Super Computing for Big Data (ET4310) - Lab 2

Chia-Lun Yeh (4718836) and Shikhar Dev (4773071)

October 20, 2018

1 Introduction

This report details the implementation of running a Spark application on the Amazon Web Service (AWS). This application aims to solve an imaginary use case where we want to predict future trends using past popular topics discussed in news articles. The prediction is done on a daily basis. To achieve this, we utilize the dataset available from GDELT Global Knowledge Graph (GKG) Version 2.1 of the GDELT project[1]. The GDelt project aims to monitor news outlets around the world and extract information from them. It is updated every 15 minutes by adding one CSV file to the dataset. Each row of the file contains the data from one article. We are interested in one column of the data, the ALLNAMES field. This field contains all proper names referenced in the news articles, such as people, organization, events, movements, wars, and named legislation[2], which we use as the topics.

To be more concrete, we retrieve the top 10 mentioned topics of each day by counting the terms in the ALLNAMES fields and aggregate them by date. Let's assume that past data are still being augmented with archive articles from different sources and are thus changing. This means that we need to process the entire dataset each day to get the most accurate result.

2 Configuration

We start with our original RDD implementation, which runs faster on our local machine than the Dataset implementation. The configuration and runtime on a subset of the dataset is shown in Table 1.

Table 1: Run time on subsets of the dataset with a cluster.

	# instance	instance type	memory (GiB)	# vCPU	956 files	21456 files
Master	1	c4.xlarge	7.5	4	2.2 min	20 min
Core	15	c4.2xlarge	225	120		

As the time performs better than linear growth with respect to the number of files, it seems to scale well. Running on the entire dataset, however, results in OutOfMemoeryError of Java heap space. This is the case even if the instance type is changed to c4.8xlarge, which has a 60 GiB of memory. This problem signals that JVM is running out of memory either on the driver or the executor. We use Apache Ganglia to monitor the cluster but the problem cannot be spotted. Examining the Environment tab of SparkUI, we see that only 4 cores are used per executor, which does not match the resource that we have.

[1] https://www.gdeltproject.org/

[2] http://data.gdeltproject.org/documentation/GDELT-Global_Knowledge_Graph_Codebook-V2.1.pdf

Table 2: Settings of a cluster with 1 m4.xlarge master node and 10 m4.4xlarge core nodes.

Setting	Description	Default	Apply config
spark.driver.memory	Amount of memory to use for the driver process.	1024M	11171M
spark.executor.memory	Amount of memory to use per executor process.	5120M	51316M
spark.executor.cores	# cores to use on each executor	4 (SparkUI)	32
spark.executor.instances	# executors	10 (Ganglia)	10
spark.default.parallelism	# partitions in RDDs	# cores on all executor	640

3 Modification

Instead of setting the value manually, we set the *maximizeResourceAllocation* flag to true so that Amazon EMR automatically updates the spark-defaults.conf file. The executors are configured to utilize the maximum resources possible on each node in a cluster. The following values are set dynamically in this case: driver memory, executor memory, number of executors, number of cores used in each executor, and RDD parallelism [3]. In table 2, we compare the values between before and after applying the configuration. When the flag is not set, most of the values were not shown in SparkUI and the default values are considered to be used. After applying the flag, the 16 GiB memory of the master node and 64 GiB memory of the core nodes are almost fully utilized. Some memories are used as off-heap memory so these values are reasonable. (Side note: The number of executor.cores is 2 times the number of vCPUs, and the parallelism is 4 times as we expected. It is a bit mysterious but we do not mind higher parallelism)

With this setting, we were able to process the entire dataset with a significant performance boost. Although we encountered loss of around 30 tasks for the first run of Config 2, the second run was successful. We believe that the loss of tasks was due to network issue within the cluster and not in our code. The configurations (Config) and the runtime are as follows:

- Config 1: 1 master (m4.xlarge) and 15 cores (m4.4xlarge)
  Time: 25:10

- Config 2: 1 master (c4.8xlarge) and 19 cores (c4.8xlarge)
  Time: 13:44

To this end, we have successfully met the requirement of running the application within half an hour using 20 c4.8xlarge instances. However, the cluster configuration is chosen quite arbitrarily and requires some justification. Figure 1 shows some graphs we get from Apache Ganglia. Note that the graphs contain half of the first try but we are analyzing only the second run (the right half of each graph contains the information that is of interest). The time when each stage is being executed is shown in the bottom graph. The graphs represent the CPU usage of each instance in the cluster, the CPU usage of the cluster, the memory usage of the cluster, and the network of the cluster (left to right, top to bottom).

