Introduction

We are going to implement 3 sorting algorithms which are Spark, Hadoop and linux sort, after that we are going to compare the results and find out which is the best one.

We measured performance using the metrics listed below:

HADOOP

We firstly create the file using gensort and put it in hadoop.

Hadoop is a open source framework. Hadoop is used when handling large amount of data. Large amount such as 500GB or 1Pb can be handled using Hadoop. We are going to implement Hadoop sort by using the file generated by the gensort. Hadoop uses multiple machines to process large amount of data concurrently, which improves the overall efficiency and also reduces the time to process large amount of data. All the data is stored in hdfs. Hdfs provides a better output when compared to the traditional file systems. YARN (yet another resource negotiator) is used to schedule task. Map Reduce is another framework which takes the input dataset which can be computed in key and value pair output to receive good results.

SPARK

A distributed processing system that is open source and frequently used for big data tasks. Apache Spark supports general batch processing, streaming analytics, machine learning, graph databases, and ad hoc queries. It uses in-memory caching and optimized execution for quick performance.

1.1.1 Performance

TABLE 1: PERFORMANCE EVALUATION OF SORT (REPORT TIME TO SORT IN MILLISECONDS); EACH INSTANCE BELOW NEEDS A TINY. INSTANCE FOR THE NAME NODE

Experiment	Linux	Hadoop Sort	Spark Sort
1 small instance, 3GB dataset	93,021 ms	230000ms	292000ms
1 small instance, 6GB dataset	335,588 ms	930000ms	763000ms
1 small instance, 12GB dataset	710,880 ms	this config is not possible.	this config is not possible.
1 large instance, 3GB dataset	39,901 ms	187000ms	284000ms
1 large instance, 6GB dataset	157,236 ms	519000ms	607000ms
1 large instance, 12GB dataset	471,815 ms	896000ms	1367000ms
1 large instance, 24GB dataset	1,210,685 ms	2876000ms	2805000ms
4 small instances, 3GB dataset	N/A	25970ms	300770ms
4 small instances, 6GB dataset	N/A	790320ms	412700ms
4 small instances, 12GB dataset	N/A	299810ms	112128ms
4 small instances, 24GB dataset	N/A	9702337ms	2598000ms

Some of the things that will be interesting to explain are: how many threads, mappers, reducers, you used in each experiment; how many times did you have to read and write the dataset for each experiment; what speedup and efficiency did you achieve?

For 1 small instance 3 GB dataset in Linux, we use 2 threads
Hadoop sort: mappers = 48
Reducer = 1
Number of bytes read = 3001720120
Number of bytes written = 3150000000
Hdfs bytes read = 3001728110
Hdfs bytes written =3150000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 1 small instance 6 GB dataset in Linux, we use 4 threads
Hadoop sort: mappers = 95
Reducer = 1
Number of bytes read = 6403777700
Number of bytes written = 6300000000
Hdfs bytes read = 6403778320
Hdfs bytes written = 6300000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 1 small instance 12 GB dataset in Linux, we use 4 threads
Hadoop and Spark not possible

For 1 large instance 3 GB dataset in Linux,
Hadoop sort: mappers = 47
Reducer = 1
Number of bytes read = 3200029600
Number of bytes written = 31500000000
Hdfs bytes read = 32021129600
Hdfs bytes written = 31500000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 1 large instance 6 GB dataset in Linux,
Hadoop sort: mappers = 94
Reducer = 1
Number of bytes read = 6912270800
Number of bytes written = 6300000000
Hdfs bytes read = 6912372820
Hdfs bytes written = 6300000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 1 large instance 12 GB dataset in Linux,

Hadoop sort: mappers = 188
Reducer = 1
Number of bytes read = 14025470523
Number of bytes written = 12600000000
Hdfs bytes read = 1402558264
Hdfs bytes written = 12600000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 1 large instance 24 GB dataset in Linux,
Hadoop sort: mappers = 188
Reducer = 1
Number of bytes read = 15034643782
Number of bytes written = 25200000000
Hdfs bytes read = 15036637283
Hdfs bytes written = 25200000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 4 small instance 3 GB dataset in Linux,
Hadoop sort: mappers = 46
Reducer = 4
Number of bytes read = 3201770330
Number of bytes written = 31500000000
Hdfs bytes read = 3201800245
Hdfs bytes written = 31500000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 4 small instance 6 GB dataset in Linux,
Hadoop sort: mappers = 97
Reducer = 4
Number of bytes read = 6402270631
Number of bytes written = 6300000000
Hdfs bytes read = 6402568532
Hdfs bytes written = 6300000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 4 small instance 12 GB dataset in Linux,
Hadoop sort: mappers = 218
Reducer = 4
Number of bytes read = 13030607290

Number of bytes written = 12600000000
Hdfs bytes read = 13032903827
Hdfs bytes written = 12600000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

For 4 small instance 24 GB dataset in Linux,
Hadoop sort: mappers = 218
Reducer = 4
Number of bytes read = 25526507063
Number of bytes written = 25200000000
Hdfs bytes read = 25528348930
Hdfs bytes written = 25200000000

Spark Sort
Driver memory = 4GB
Executor Memory = 2GB
Executor Core = 1

PLOTS

Here is an example of a plot that has cpu utilization and memory utilization (https://i.stack.imgur.com/dmYAB.png), plot a similar looking graph but with the disk I/O data as well as a 3rd line.

Do this for both shared memory benchmark (your code) and for the Linux Sort.

