Assignment – 5 Report

NAME:- PRANAV MURALI
CWID:- A20555824

Hadoop

It is known that in hadoop, data is sorted during the sort/shuffle phase. We leveraged this idea to implement Hadoop Sort. In this sort, the input is the file generated by gensort. The mapper will produce the output of key value pairs where the first 10 bytes is the key and the rest is the value. After this the sort/shuffle phase happens. We have created a partition function which partitions the output of the map phase and gives it to reducers based on the first character of the key. It is known that gensort uses 128 ASCII characters. Hence, we divided this with the number of reducers to identify the set of characters for each partition. For example, if we have 2 reducers, keys with ascii value of first character between 0-63 will go to reducer 1 and 64-17 will go to the second reducer. The next phase which is reducer is just an identity operation where it writes input to the output file. Since we used a custom partitioning function that preserves order across multiple files, if we have multiple reducers, we just have to concatenate them to get final sorted output. Also, to decide the number of reducers we followed the industry standard of using 0.95 and 1.75 times the number of nodes in the cluster. With 0.95 all of the reduces can launch

immediately and start transfering map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.

Spark

The first step in Spark was to generate RDD of gensort output, by using the following command:
JavaRDD<String> textFile = sparkContext.textFile(input)
This way, every line in the text file turns into an RDD file that now physically sits in the datanode's memory.
Our goal is to create a RDD pair of key and value, therefore, we divided each line into 2 substrings of key-value pairs and finally we iteratively created another RDD. Different from the first set of RDD, this new RDD is joined pair of key-value.
The final step was to sort all the pair RDD by key, and output the results.

Trades offs:
In order to format Spark output, the default output uses brackets at the beginning and end of each line, we used map() method followed by .coalesce(1), to consolidate all data partitions into one file, which results in data movement from one partition to another and can impact the job running time.

Running variables:
For the large cluster we tried to optimize Spark performances by tuning the amount of memory allocated to each execueter and the number of cores.
The number of cores is essentially the number of concurrent tasks an executor can run. Based on reading online material the best optimal number for concurrent tasks per executor should be 5. In our case, we have 16 cores per node, 1 leaving for the OS, then 15/5 = 3 executors per node.
Memory per executor by the following calculation: 32GB/3 = ~10GB, after reducing the overhead we get 9GB memory per executor.

- For 1GB data record:
  executor-memory 7gb, executor-cores 4. We reduced the running time by 11.04 sec
- For 4GB data record:
  executor-memory 7gb, -executor-cores 5. We reduced the running time by 29.22 sec
- For 16GB data record:
  executor-memory 6gb executor-cores 2. We reduced the running time by 156.31 sec

We tried running the 16GB dataset with 4 and 5 cores per each executor and 7gb memory, it resulted in hdfs entering safe mode.

Q and As:

Please explain your results, and explain the difference in performance in **Table 1**?

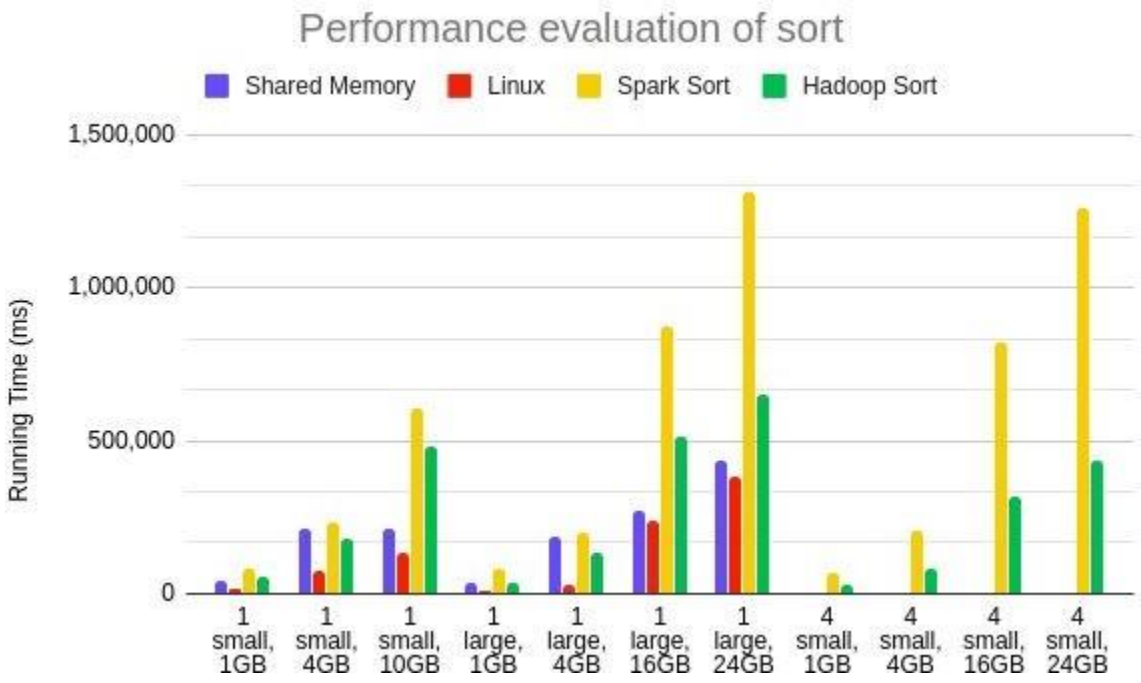
Table 1

Experiment (instance, dataset)	Shared Memory(sec)	Linux (sec)	Hadoop Sort (sec)	Hadoop Efficiency (w.r.t linux sort)	Spark Sort (sec)	Spark Efficiency (w.r.t linux sort)
1 small, 1GB	40.590	13.600	54.166	25.11%	81.170	16.75%
1 small, 4GB	214.870	73.050	178.673	40.88%	232.560	31.41%
1 small, 10GB	215.440	133.630	480.362	27.82%	605.600	22.07%
1 large, 1GB	38.800	7.420	37.335	19.87%	79.520	9.33%
1 large, 4GB	186.400	32.200	132.132	24.37%	202.340	15.91%
1 large, 16GB	269.246	238.830	511.943	46.65%	873.090	27.35%
1 large, 24GB	437.180	384.940	653.532	58.90%	1311.910	29.34%
4 small, 1GB	NA	NA	31.603	43.03%	68.050	19.99%
4 small, 4GB	NA	NA	79.289	92.13%	205.830	35.49%
4 small, 16GB	NA	NA	316.634	NA	821.680	NA
4 small, 24GB	NA	NA	433.795	88.74%	1258.630	30.58%

A. Based on our experiments and the table results, we are noticing that for 1 small instance and 1 large instance in almost every test case, Linux Sort is the fastest one and MySort is slightly slower than Linux but gives better results than Hadoop and Spark.
For data size 1GB and 4GB, Linux Sort and MySort perform all sort calculations in memory sort, which improve the speed of the i/o, computation and overall running time. For external memory sort we implemented multi-threading to run parts of the code in parallel. Also, the difference between 1 small instance and 1 large instance for those files record size is neglectable. Also, hadoop at 4 small nodes is faster than hadoop in 1 large instance even though the cumulative memory and cores are the same for both. This is because there are 4 mappers in hadoop doing map tasks simultaneously but only one mapper in 1 large instance, and this gets us the difference.
Based on what we learned and discussed in class, Spark is faster than Hadoop, mainly because Spark leverages the memory for calculation and storing and pulling data and real time, iterative data processing.
Our results reflected the opposite, meaning, Hadoop running time is less than Spark therefore Hadoop returns better performances.
There are number of factors that impact the two applications performances:

- Both software are designed to process huge amounts of data, their performances are not optimal with small files. Due to storage limitation, the largest file we ran was 24GB instead of 64GB. Very likely we would see different results for the 64GB.

- Spark uses a .coalesce() method to merge all output data partitions to one location on disk. It is a costly operation. Please see more details in the design trades off.



We also observe a trend for those test cases and when running Hadoop and Spark on 4 small instances: as we increase the data record size, Hadoop and Spark are performing better.
Linux Sort and MySort perform better with relatively smaller data sets while Hadoop and Spark specialized in processing massive amounts of data.
For this purpose, both Hadoop and Spark are designed in a centralized approach, where the master node communicates with the user and manages the jobs that need to be done and the workers (datanodes). The workers communicate solely with the master when they receive and finish a job. This enables running calculations in parallel and therefore decreasing the time to completion of large data sets. When running small data sets, although Spark and Hadoop use their parallel computing advantages, they have a lot of overhead for back and forth communication with the master node.