We can observe a few things:

1. The master node is an overkill as the CPU utilization is very low.

2. The network bandwidth, in this case around 10 GB/s, is a bottleneck when reading the input files. This argument is based on the fact that the network has reached its maximum throughput at the first stage but CPU usage is around 70%.

3. Memory capacity can possibly be reduced as the usage is less than 60% even at its peak.

In the following, we will optimize the application based on these observations.

Figure 1: Analysis from Apache Ganglia of Config 2.

4 Improvements

In this section, we will discuss the configurations that we have experimented with based on previous observations as well as new observations along the way. But before diving into all the experiments, we also want to make sure that we have optimized the application code itself. The application is somehow simple, it reads in the data files once, execute 2 shuffles to group the data by the (date, words) pairs, and then by the date. The only thing that we could do is reduce the amount of data that is being shuffled as shuffling is an expensive operation that involves disk I/O, data serialization, and network I/O [2]. As a side note, we don't think there is a need to cache the RDDs as no single RDD is used in an iterative way [6].

4.1 Modifying the application

We could not think of a way to write our code so that it requires less shuffling. However, we can configure the Kryo library to serialize data when shuffling. By default, Spark serializes objects using Javas *ObjectOutputStream* framework, but the Kryo library is said to be significantly faster and more compact than Java serialization [2]. We added the configuration as suggested in [1] and ran it on 21456 files with Config 4. However, both runs have the same outcome as shown in Figure 2. The promised speed and compactness did not show in our application. It is possible that we did not configure it correctly or the advantage will show if we process on the entire dataset.

Description	Submitted (UTC+2)	Duration	Tasks succeeded / total	Input	Output	Shuffle read	Shuffle write
collect at lab1_rdd.scala:57	2018-10-15 13:54 (UTC+2)	5 s	608 / 608			2.1 GiB	
map at lab1_rdd.scala:46	2018-10-15 13:53 (UTC+2)	20 s	608 / 608			8.2 GiB	2.1 GiB
flatMap at lab1_rdd.scala:40	2018-10-15 13:52 (UTC+2)	1.7 min	21,461 / 21,461	564.1 GiB			8.2 GiB

Figure 2: The duration and the amount of data shuffled running a subset of the dataset.

From the experiment, we came to the observation that our application might be by far limited by the input stage. Since we process with only 2 columns of each file, the input overhead is a lot more critical than the data that needs to be shuffled. We did not include Kryo in the other experiments.

4.2 Tuning instance type and numbers

We mainly use compute-optimized instances as the memory requirement of the application is not a bottleneck. C5 is the newest generation of the compute-optimized family and is said to provide improved processing power at a lower cost [4]. Thus we mainly experiment with the C5 family. Following are the motivation for our experiments:

- Config 3: Compare whether C4 and C5 family has an actual difference.
- Config 4: Test whether m4.xlarge is sufficient as a master node; test whether c5.4xlarge would double execution time compared to c5.9xlarge.
- Config 5: The hypothesis here is that using a few large nodes can reduce the amount of data transferred between nodes. We assume that data transfer between CPUs within a node is faster.
- Config 6: Test the effect of many small nodes.

Table 3: Summaries of the execution time and some Ganglia metrics of different configurations.

