching for a way to optimize our codes performance was to introduce to our code the Kryo serializer. We made that change but, the results were the same to worse. Then with further search we found in [3] that the Dataset API uses its own “specialized encoder to serialize the objects for processing or transmitting over the network which explained our results.

Avoid complex UDFs

Another highly suggested optimization for the Dataset API is to replace the complex UDFs. We have only used one simple UDF that transformed the final output data to our desired format. After removing it, the runtime did not change so, we chose to keep it.

Cache

Then we tried to improve our performance by inspecting if we can use some memory only caching in order to improve our application’s performance. However, due to our DAG being just a straight line of transformations and no intermediate results was reused again from any other part in our application, the results of having cached a transformation in our DAG was the same as the original run.

Parallel writes to S3

The next improvement in our mind was to improve the way our results where outputted. In the beginning we just printed them to our driver node but we needed to keep them in files in order to persist them. That led to us collecting all the results to the driver node, merging them and then writing them in a single file. However, that was not the optimal way to write the output to S3 so, we changed it by making every node writing its results to S3 independently. That lead to a big improvement in performance of around 35%. In addition, if we wanted the results in a single file we could just download them and concatenate the files. Finally, we chose to write the output to S3 because its native to the Amazon’s environment providing the best write speeds so, we didn’t try a different file system like HDFS.

EMR Cloud Optimizations

In this section we will try to fine tune our cluster's configuration in order to improve performance.

Double the amount of executors

Another improvement that came to mind was what would happen if we increased the parallel computation by doubling the number of executors per node. To do that we split the resources of each node in half so, each executor would have equal computational power. As we can see in the following screenshots we had a 9.52% improvement in runtime and due to lowering the memory pool of each executor the garbage collection time dropped from 19.4% to 10.2%.

Spark Jobs (?)

User: hadoop
Total Uptime: 8.8 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

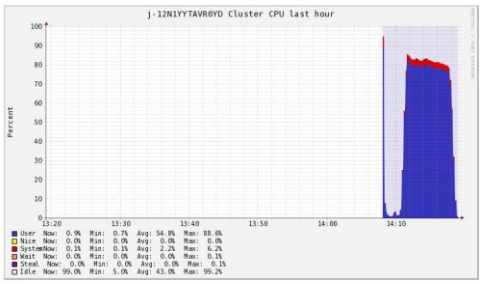
Completed Jobs (2)

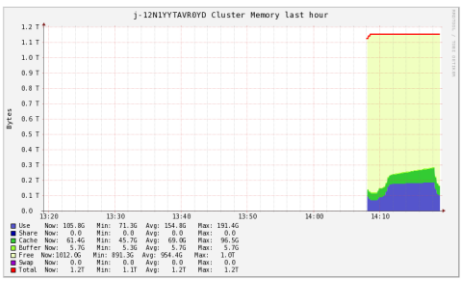
Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	save at DsDf_Implementation.scala:126 save at DsDf_Implementation.scala:126	2018/10/20 14:10:49	7.6 min	3/3	39478/39478
0	Listing leaf files and directories for 126346 paths: s3://gdelt-open-data/v2/gkg/20150218230000.gkg.csv, ... csv at DsDf_Implementation.scala:100	2018/10/20 14:10:26	10 s	1/1	10000/10000

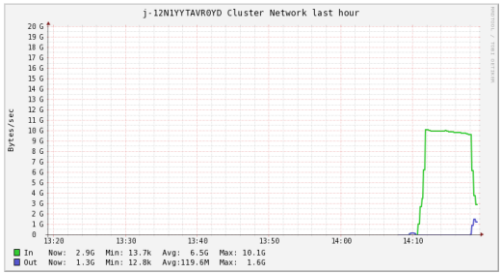
Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(40)	0	0.0 B / 487.2 GB	0.0 B	624	0	0	49478	49478	76.4 h (7.8 h)	4 TB	61.7 GB	61.7 GB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(40)	0	0.0 B / 487.2 GB	0.0 B	624	0	0	49478	49478	76.4 h (7.8 h)	4 TB	61.7 GB	61.7 GB	0