- How many threads, mappers, reducers, you used in each experiment?
  A. Although we used multiple threads, mappers, reducers and executors, the table here shows the best configuration we found for each dataset for each sorting technique in Table 2 below

Table 2

Experiment	Shared Memory	Linux	Hadoop Sort	Spark Sort
1 small.instance, 1GB dataset	In memory sort 1 thread 8 GB Memory	8 GB Memory 48 Threads	1 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
1 small.instance, 4GB dataset	In memory sort 1 thread 8 GB Memory	8 GB Memory 48 Threads	2 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
1 small.instance, 10GB dataset	48 Threads 8 GB Memory	48 Threads 8 GB Memory	2 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
1 large.instance, 1GB dataset	In memory sort 1 thread 8 GB Memory	8 GB Memory 48 Threads	2 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
1 large.instance, 4GB dataset	In memory sort 1 thread 8 GB Memory	8 GB Memory 48 Threads	2 Reducers	driver-memory: 4GB executor-memory: 7GB executor-core: 5
1 large.instance, 16GB dataset	48 Threads 8 GB Memory	48 Threads 8 GB Memory	2 Reducers	driver-memory: 4GB executor-memory: 6GB executor-core: 2
1 large.instance, 24GB dataset	48 Threads 8 GB Memory	48 Threads 8 GB Memory	1 Reducer	driver-memory: 4GB executor-memory: 2GB executor-core: 2
4 small.instances, 1GB dataset	NA	NA	7 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
4 small.instances, 4GB dataset	NA	NA	4 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
4 small.instances, 16GB dataset	NA	NA	7 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1
4 small.instances, 24GB dataset	NA	NA	7 Reducers	driver-memory: 4GB executor-memory: 2GB executor-core: 1

We used 48 threads for linux sort and Mysort which seemed to perform best at that number. For Hadoop since the block size is fixed, and is 64 MB in our case, we used (Dataset size in MB)/64MB number of mappers for each dataset. For example for 1 GB data, we used 1024/64 = 16 mappers. For reducers, we followed industry standards and majorly tested our hadoop sort program with reducers equal to 0.95 and 1.75 times the number of nodes. Hence for 1 large instance, we majorly tested for 1 and 2 reducers and for 4 mall instances we performed experiments using 4 and 7 reducers.

- How many times did you have to read and write the dataset for each experiment?
    A. For MySort and Linux sort when the dataset fits in the memory, we use internal sorting which implies we read and write the dataset once. However, when the dataset doesn't fit in memory, we use external sort and for this, we read twice and write twice.
    B. In the case of Hadoop, we read twice and write twice. First read is at the start of map tasks and the next read is by reducers during the shuffle phase. Also, first write is at the end of the map task and next write is after the reducers task.
    C. Spark stores intermediate results in the memory instead of on the disk. For scenarios where the file size is larger than the memory, Spark will store the intermediate results on the disk.
- What speedup and efficiency (in comparison to linux sort) did you achieve?
    A. Please find this in the Table 1

- What conclusions can you draw? Which seems to be best at 1 node scale (1 large.instance)? Is there a difference between 1 small.instance and 1 large.instance? How about 4 nodes (4 small.instance)?
    A. At one node scale 1 large instance outperforms 1 small instance and the reason is that 1 large instance is more powerful in terms of memory and cores. Also, this is more evident with the increase in the size of the data. For example for a 4GB dataset on 1 small instance, hadoop sort took 178.673 seconds but the same sort on 1 large instance took 132.63 seconds. The same for other sorts too. At 4 nodes, hadoop sort and spark sort perform better for the same dataset. This is because their parallel work design, meaning more workers are available for the master node to assign tasks in parallel . Also, these 4 nodes do not hold for linux and shared memory sort since they run on a single node. Although it seems like linux and MySort run faster than hadoop and spark, and this is because the dataset size is small and hence overhead costs like implementation in Java and Scala(for hadoop and spark) vs C (for linux and MySort), starting the cluster costs, communication between nodes over network costs dominate for smaller datasets. Once we go to higher datasets, hadoop and spark will outperform these. Also, it is much costlier to build a single node with lots of cores (required by linux and MySort) than combine multiple smaller nodes(used by Hadoop and Spark) to perform sort.
    As mentioned in the code design, for 1 large vm we change Spark configurations to achieve optimized solutions, we noted an improvement in running time, however, the 4 small instances cluster perform better. Hence, hadoop and spark will be better.

- What speedup do you achieve with strong scaling between 1 to 4 nodes? What speedup do you achieve with weak scaling between 1 to 4 nodes?

  A. On strong scaling from 4 GB on 1 small to 4 GB on 1 large, hadoop performed quite differently. It is faster on 1 large instance where it took 79.289 secs vs 1 small where it took 178 seconds. On analyzing for other datasets as well, hadoop's performance seemed to increase at least 2 times

  For 4GB on 1 small instance, hadoop took 178 seconds and for 16 GB on 4 small instances, hadoop took 316 seconds which implies weak scaling indeed improves the performance of hadoop sort by at least 4 times.

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

  A. For smaller datasets, the overhead costs in hadoop are masking the benefits of using hadoop over shared memory sort. At smaller datasets like 24 GB on 1 large instance, hadoop sort took 653.532 seconds whereas shared memory took 437.18 seconds. This implies hadoop should have at least 1.5 times the current number of instances (assuming the overhead costs remain the same). This means Hadoop should either have 2 large instances to achieve the same performance. However, with 4 small instances, the difference between time taken by hadoop and shared memory sort slowly decreases from 1 GB to 24 GB and even though at 24 GB shared memory takes less time, as we increase the dataset, hadoop will outperform shared memory as evident by this change.

- 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?

  A. Shared memory sort speed on one node using in memory sort or multi-threading for external sort comes into play, therefore we observe that shared memory sort outperforms Spark on small or large 1 instances.

  Of course we need to take in account JVM starting time, and file size.

  The file size will determine if shared memory sort and Hadoop will use the memory to perform the sort or external. When both algorithms are used in memory search the difference in running time between the two can be double at max, when they perform external sort then Spark running time spikes in comparison to shared memory sort. When running a 10GB file, Spark running time is almost 3 times more than shared memory sort for small instances. For the large 1 instance for 16GB size file, spark running time is almost 4 times more than shared memory. Based on those experiments we'll need around 3-4 more VM per instance. The largest data size we ran is 24GB, our assumption that increasing the data size,Spark will output Shared Memory sort or the results will be close.

- Can you draw any conclusions on the performance of the bare-metal instance performance from HW5 compared to the performance of your sort on a large instance through virtualization?
    A. Yes. The shared memory sort on the bare metal instance did not have any overhead costs which resulted in much lesser sort times. On an average there is a 25% increase in sort time and this is the virtualization cost.

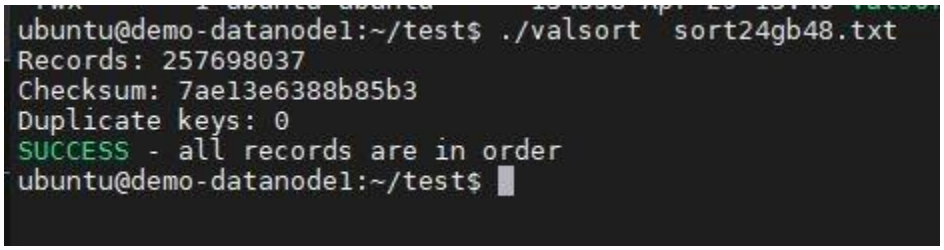
- Can you predict which would be best if you had 100 small.instances? How about 1000? Compare your results with those from the Sort Benchmark (http://sortbenchmark.org), specifically the winners in 2013 and 2014 who used Hadoop and Spark.
    A. Both hadoop and spark are very good for sorting applications. Spark is expensive compared to hadoop sort since in Spark, we need more RAM which is costlier than Hadoop solution which uses Hard Disk. Hence, at 100 small instances, spark sort is best since the performance is much better than Hadoop and this shadows the costs. However, as we increase the number of nodes, the cost of sorting will be much higher for spark compared to Hadoop. Also, as the number of nodes increase, the number of nodes that may fail will also increase. Replacing nodes in a spark cluster is expensive (in terms of hardware) compared to a node in hadoop cluster. Also, since sorting is not an iterative application, the performance will not differ greatly between hadoop sort and spark sort. Hence, at 1000 instances, hadoop is a better solution.

Example on how we performed valcheck on multiple reducer outputs to test our hadoop sort:

As seen below, first we got all the files from hdfs's output directory and put them in the local directory. Then we concatenated all the files and then ran valsort on the concatenated file which passed the valsort check. This proves that our Hadoop sort algorithm works perfectly.

```
-rw-r--r-- 1 ubuntu ubuntu 361816500 Apr 30 04:22 part-r-00002
-rw-r--r-- 1 ubuntu ubuntu 350258400 Apr 30 04:22 part-r-00003
-rwxr-xr-x 1 ubuntu ubuntu    134558 Apr 30 04:21 valsort*
ubuntu@demo-datanode1:~/valcheck$ cp part-r-00001 valcheck
ubuntu@demo-datanode1:~/valcheck$
ubuntu@demo-datanode1:~/valcheck$ ll
total 1401916
drwxrwxrwx 2 ubuntu ubuntu      4096 Apr 30 04:22 ./
drwxrwxrwx 8 ubuntu ubuntu      4096 Apr 30 04:20 ../
-rw-r--r-- 1 ubuntu ubuntu         0 Apr 30 04:22 _SUCCESS
-rw-r--r-- 1 ubuntu ubuntu         0 Apr 30 04:22 part-r-00000
-rw-r--r-- 1 ubuntu ubuntu 361666900 Apr 30 04:22 part-r-00001
-rw-r--r-- 1 ubuntu ubuntu 361816500 Apr 30 04:22 part-r-00002
-rw-r--r-- 1 ubuntu ubuntu 350258400 Apr 30 04:22 part-r-00003
-rw-r--r-- 1 ubuntu ubuntu 361666900 Apr 30 04:22 valcheck
-rwxr-xr-x 1 ubuntu ubuntu    134558 Apr 30 04:21 valsort*
ubuntu@demo-datanode1:~/valcheck$ rm part-r-00001
ubuntu@demo-datanode1:~/valcheck$
ubuntu@demo-datanode1:~/valcheck$ cat part-r-00002 >> valcheck
ubuntu@demo-datanode1:~/valcheck$ rm part-r-00002
ubuntu@demo-datanode1:~/valcheck$ cat part-r-00003 >> valcheck
ubuntu@demo-datanode1:~/valcheck$ ll
total 1390772
drwxrwxrwx 2 ubuntu ubuntu       4096 Apr 30 04:23 ./
drwxrwxrwx 8 ubuntu ubuntu       4096 Apr 30 04:20 ../
-rw-r--r-- 1 ubuntu ubuntu          0 Apr 30 04:22 _SUCCESS
-rw-r--r-- 1 ubuntu ubuntu          0 Apr 30 04:22 part-r-00000
-rw-r--r-- 1 ubuntu ubuntu  350258400 Apr 30 04:22 part-r-00003
-rw-r--r-- 1 ubuntu ubuntu 1073741800 Apr 30 04:23 valcheck
-rwxr-xr-x 1 ubuntu ubuntu     134558 Apr 30 04:21 valsort*
ubuntu@demo-datanode1:~/valcheck$ rm part-r-00003
ubuntu@demo-datanode1:~/valcheck$ ./valsort valcheck
Records: 10737418
Checksum: 51eaa7ee3566d0
Duplicate keys: 0
SUCCESS - all records are in order
ubuntu@demo-datanode1:~/valcheck$
```