You might find some online info useful on how to monitor this type of information (https://unix.stackexchange.com/questions/554/how-to-monitor-cpu-memory-usage-of-a-single-process). For multiple instances, you will need to combine your monitor data to get an aggregate view of resource usage.

Do this for all three versions of your sort.

After you have all four graphs (2 system configurations and 3 different sort techniques), discuss the differences you see, which might explain the difference in performance you get between the two implementations. Make sure your data is not cached in the OS memory before you run your experiments.

Questions and Answers

Q1. WHAT CONCLUSIONS CAN YOU DRAW?

Linux sort outcasts Hadoop sort and Spark when we are dealing with 1 small instance and 1 large instance. As we can see our output table for 3GB Hadoop sort small instance take 230000ms and 1 large instance takes 187000ms. The efficiency is increase near about 18%. The same can be convey from the output observation for another sort as well. The efficiency is increased when performed using 4 nodes for the same dataset. This is achieved because of parallel work, in this case the master's node can assign task parallelly to more workers.

The recent observation about the execution time for the make seems better for Linux and Mysort, this is because of the small dataset including other cost such as clustering cost, network communication of nodes cost and implementation cost Linux and Mysort outperform for small dataset. But the same is not true for higher dataset size where Hadoop and spark performs efficiently without any doubt.

Q2. Which seems to be best at 1 node scale (1 large.instance)? Is there a difference between 1 small.instance and 1 large.instance?

Single large instance performs better than a single small instance working with a single node because it has greater memory and core power. This conclusion is justified with the larger datasets in our experiments for 6GB and 12 GB. Hadoop sort, for instance, takes 230 seconds for a 3GB dataset on a single small instance, but just 187 seconds on a single large instance. This is evident with the other data sets as well.

Q3. How about 4 nodes (4 small.instance)?

Both our sorting techniques perform better when we tried running them on a cluster on 4 nodes. This conclusion is explained by the fact that these sorting techniques work on a parallel execution design as they have worker nodes available to delegate their task and get them done parallely.

It can be assumed incorrectly that linux sort is a better way to sort these files but it is not correct as the data files used in this experiment were small and hence the network overhead was a lot for such small data, which deteriorated the performance of our Hadoop and spark sorting.

A configuration with several small nodes is much beneficial when it comes to the cost of the machine and hence this approach is preferred over a configuration with large number of cores in a single machine.

Q4. What speedup do you achieve with weak scaling between 1 to 4 nodes?

Hadoops performance varied a lot on strong scaling performed on 3Gb data on small and large instance. On one large instance, it completed in 187 seconds as opposed to 230 seconds on one small instance. While performing the experiment on data sets with larger size, Hadoop's speed improved by a toll of 30% approximately.

Hadoop took 230 seconds for 3GB on 1 small instance and 299 seconds for 12 GB on 4 small instances, this suggests that weak scaling improved the performance by a factor of 4 approximately.

Q5. How many small.instances do you need with Hadoop to achieve the same level of performance as your shared memory sort?

For this assignment, no shared memory sort was performed as per the instructions but if we had to compare and calculate the number of instance required to achieve the same level of performance, we can calculate that by comparing the time for both and calculate the percent change in the time required by both the methods. This would give us a factor (x) that should be multiplied with the current number of instances.

Q6. How about how many small.instances do you need with Spark to achieve the same level of performance as you did with your shared memory sort?

For this assignment, no shared memory sort was performed as per the instructions but if we had to compare and calculate the number of instance required to achieve the same level of performance, we can calculate that by comparing the time for both and calculate the percent change in the time required by both the methods. This would give us a factor (x) that should be multiplied with the current number of instances. We have to consider other factors as well such as JVM starting time and the file size.

Q7. Can you predict which would be best if you had 100 small. instances? How about 1000?

Since Spark uses a lot of the main memory, it becomes expensive than Hadoop requiring secondary memory due to the increased cost of main memory required. When considering 100 nodes, Spark is a better approach than Hadoop which works more efficiently for the cost employed. When considering an even larger cluster with the number of nodes hitting the order of 1000, the primary memory required by Spark will exceed and hence Hadoop will become a better approach. Moreover, the cost of replacing nodes in case of failure will again become an expensive solution and hence Hadoop comes out as a better approach for 1000 instances.

Q8. Compare your results with those from the Sort Benchmark (http://sortbenchmark.org), specifically the winners in 2013 and 2014 who used Hadoop and Spark. Also, what can you learn from the CloudSort benchmark, a report can be found at