Config	Settings	Exe. Time 1st Stage (min)	Exe. Time 2nd Stage (min)	Total Time (min:sec)	Max CPU usage (%)	Max Network BW (GB/s)
1	1 master (m4.xlarge) 15 cores (m4.4xlarge)	19	5.0	25:10	Nan	Nan
2	1 master (c4.8xlarge) 19 cores (c4.8xlarge)	9.3	4.0	13:44	80.7	10.6
3	1 master (c5.4xlarge) 19 cores (c5.9xlarge)	5.3	8.5	14:34	82.4	14.0
4	1 master (m4.xlarge) 19 cores (c5.4xlarge)	11	2.1	13:51	79	6.8
5	1 master (m4.xlarge) 9 cores (c5.18xlarge)	8.5	Fail	Nan	Nan	8.5
6	1 master (m4.xlarge) 25 cores (c5.2xlarge)	16	Fail	Nan	Nan	4.7

Table 3 shows the results of the experiments. Config 3, with the C5 family, didn't run faster than its C4 counterparts. We made sure that it ran smoothly without removal of executors. Note that we tuned down the master node from 8xlarge to 4xlarge but the master node is under very low load and should not have any effect on the performance.

Config 4 is a total surprise. The configuration has less than half of the parallelism than Config 2 and 3, yet it ran as fast as them. It ran the second stage a lot faster in particular. We observed that the shuffle-write after the first stage was 51.7 GiB for this configuration while in Config 3 it was 70.4 GiB. We're not sure if there is randomness in task assignment and partition that could lead to different shuffling behaviour for the same code.

Config 5 failed due to loss of tasks. We ran it twice and in both runs, it happened after the first stage was finished. Both run needed to rerun the first stage to recover the loss. We're not sure what the problem was. However, we observed 2 things from the graphs of Apache Ganglia as shown in Figure 3. First of all, 18.xlarge has a low utilization of CPU. The 72 vCPUs is not efficiently used. Second, network bandwidth (BW) can change drastically in two consecutive runs.

Config 6 failed at the second stage because the node memory was insufficient. (Container killed by YARN for exceeding memory limits) This error can be fixed by disabling NodeManagers

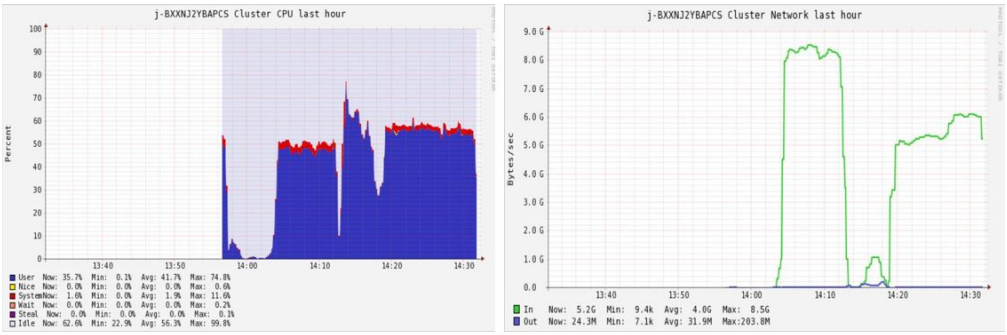

Figure 3: Graphs from Apache Ganglia of two consecutive runs on Config 5

checking virtual memory usage in containers. However, we checked the nodes' memory requirement of this application from other configurations and concluded that 16 GiB of memory would be very tight. For example, in the left plot of Figure 4, it seems that the maximum usage can reach around 20 GB. Therefore, 2xlarge types are too small for the application.

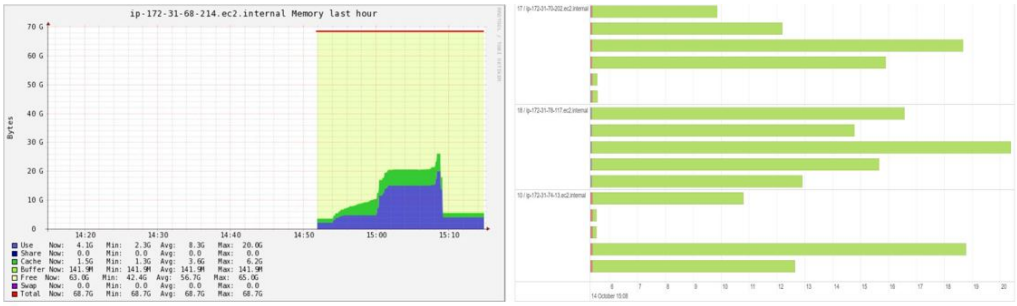

Figure 4: Memory usage of a random node and the executor process of Config 3.