Valsort check for 24 GB data on Hadoop sort:



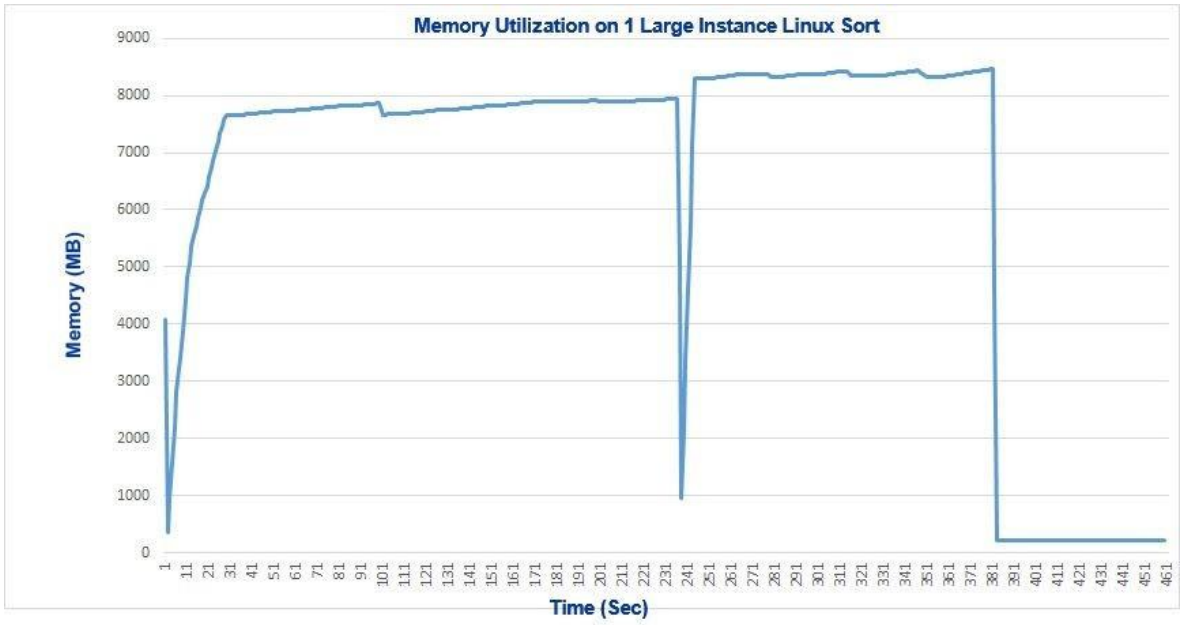
For spark sort, we attached valsort check in sparksort24GB.log

Graphs for the highest dataset in our experiments (24GB)

Linux Sort

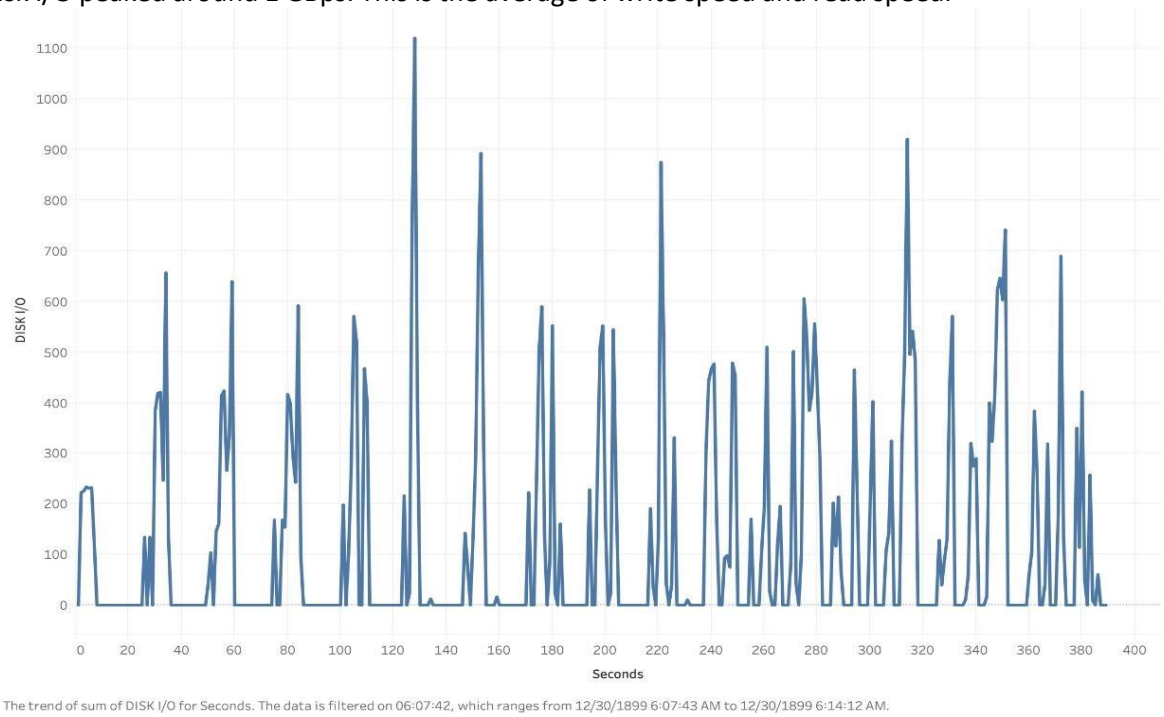
Memory Used:

The memory usage seemed to be the same compared to bare metal instance and virtualization doesn't seem to affect memory usage and this is expected.



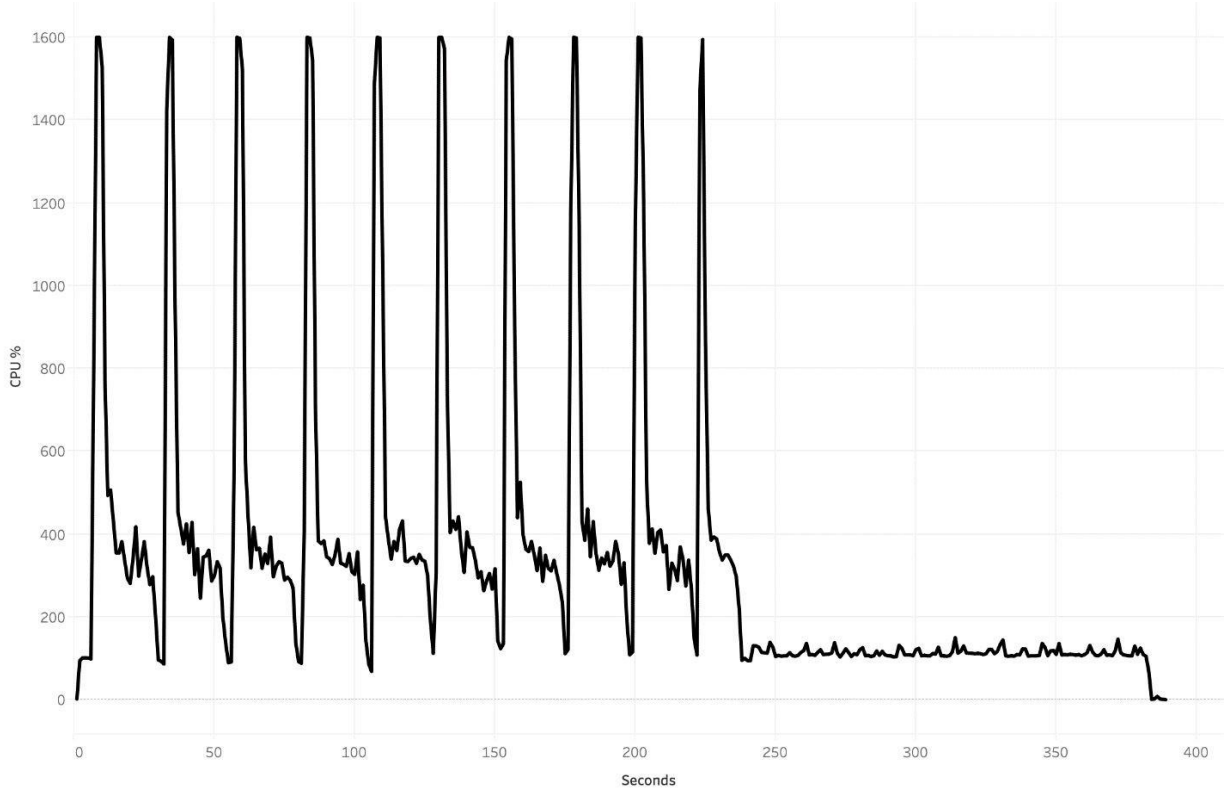
Disk I/O in MBPS:

Disk I/O peaked around 1 GBps. This is the average of write speed and read speed.