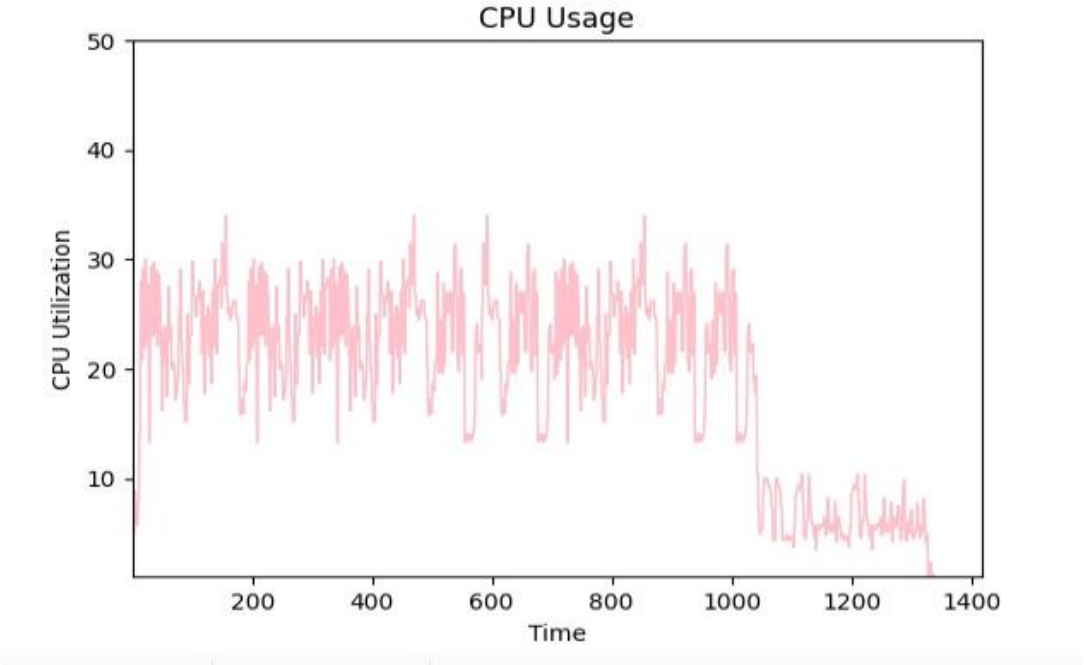
The Spark sort in our experiment on an instance of 4 core 8gb ram for a 3GB data took 292 seconds. The Spark sort in the benchmarking for 2014 used 207 EC2 instances with 32 cores and 244Gb of memory and sorted 100Tb of data in 1406 seconds. Considering the time for 3 GB data, it comes out to be 0.04218 seconds with 207*32 cores. Factorizing this value for 4 cores, we get 69.8 seconds for 3GB data using 4 cores. Hence our Spark sort is approximately 4 times slower than the one used in 2013.

The Hadoop sort implemented in the benchmarking sorted 102.5TB data in 4328 seconds with 2100 nodes of 2 cores with 64 GB memory. This boils down to 133 seconds for 3GB data on 4 cores. Our Hadoop experiment for 3GB data on 4 cores took 230 seconds. Hence the Hadoop implementation for 2014 is 2X faster than our implementation of Hadoop sort.

Hadoop Sort for Large Instance

CPU

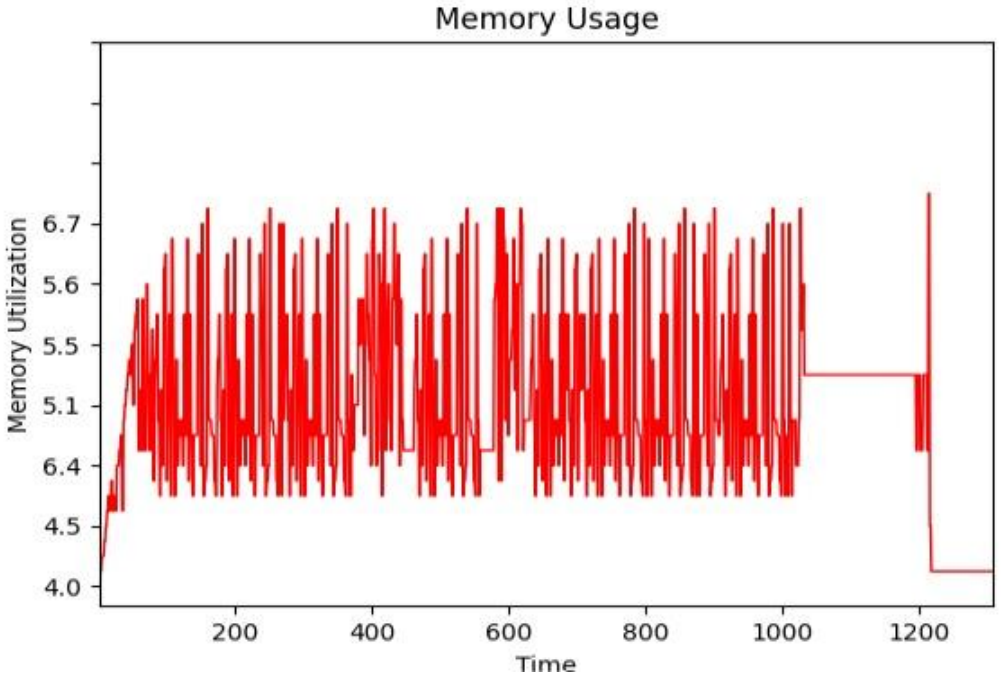
In this graph we are comparing the CPU utilization vs time, where CPU utilization is on Y-AXIS and time is on X-Axis. In sort, shuffle, and map phase we can see that the CPU utilization is more in the start. In the later part when the writing data operation is performed, we can see that the CPU usage is reduced efficiently, hence in the writing operation not much of CPU is used. In conclusion initial CPU usage is high and in the later stages it is low.



Time has been scaled to adjust the curve in the graph.