5 Recommendation of Configuration

We decided to use the cost of the application as the metric to choose our final configuration. The cost is defined as follows

$$cost = executionTime(sec) \times costPerTime(\$/sec) \tag{1}$$

Development and maintenance costs are also an important factor to consider but we neglect those at the moment because we believe that if we find the best configuration of the application, it can be run for years with minimal maintenance. The development cost would also be evened out after a long time.

Figure 5 summarizes our experiments and the costs. For each of these experiments, we used the same application code. The dataset was of roughly 3.7 TB for each of these experiments. Since the database is updated every 15 minutes, the amount of data being processed continuously increases. We did not note down the exact size but the difference is several hundreds of files at the maximum. The cost is calculated by the on-demand cost of Amazon EC2 instances plus the cost of Amazon EMR. We use the on-demand cost due to the ease of comparison. Spot-instances can be used for the application as it is not real-time and thus tolerant to some instability.

Based on the above observations, we concluded that Config 4 is the most efficient configuration, both in terms of cost as well as runtime. Figure 6 confirms that the master node is more efficiently used in terms of CPU and memory, and the overall cluster memory is also adequate.

Config ID	C1	C2	C3	C4	C5	C6
Master Node	1 M4 . XLarge	1 C4 . 8XLarge	1 C5 . 4XLarge	1 M4 . XLarge	1 M4 . XLarge	1 M4 . XLarge
Worker Node	15 M4 . 4XLarge	19 C4 . 8XLarge	19 C5 . 9XLarge	19 C5 . 9XLarge	9 C5 . 18XLarge	25 C5 . 2XLarge
Runtime (Minutes)	25.17	13.73	14.57	13.85	Did not Complete	Did not Complete
EC2 Cost (Hourly Cost) (USD)	12.2	31.82	29.41	13.12	27.74	10.15
Associated EMR charges (USD)	0.96	5.4	5.3	5.19	2.49	2.68
Total hourly cost (USD)	13.16	37.22	34.71	18.31	30.23	12.83
Actual Operation Cost for for 3.7 TB	5.52	8.52	8.43	4.23	NA	NA

Figure 5: A summary of the costs of the experiments.

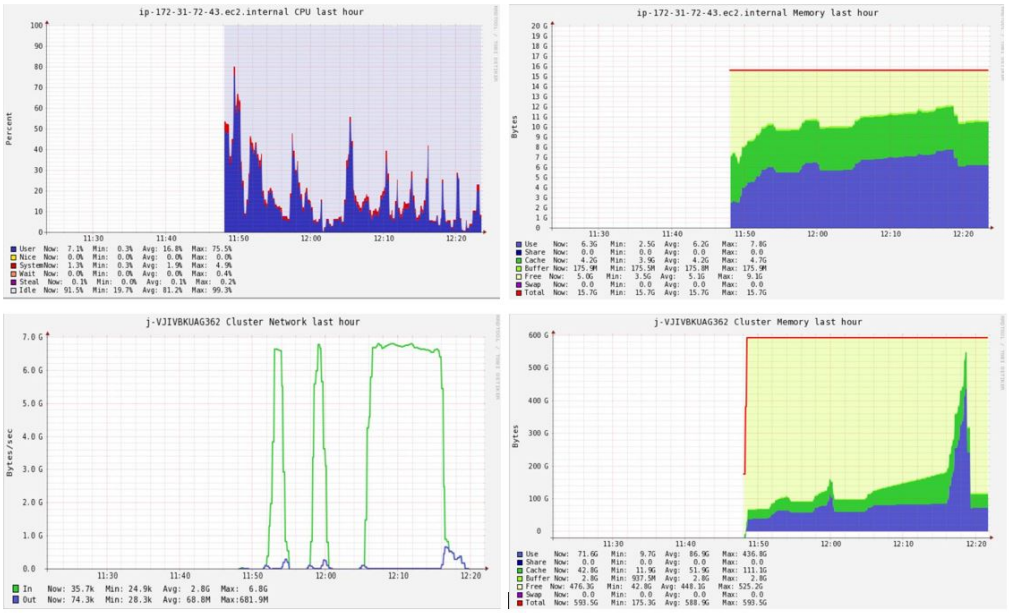

Figure 6: Top: CPU and memory usage of the master node. Bottom: Network and memory usage of the cluster.