CPU Utilization:

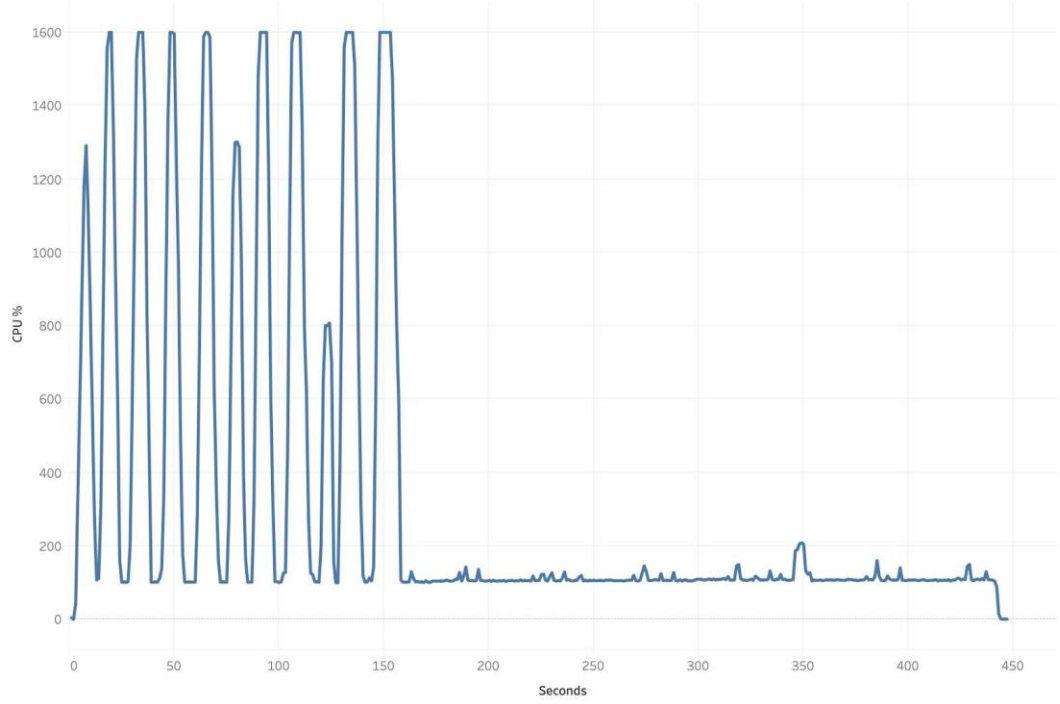
As expected, CPU % is more than 100% since multiple cores are used for this sort.



The trend of sum of CPU % for Seconds. The data is filtered on 06:07:42, which ranges from 12/30/1899 6:07:43 AM to 12/30/1899 6:14:12 AM.

Shared Memory Sort

CPU Utilization:

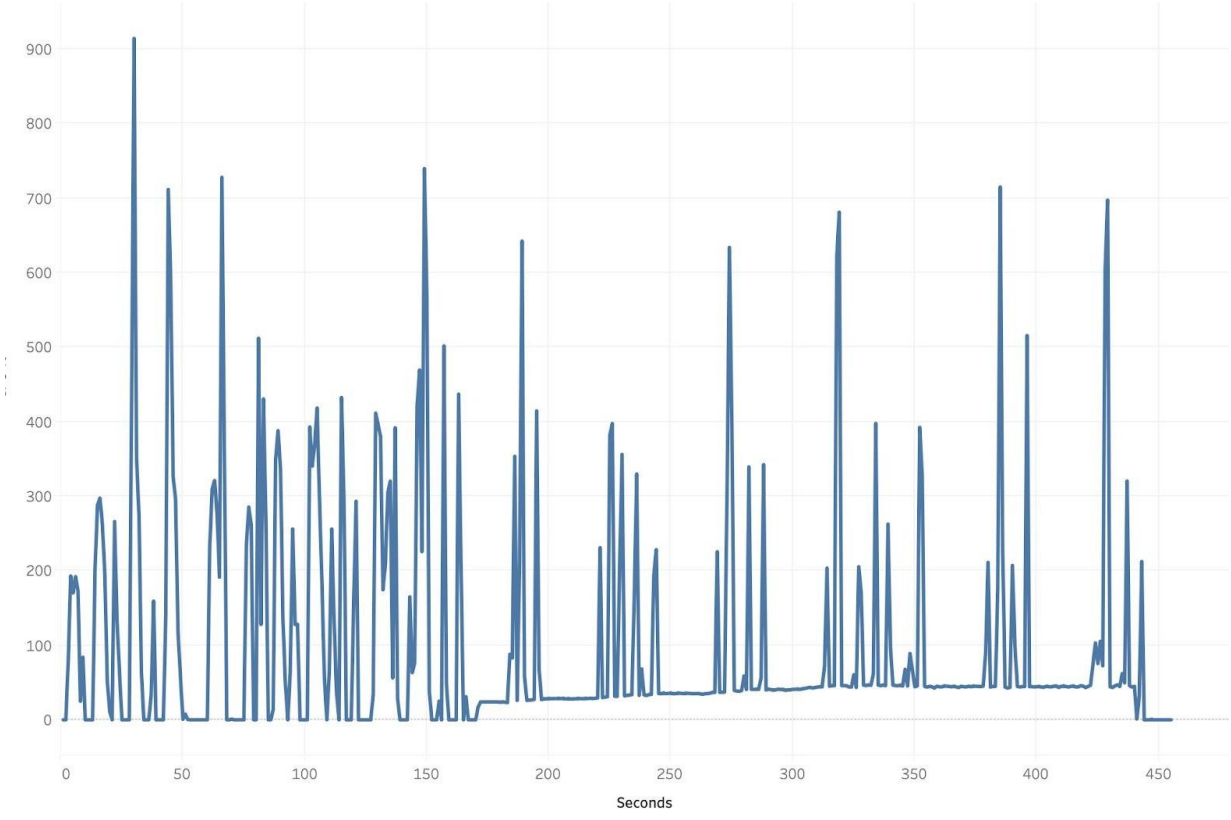


The trend of sum of CPU % for Seconds. The data is filtered on 07:01:27, which ranges from 12/30/1899 7:01:28 AM to 12/30/1899 7:08:54 AM.

DISK I/O:

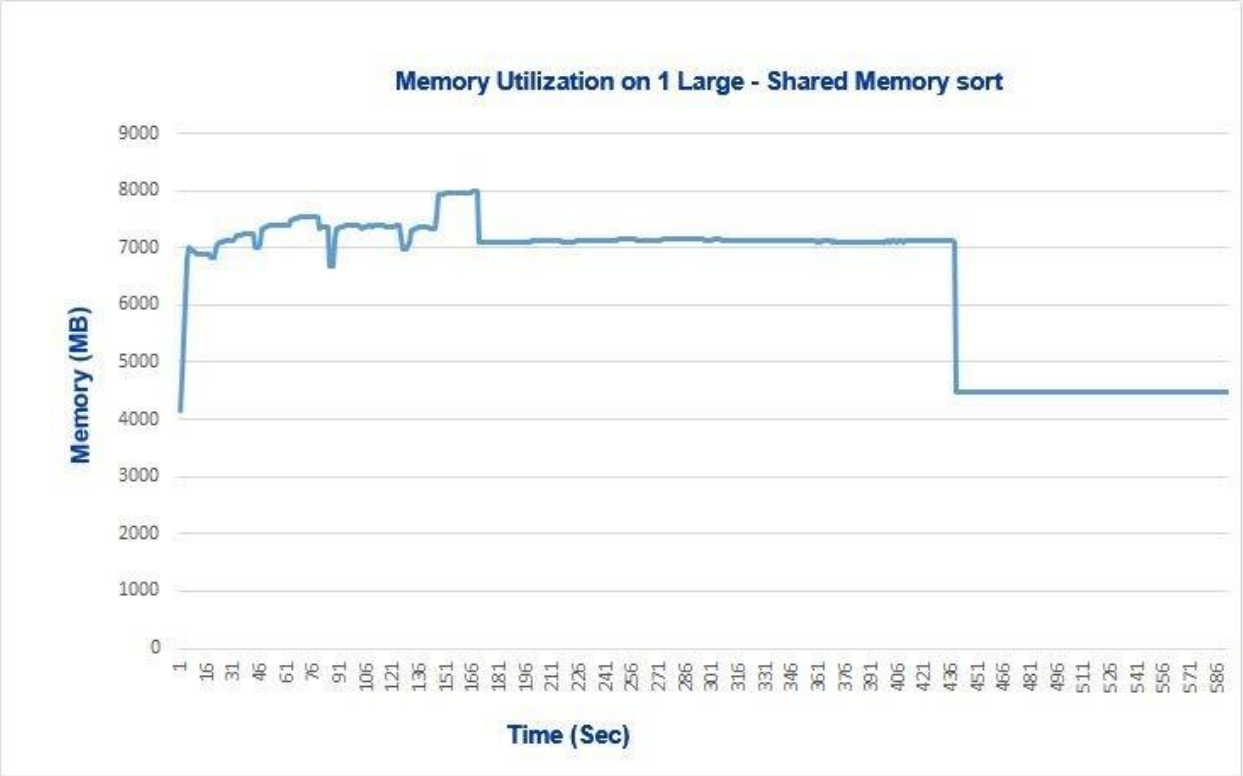
The disk I/O in myshared is not as fast compared to linux sort since it does not use multithreading in the initial read phase of the file.