Memory

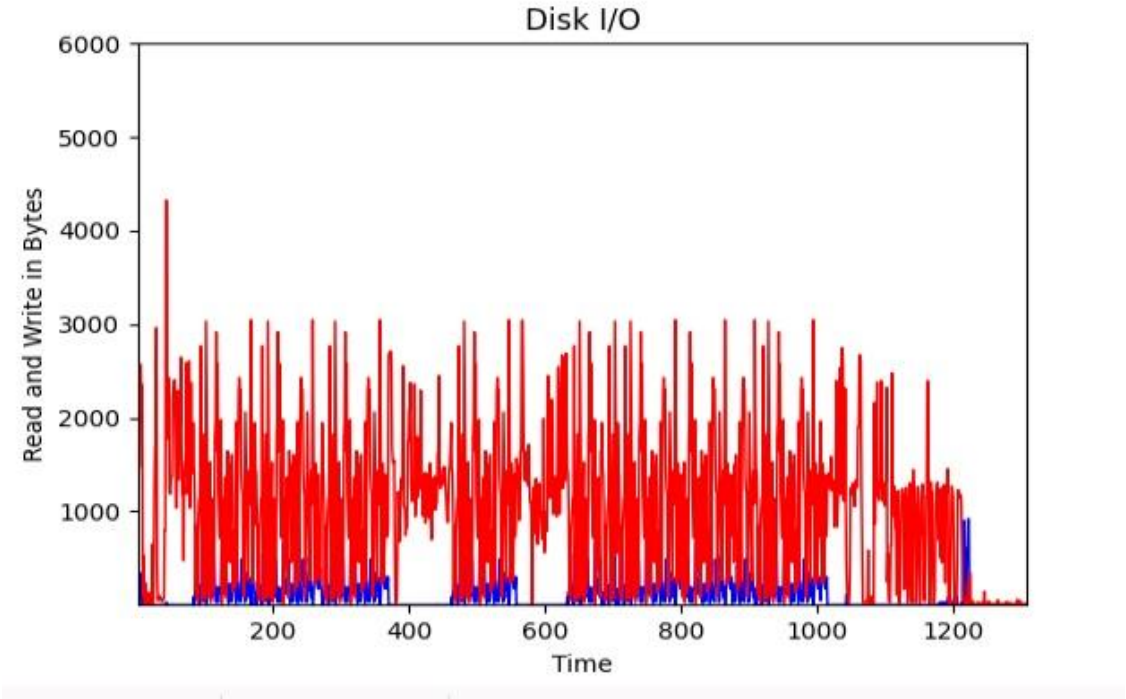
In the initial stage we can see that the memory utilization is more where sort and shuffle stage takes place later on it becomes stagnant and in the reduce stage the CPU utilization drops. We have used glance command to keep a tab on the CPU utilization and memory usage along with disk I/O.



Time has been scaled to adjust the curve in the graph.

Disk I/O

We can see that the Read and write operations initially are high. In the later stages you can see that the average in the latter part of the graph decreases substantially. Blue lines indicate the number of bytes read per second and the red line indicates the number of bytes written per second. These are operations are performed throughout the map and reduce functions and hence we can see a constant change in the graph.

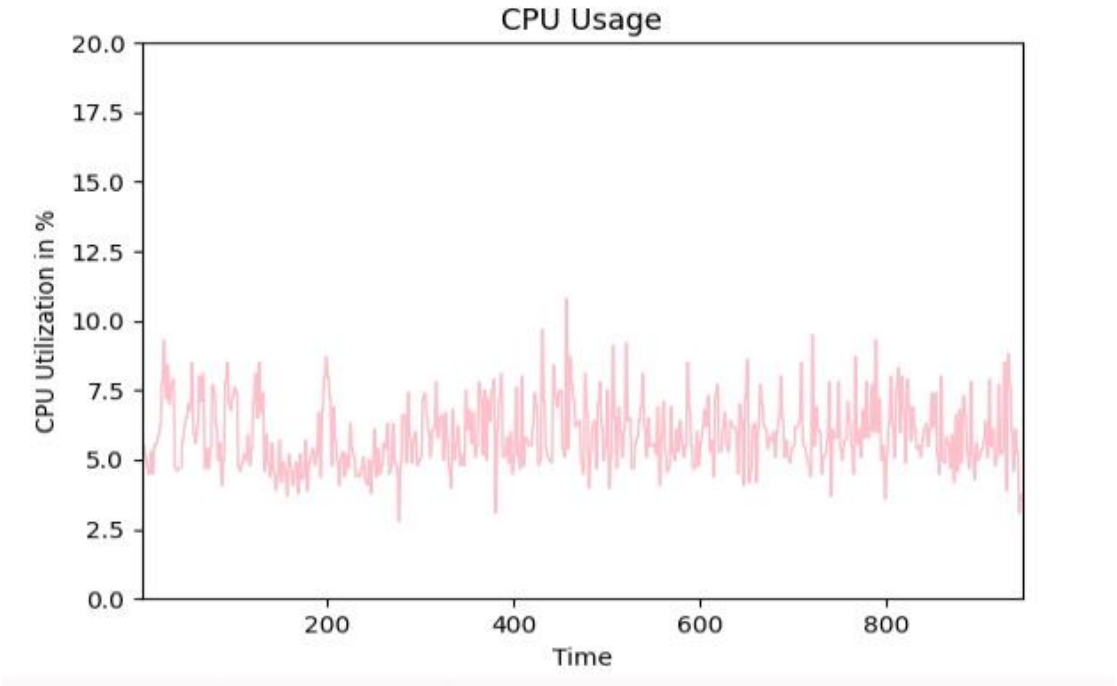


Time has been scaled to adjust the curve in the graph.