It should be noted that there are more modifications possible to tune the performance to even further efficiency. However, we do not expect significant performance gains to justify the time required in further tuning at this point.

6 Conclusion

We were able to successfully process the entire data set within 30 minutes for multiple configurations. The primary objective of our experiments was to find the configuration that processes the entire data set the fastest, and with the least cost. Beside the best configuration we found, we could draw some conclusions from the experiments.

- The network BW between Amazon EMR and Amazon S3 (where the data is located) is arguably the most important factor to the execution time of our application. However, the BW is unknown beforehand and unstable. As observed from Table 3, it is unclear whether BW depends on the number of instances, the instance types, or both, and how it depends on them.
- The processing time of the first stage depends on the network. However, the processing time of the second stage is somehow arbitrary in our case. We don't observe reduced processing time when more vCPUs are dedicated.
- The master node is not involved in the actual computation, but only responsible for scheduling and monitoring operations. Thus, we dont need a compute-intensive node. A general

m4 node is suitable since it provides a good balance between memory and computing power.

- Tuning Spark application is a tedious but important process. Apache Ganglia can help with identifying bottlenecks but changing the number of instances and the instance type of the cluster do not lead to expected results. More benchmark tests need to be done on the network throughput and processing power of different cluster configurations.

7 Future Improvement Plans

Finally, we want to point to some possible improvement plans.

1. Investigate methods to assign tasks that process the files of the same date to the same node. If a node can access and process the data in a unit of date, it can do all the aggregation within itself. This can minimize shuffling.

2. We compared BW mainly on the cluster level but we had a preliminary observation that each node in a cluster can have very different throughput. This is worth looking into although it's quite possible that there's nothing we can do about it.

3. Dive into Yarn configuration as suggested in [5] to see if we can better optimize the resource for the tasks.

4. As the dataset increases, the amount of data being shuffled would increase. This makes us believe that the Kryo library is still worth investigating.

5. Write a script to automate all the tests instead of manually launching, monitoring, and terminating each cluster.

Table 4: Memory, CPUs, storage, network, and the costs of several instance types that we use.

Instance Types	Memory	vCPUs	Storage	Network	On-Demand Cost
m4.xlarge	16.0 GiB	4 vCPUs	EBS only	High	$0.200 hourly
m4.4xlarge	64.0 GiB	16 vCPUs	EBS only	High	$0.800 hourly
c4.xlarge	7.5 GiB	4 vCPUs	EBS only	High	$0.199 hourly
c4.2xlarge	15.0 GiB	8 vCPUs	EBS only	High	$0.398 hourly
c4.8xlarge	60.0 GiB	36 vCPUs	EBS only	10 Gigabit	$1.591 hourly
c5.2xlarge	16.0 GiB	8 vCPUs	EBS only	Up to 10 Gbps	$0.340 hourly
c5.4xlarge	32.0 GiB	16 vCPUs	EBS only	Up to 10 Gbps	$0.680 hourly
c5.9xlarge	72.0 GiB	36 vCPUs	EBS only	10 Gigabit	$1.530 hourly
c5.18xlarge	144.0 GiB	72 vCPUs	EBS only	25 Gigabit	$3.060 hourly

References

[1] 2 tunings you should make for spark applications running in emr. https://www.knowru.com/blog/2-tunings-you-should-make-spark-applications-running-emr/.

[2] Apache spark: Tuning spark. https://spark.apache.org/docs/latest/tuning.html.

[3] Aws big data blog. https://aws.amazon.com/blogs/big-data/submitting-user-applications-with-spark-submit/.

[4] Aws ec2 instance comparison: C4 vs c5. https://www.learnaws.org/2017/11/17/comparing-ec2-c4-c5/.

[5] Best practices for yarn resource management. https://mapr.com/blog/best-practices-yarn-resource-management/.

[6] To cache or not to cache, thats the million dollar question, by adrian popescu. https://unraveldata.com/to-cache-or-not-to-cache/.