Disk I/O(MBps)



Memory Usage:

Since we restrict the memory usage to 8 GB in shared memory sort, we can see in the graph that memory usage never exceeds 8GB. Also memory usage is high at the beginning where multiple threads work on sorting at the same time. However, later during the merging phase, memory usage is low as expected.

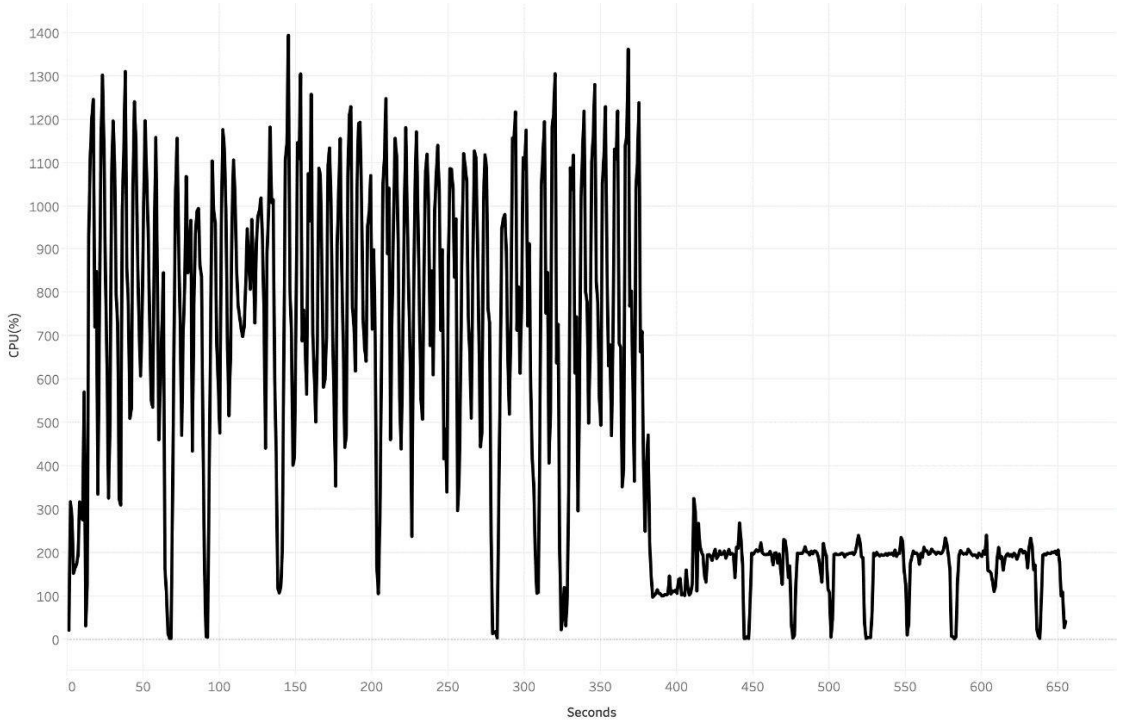


Hadoop Sort on 1 Large Instance

CPU Utilization:

The cpu utilization is more at the beginning where map, sort and shuffle phase happen. But during the reduce stage, since it is the identity operation of writing data to disk, we see that not much cpu is used.

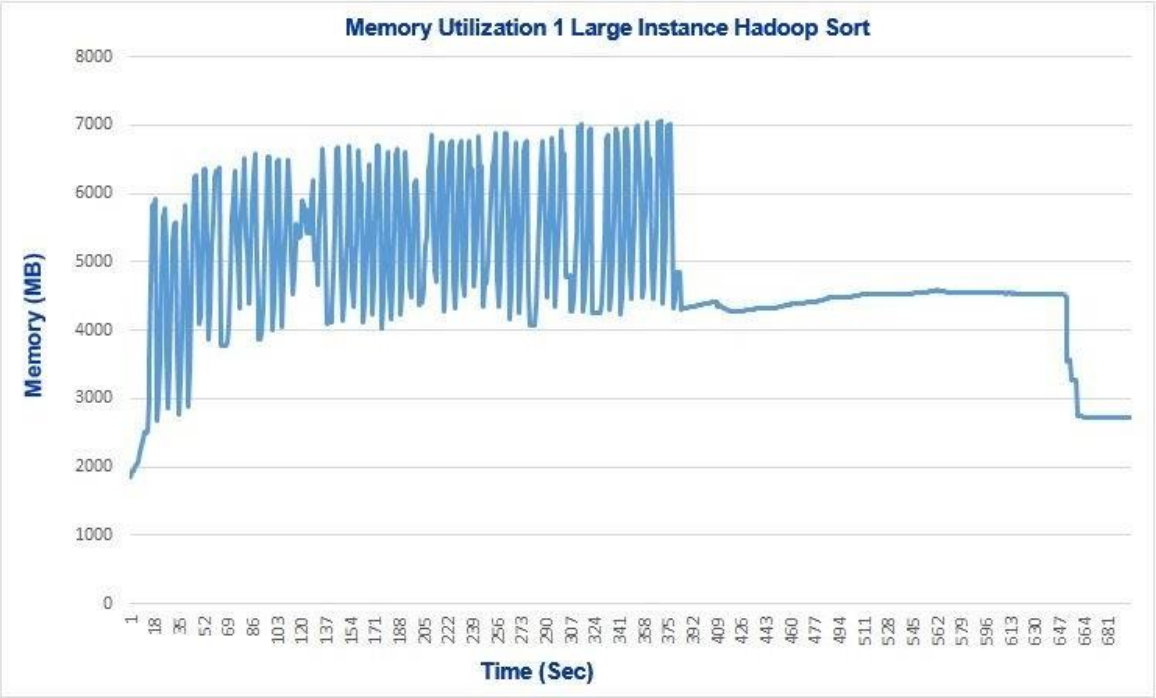
Also, the reduce phase is much shorter compared to map phase.



The trend of sum of CPU(%) for Seconds. The data is filtered on 07:26:43, which ranges from 12/30/1899 7:26:44 AM to 12/30/1899 7:37:38 AM.

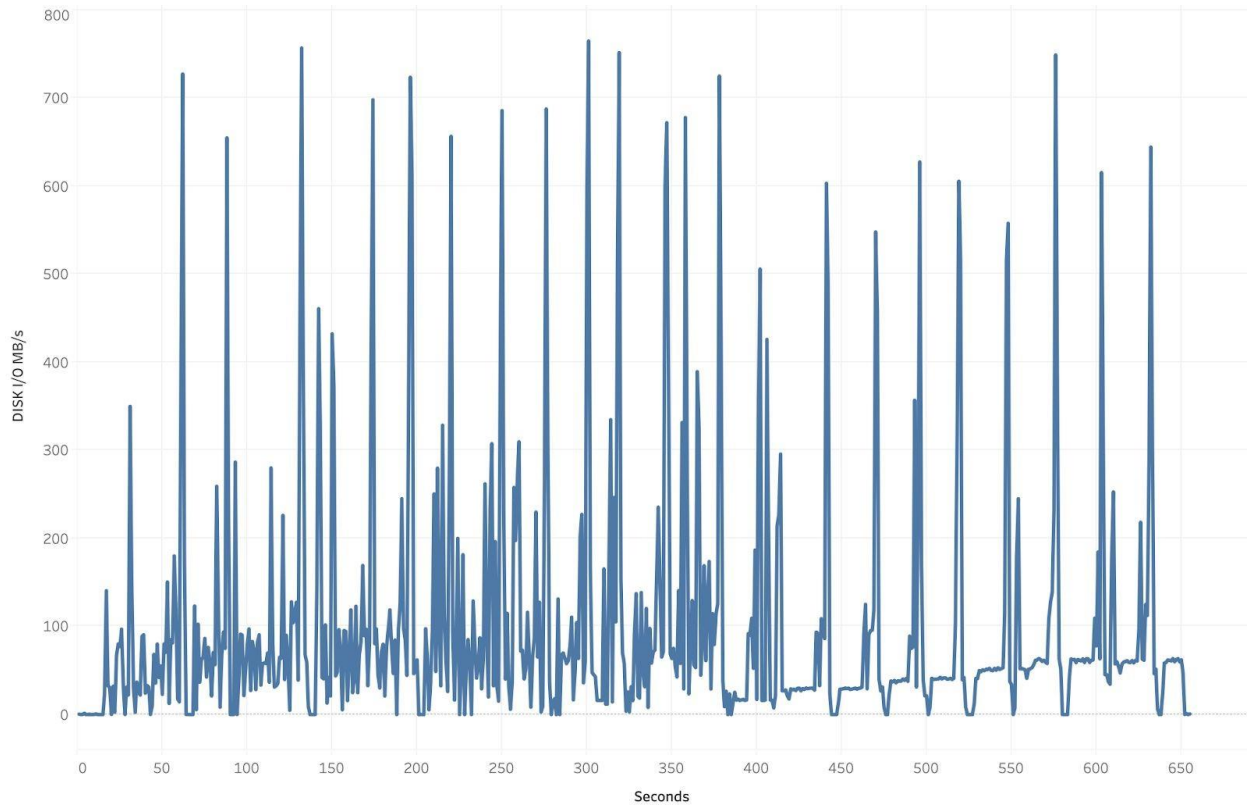
Memory:

We used sar -r 1 command to get the information about memory used. However, sar -r 1 command gives the aggregate value including the cache. We have subtracted the cache output to get the correct memory usage at every second. Hadoop did not use more than 8 GB on 1 large instance at any point of time. Also, the memory utilization is more at the beginning phase where sort/shuffle happens but at reduce stage, the memory utilization is not so high.