Spark Sort for Large Instance

CPU Utilization

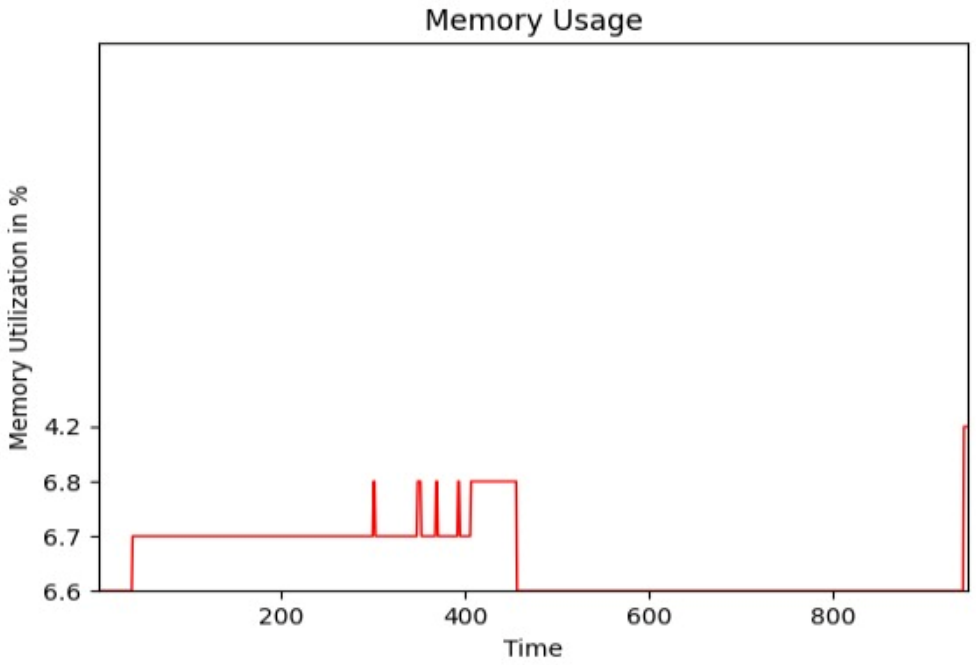
Spark uses a series of parameters, including the number of executors and the number of cores per executor. Throughout the entire period, CPU usage is high, but it spikes in the first seconds when we also notice a little increase in disk I/o speed.



Time has been scaled to adjust the curve in the graph

Memory

Memory utilized by spark is maximum 6.8%. It remained stagnant initial part of the gragh and later on it fluctuated towards the end of the graph.



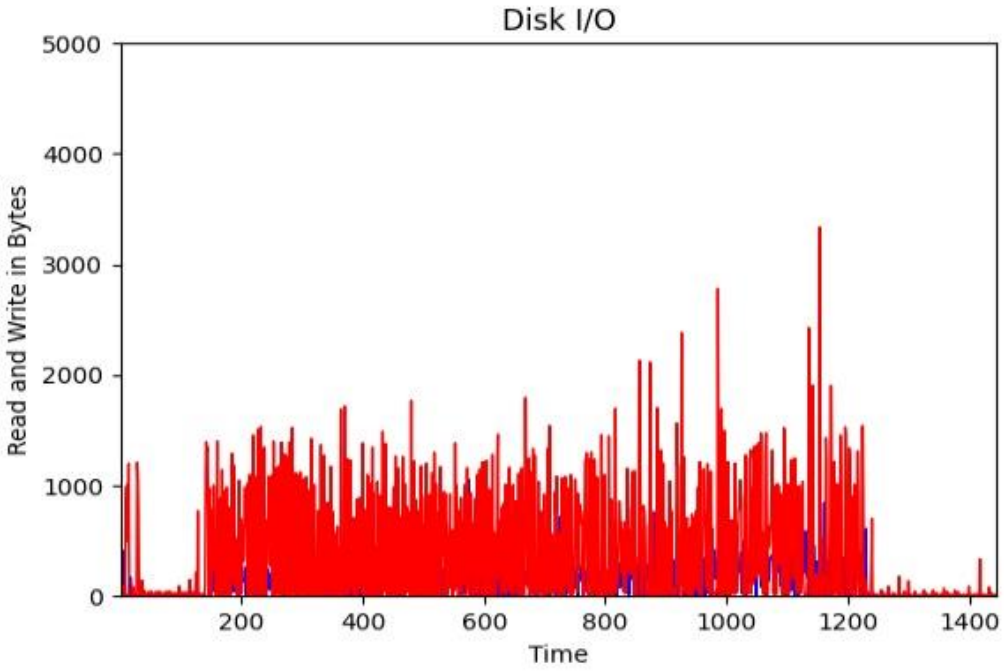
Time has been scaled to adjust the curve in the graph

Disk I/O

Except for the start ie 150 sec we observed closed throughput in graph. Spark uses disk except the first 150sec more frequently because of the initial map phase, and stores in the memory.