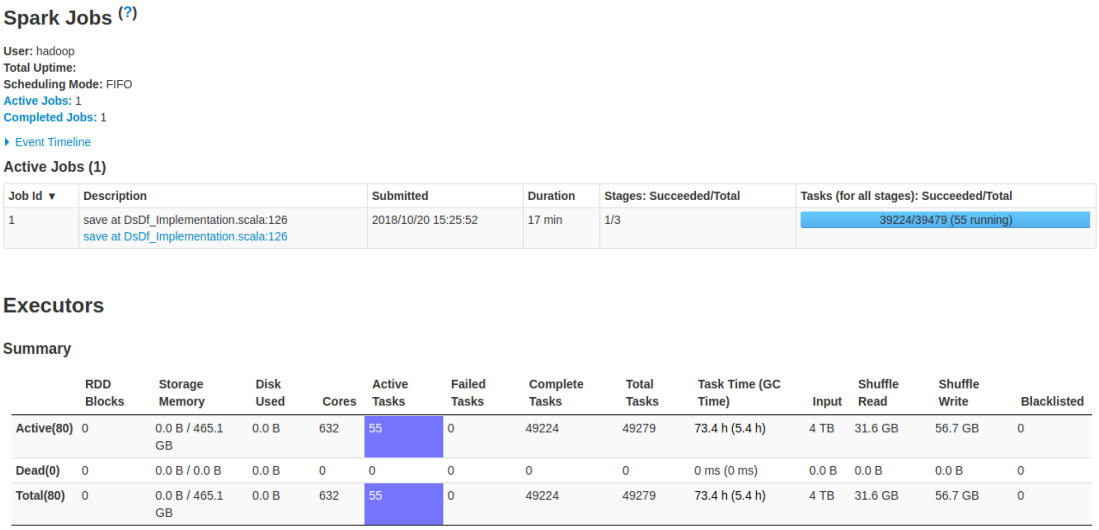
Additionally, the following graphs in ganglia showed as expected better CPU and memory utilization, the CPU rose from 70% to 85% and the memory use increased 30 Gb. However, the 10Gigabit network bottleneck still remained. The cost of this run is 0.76$.







Finally, this divide and conquer approach does not work indefinitely, as we observe if we quadruple the amount of executor performance issues begin to arise leading to a substantial decrease in performance. Reason being that we decrease the computational power of our nodes a lot and to add to that we increase the communication between those executors. Moreover, if we increase them even more that could lead to OutOfMemoryExeption like those in our initial run before setting maximize resource allocation to true.



Spark Jobs (?)

User: hadoop
Total Uptime:
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1

Event Timeline

Active Jobs (1)

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	save at DsDf_Implementation.scala:126 save at DsDf_Implementation.scala:126	2018/10/20 15:25:52	17 min	1/3	39224/39479 (55 running)

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(80)	0	0.0 B / 465.1 GB	0.0 B	632	55	0	49224	49279	73.4 h (5.4 h)	4 TB	31.6 GB	56.7 GB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(80)	0	0.0 B / 465.1 GB	0.0 B	632	55	0	49224	49279	73.4 h (5.4 h)	4 TB	31.6 GB	56.7 GB	0

Metric to Improve

In Lab1s' report the metric that we chose to improve was the price to performance ration of our implementation. The way that we approached this was by decreasing the number of nodes in order to decrease the price but, that also decreased performance. Finally, we can chose a cluster setup that suits best our price and time constraints.

Dataset	RunTime	#Nodes	#Executors	Price/Node/Hour	Run'sCost	$/TB
4 TB	8.5 min	20	20	0.3 $/h	0.85$	0.21
4 TB	7.6 min	20	40	0.3 $/h	0.76$	0.19
4 TB	10.1 min	15	30	0.3 $/h	0.75$	0.189

From these runs, it's safe to conclude that in our use case the best option is the one with the 20 nodes with the improvement of the executor increase. The approach to decrease the number of nodes to decrease the price is baseless, as we see the price decreases by less than 1% but the runtime plummets to a 24.7% decrease. In conclusion, the best price to performance in our case is to lease the 20 Nodes that will finish the job faster costing less due to taking less time to finish.

Future improvements

A future improvement in respect to our code could be to find to not create so much garbage, in order to decrease the time that takes the garbage collector to do its job. Additionally, we tried but could not find a way to change the default garbage collector inside Amazon's EMR and changing that could also lead to performance increase. Finally, with the chaos that is spark configuration we can still make some minor tweaks in some little-known configuration parameters that could possibly lead to a minor performance increase.

References

[1]https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html

[2] https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html

[3] https://spark.apache.org/docs/2.1.0/sql-programming-guide.html