DISK I/O:

The disk I/O happens throughout the map reduce operation in hadoop since during map phase, data is read and mappers' output is written to disk which is transferred to reduce during shuffle phase and also, during reduce phase since data is written to disk.

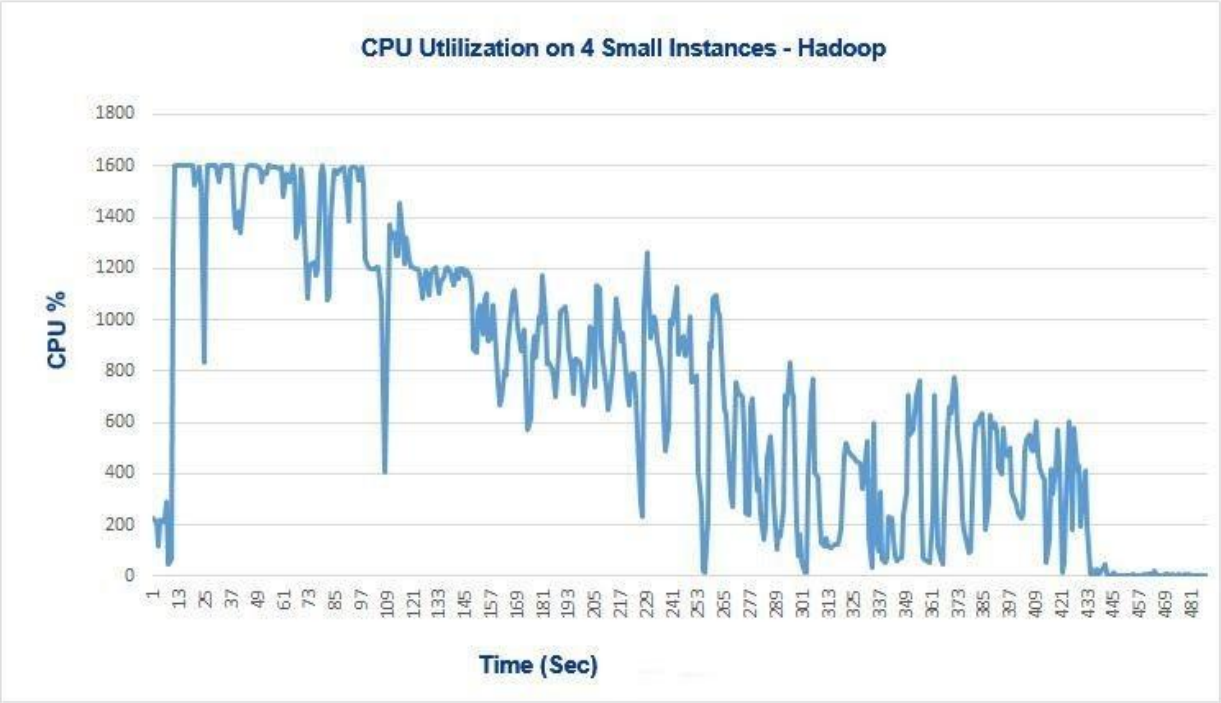


The trend of sum of DISK I/O MB/s for Seconds. The data is filtered on 07:26:43, which ranges from 12/30/1899 7:26:44 AM to 12/30/1899 7:37:38 AM.

Hadoop Sort on 4 Small Instances

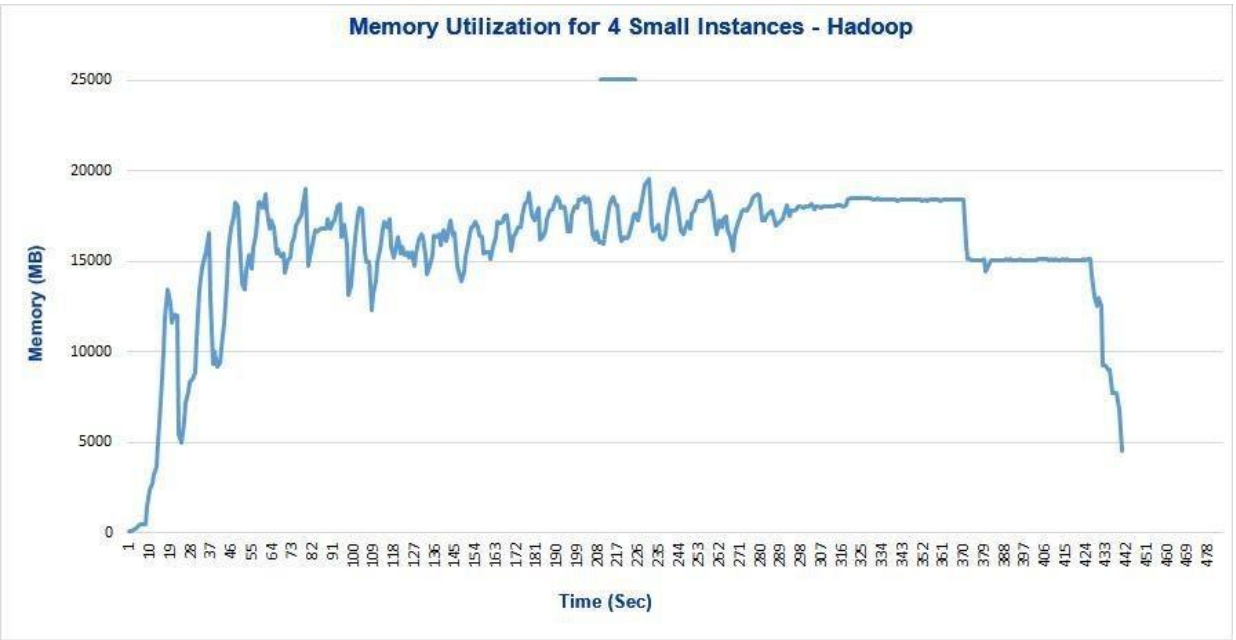
CPU Utilization:

The cpu utilization seen here is the aggregate value of cpu utilization of all nodes (data+name node). The cpu used by the name node is negligible but the data nodes use high amounts of cpu. The cpu utilization is more initially since map and sort phase happen in the beginning. Later in the reduce stage, the utilization is low since there is only data writing happening in reduce phase. Also, the reduce phase is much shorter compared to map phase especially when 4 data nodes are used compared to 1 node since multiple write operations are happening at once.



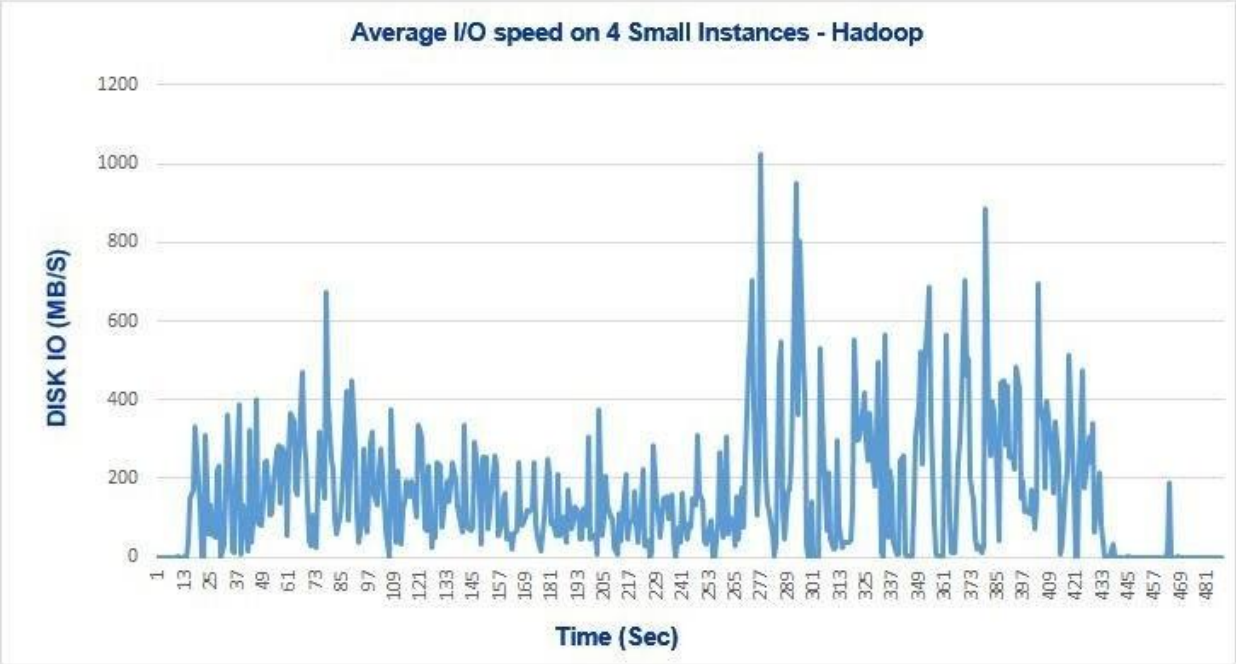
Memory:

The memory utilization in hadoop 4 large instances was around 8 gb. However, since sar command gives aggregate usage output (that includes cache), the figure below shows the usage to be higher but when subtracted the cache, we could see memory usage of around 8GB. Please note this.