Blue = Read

Red = Write

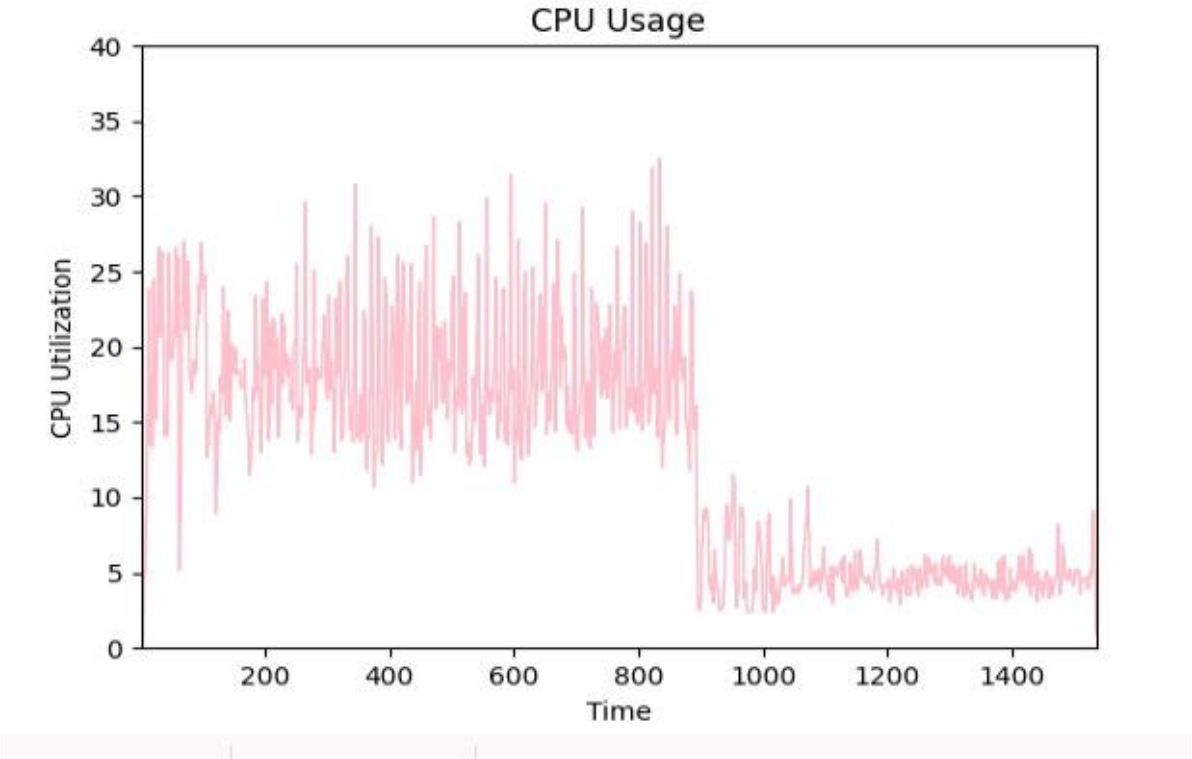


Time has been scaled to adjust the curve in the graph

Hadoop Sort for 4 Small Instance

CPU

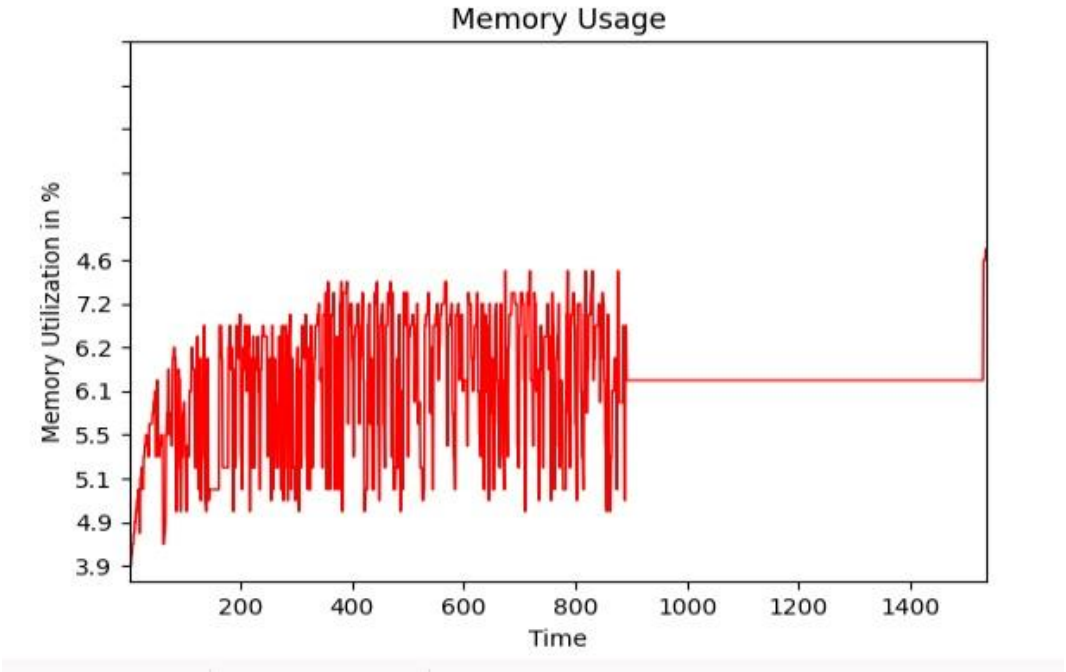
The CPU utilization is for all 4 nodes. The initial phase of the graph has fluctuation in CPU utilization because of sort and map phase. Most of the CPU is utilized by the datanode and minimum or negligible CPU is used by namenode.