DISK I/O:

During the map phase, as soon as a master node gives a worker a map phase, it gets the data from the input file, performs map operation and then moves on to the next block of the file. But during the reduce phase, only data write to file operation is happening and hence we see the spike in disk I/O speed during the reduce phase.

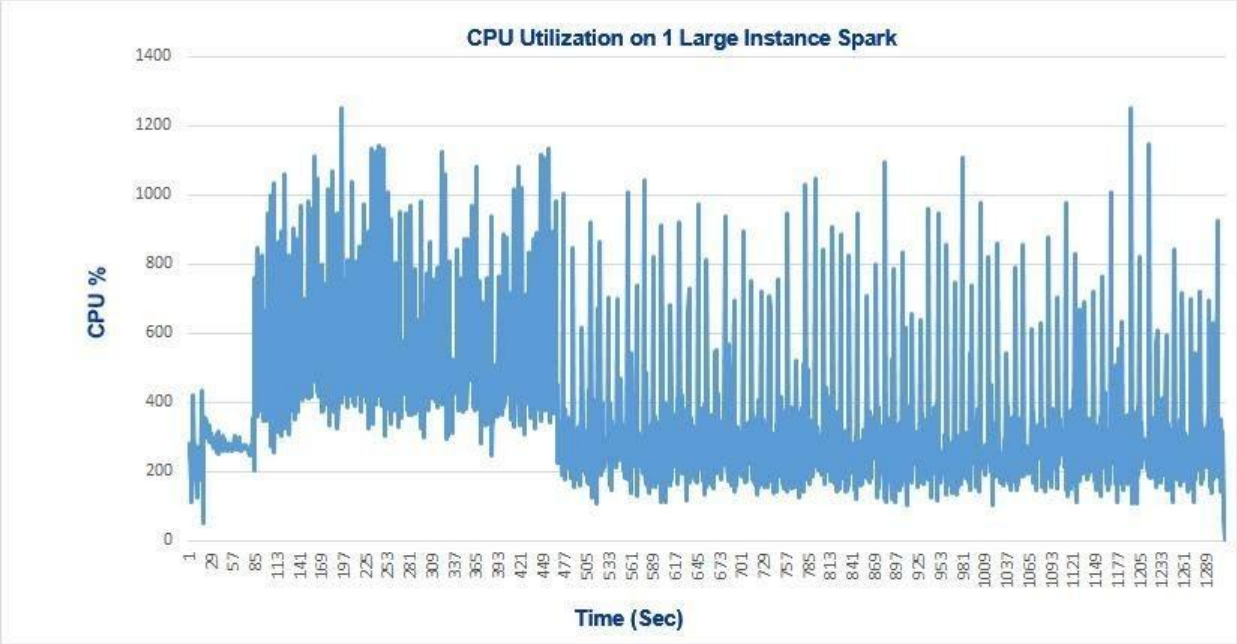


Spark - 1 Large Instance

CPU Utilization:

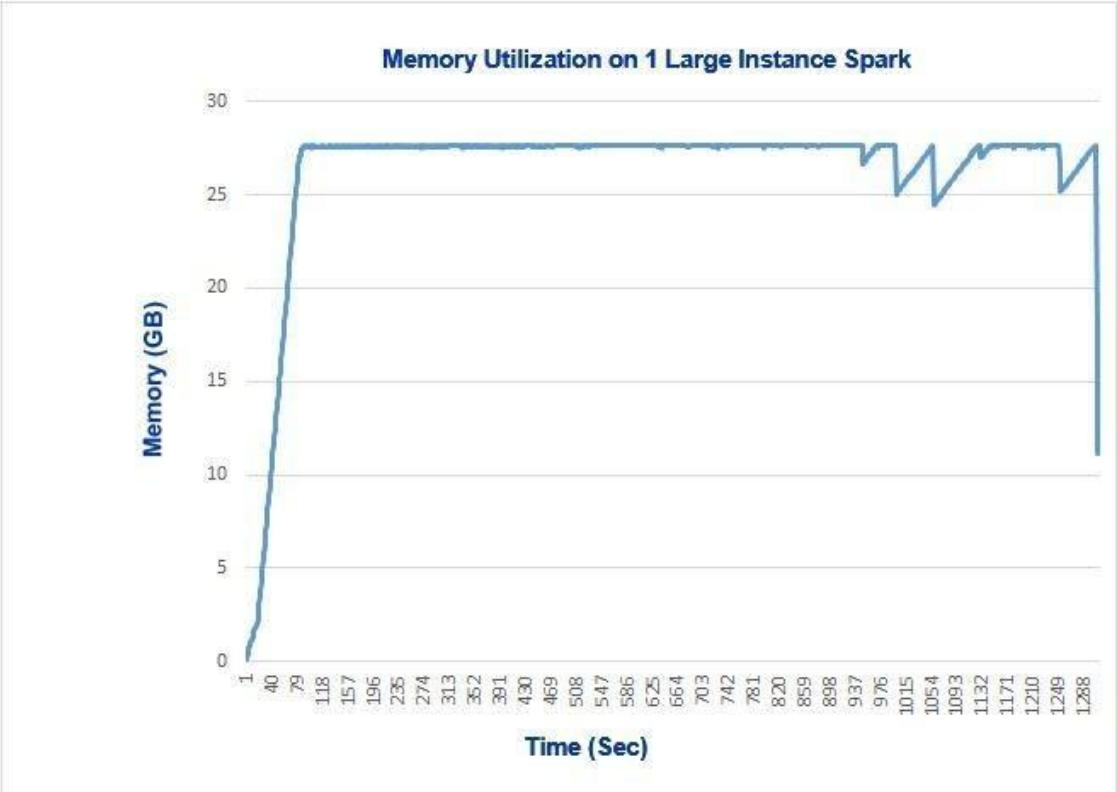
CPU utilization is high throughout the time, especially in the first 447 sec, where we also observe a slight increment in the disk i/o speed.

Spark uses the number of cores according to set variables such as numbers of cores per executor and the number of executors.



Memory:

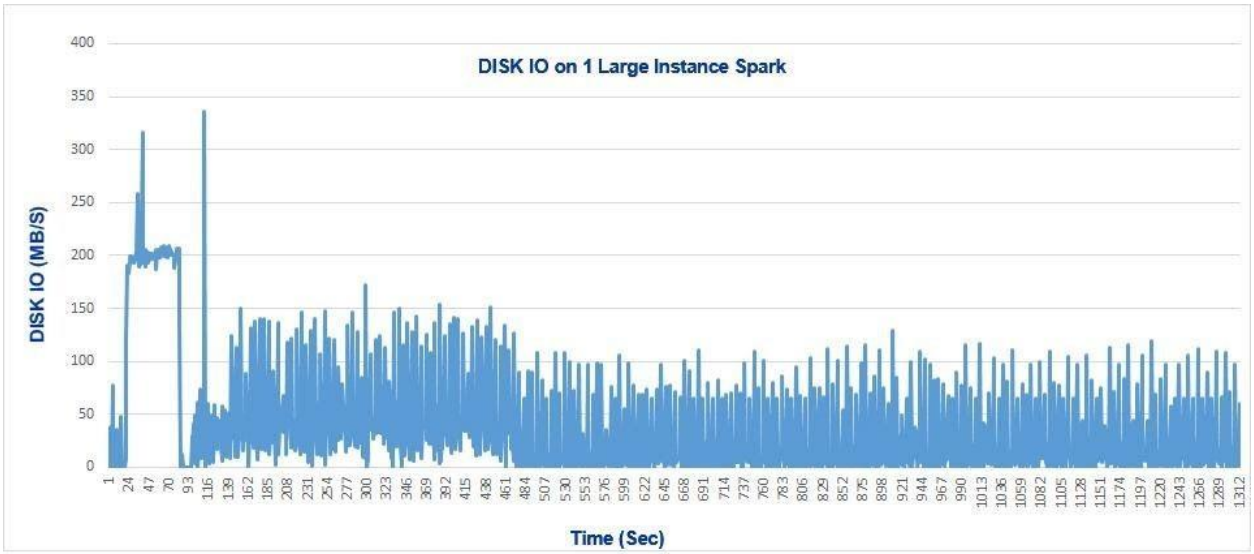
The large instance memory is larger than the file size, spark was able to perform the calculation and storing the intermediate data memory, it can be understood that it uses around 25 GB since the file size is 24 GB.



DISK I/O:

Except for the first ~116 sec, we observe that spark i/o to disk at a relatively close rate throughout the job's running time, every couple of seconds we notice a small spike in the number of MB/s.

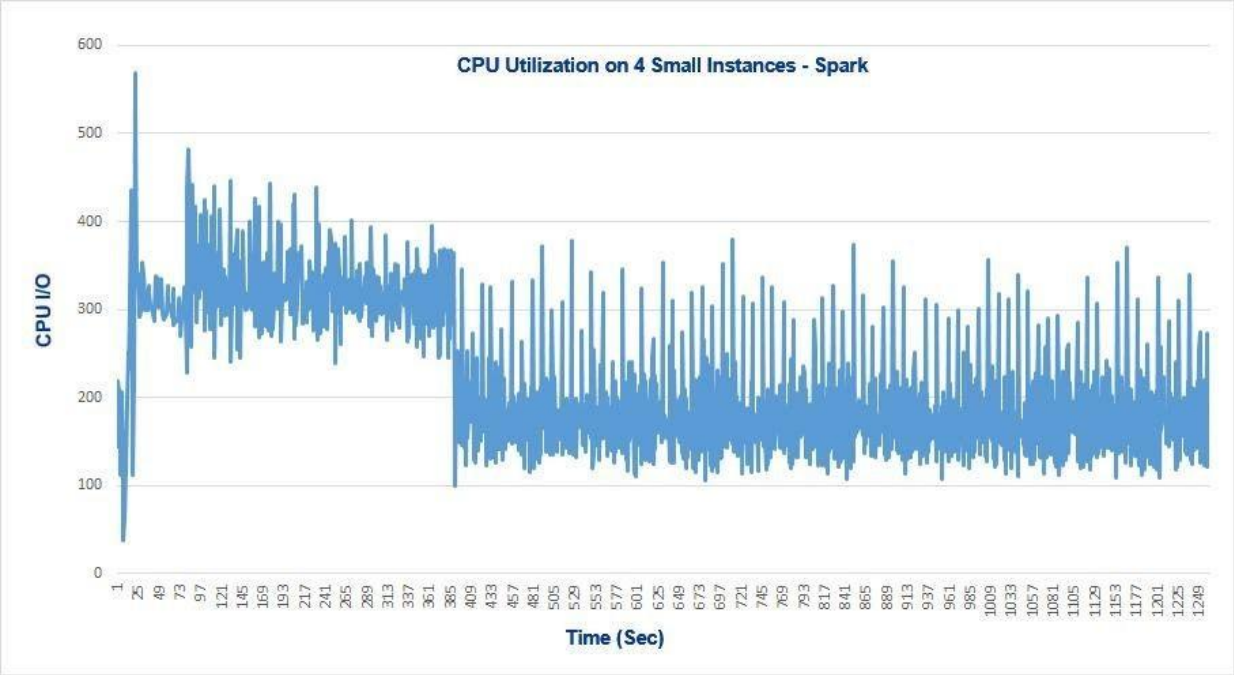
Between 116-461 sec spark uses disk i/o more frequently due to the initial map phase, where it takes the data and transforms it to RDD and stores it in memory.



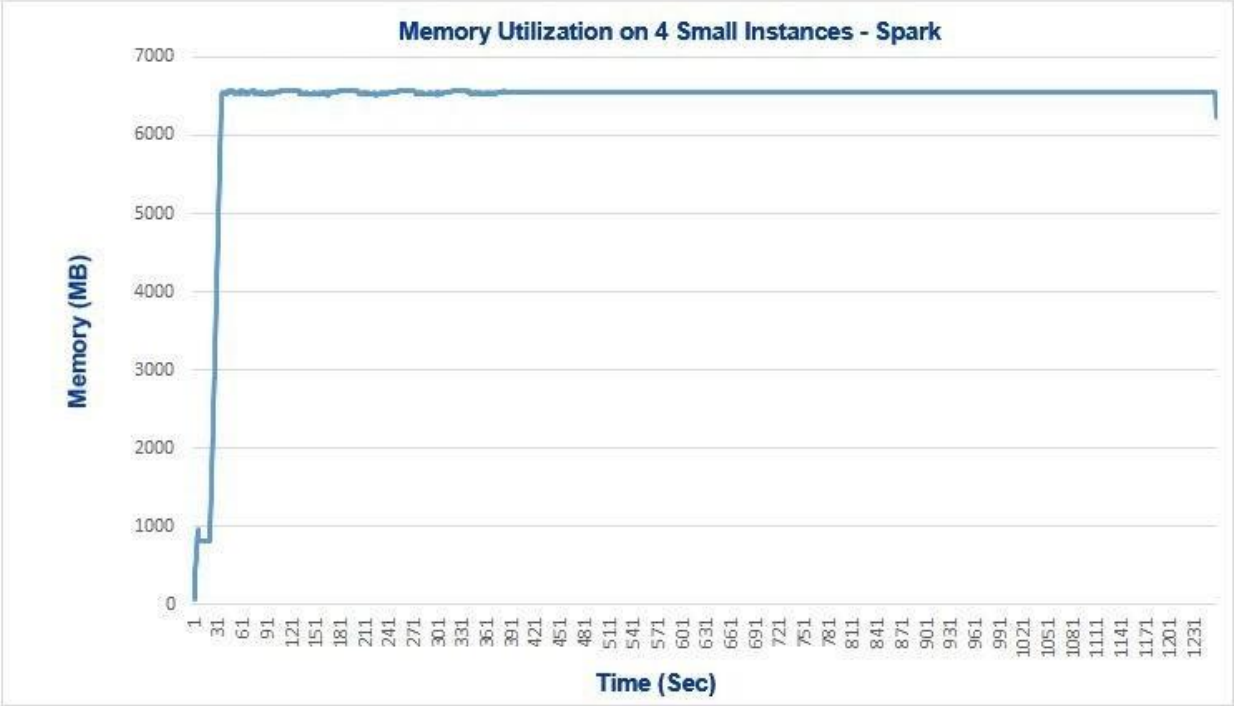
Spark – 4 Small Instances

CPU Utilization:

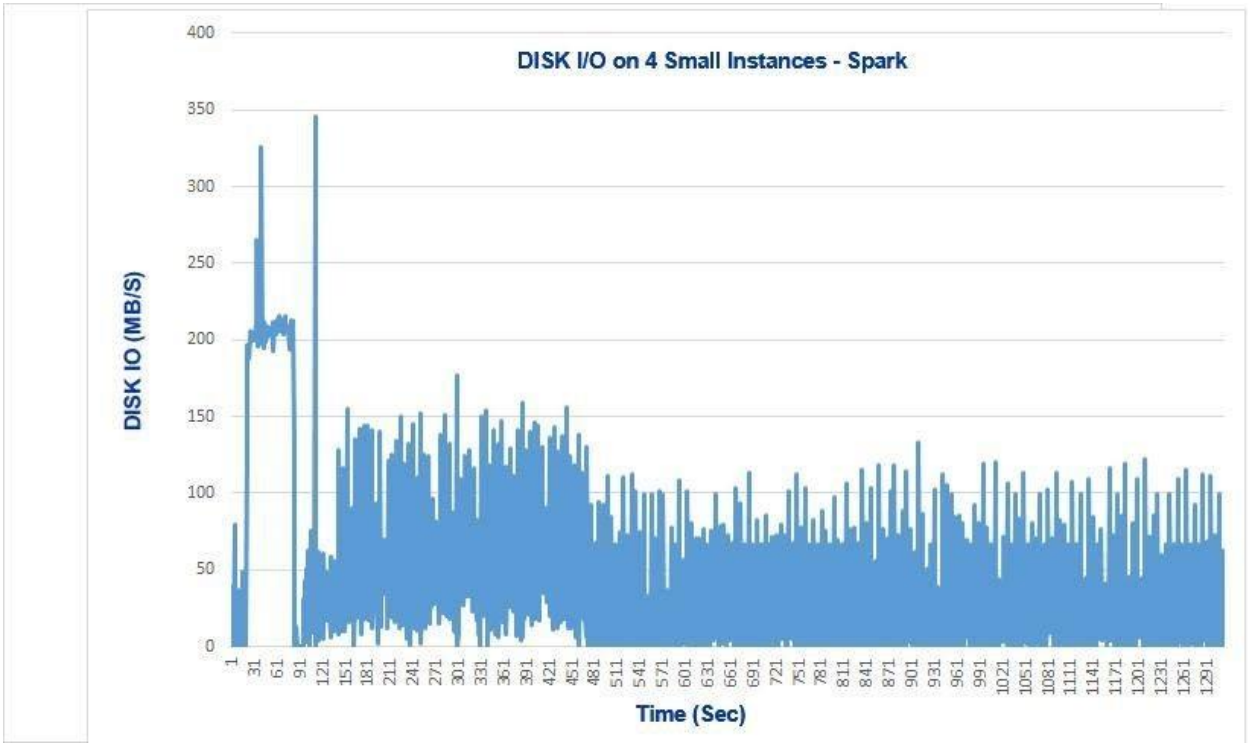
The aggregate cpu utilization for spark is as seen below. It is less compared to hadoop. Same as we observed in the graph for cpu utilization for 1 large instance, spark uses the cpu more at the beginning of the job, when it transforms data into RDD.



Memory:


Memory Utilization on 4 Small Instances - Spark
Memory (MB)
7000
6000
5000
4000
3000
2000
1000
0
1
31
61
91
121
151
181
211
241
271
301
331
361
391
421
451
481
511
541
571
601
631
661
691
721
751
781
811
841
871
901
931
961
991
1021
1051
1081
1111
1141
1171
1201
1231
Time (Sec)

DISK I/O:


DISK I/O on 4 Small Instances - Spark
DISK IO (MB/S)
400
350
300
250
200
150
100
50
0
1
31
61
91
121
151
181
211
241
271
301
331
361
391
421
451
481
511
541
571
601
631
661
691
721
751
781
811
841
871
901
931
961
991
1021
1051
1081
1111
1141
1171
1201
1231
1261
1291
Time (Sec)