Time has been scaled to adjust the curve in the graph

Memory

The memory utilization of the Hadoop 4 small instance is around 4.6%

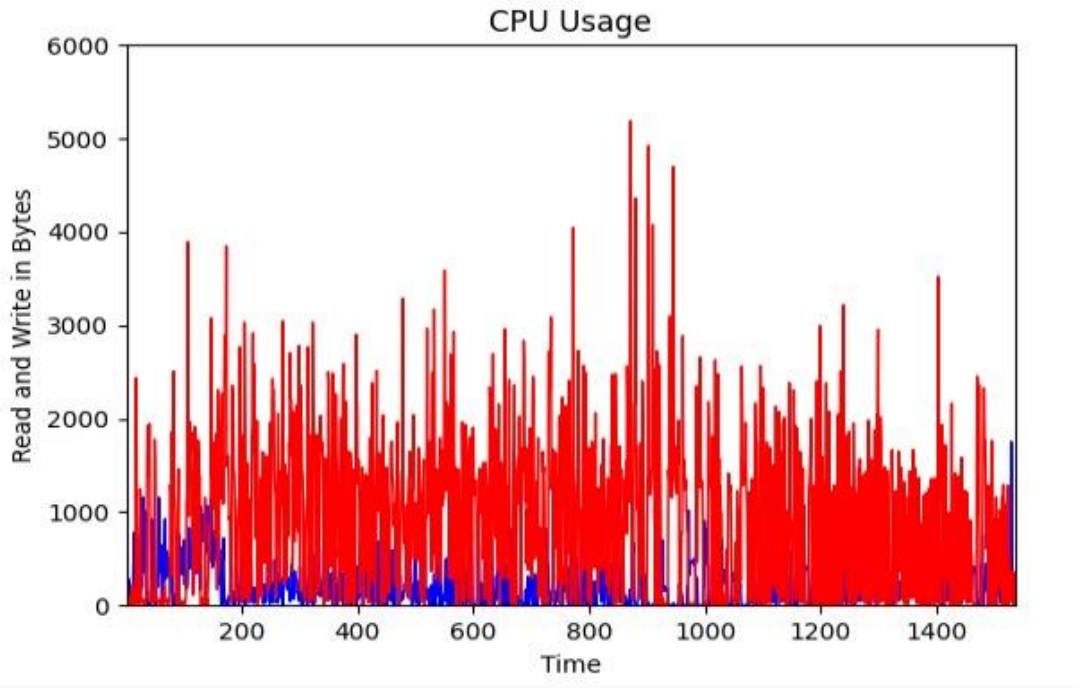


Time has been scaled to adjust the curve in the graph

Disk I/O

Blue = Read

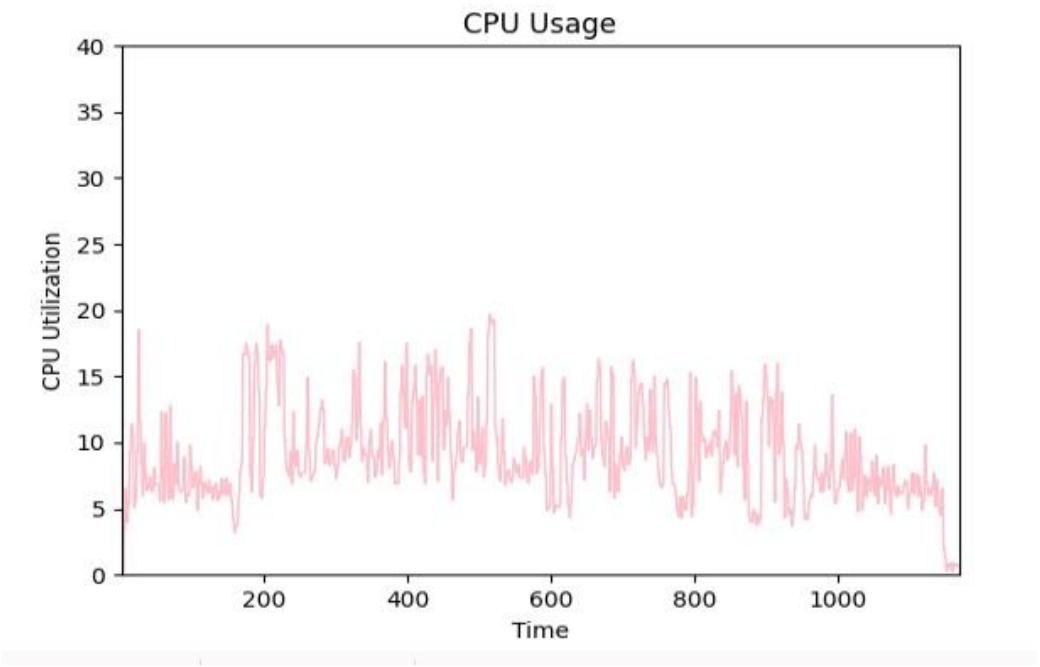
Red = Write



Time has been scaled to adjust the curve in the graph

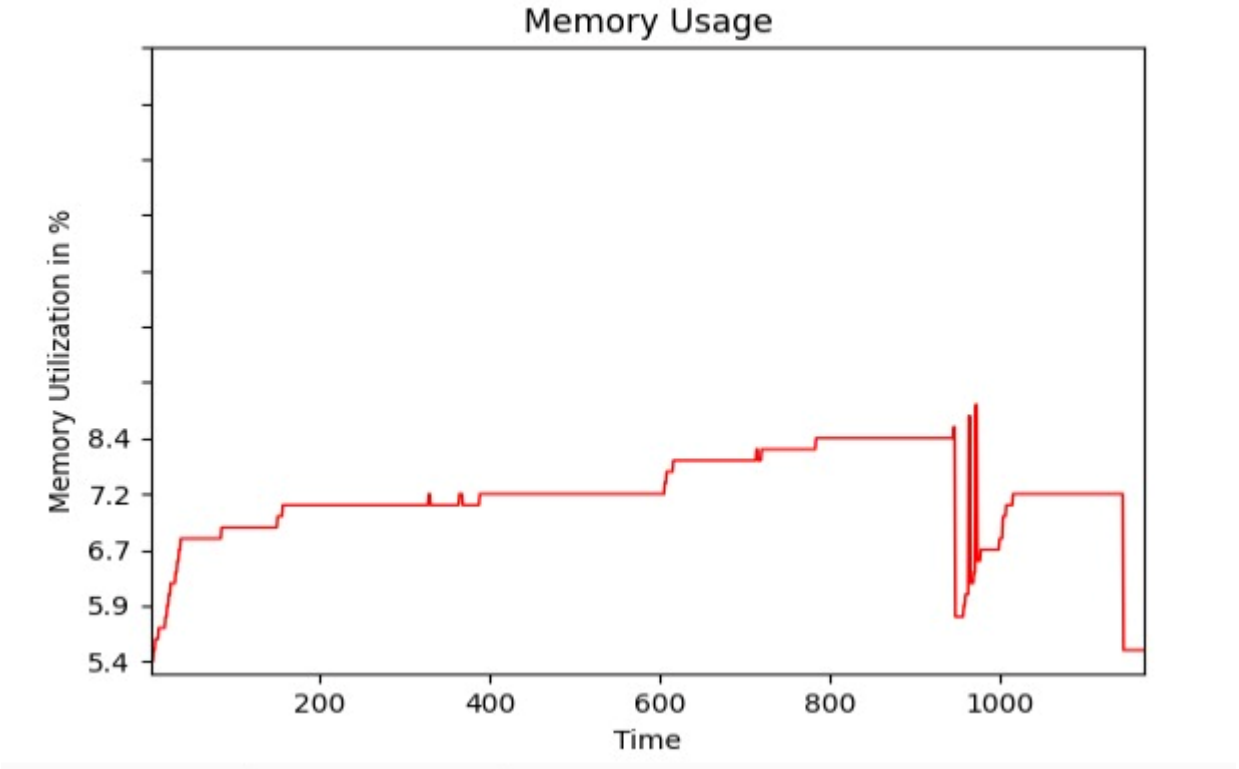
Spark Sort for 4 Small Instance

CPU



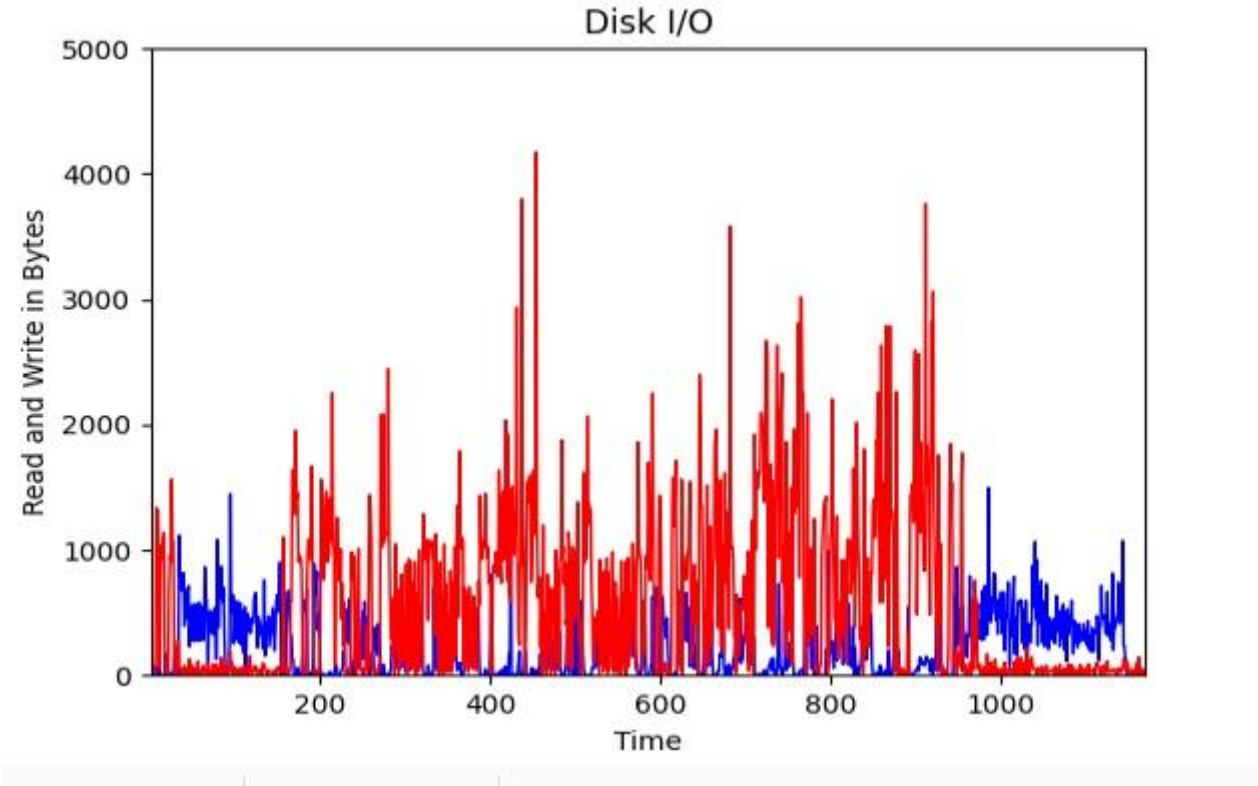
Time has been scaled to adjust the curve in the graph

Memory



Time has been scaled to adjust the curve in the graph

Disk I/O



Time has been scaled to adjust the curve in the graph