CS553 Cloud Computing

Programming Assignment 2 Report

Priyanka Makhijani A20392003

Prajakta Pardeshi A20352045

Problem statement:

This programming assignment covers the method of sorting larger datasets 128GB and 1TB on Amazon instances. Methods implemented to sort the data are:

1. Shared Memory Tera-sort

Methodology: (Psuedocode)

For shared memory the following approach is used.

1. Create chunked data = (Total file to be sorted)/(RAM size)

2. Read the data in main memory and sort by mergesort method.

3. Write the sorted data to disk which are temporary files.

4. Repeat steps 1,2 and 3 until all of the data is in sorted chunks, which now need to be merged into one single output file.

5. Read the first sorted chunk into input buffers in main memory and allocate the remaining MB for an output buffer.

6. Perform a k-way merge and store the result in the output buffer. Whenever the output buffer fills, write it to the final sorted file and empty it. Repeat for all the chunks. Please note that in place merge sort is used.

References for Shared memory:

https://en.wikipedia.org/wiki/External_sorting

K-way merge logic has been referred from the links below:

http://www.geeksforgeeks.org/external-sorting/

2. Hadoop TeraSort

Methodology: (Psuedocode)

Sorting is one of the basic MapReduce algorithms to process and analyze data. MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.

1. Sorting methods are implemented in the mapper class itself.

2. After tokenizing the values in the mapper class, the Context class (user-defined class) collects the matching valued keys as a collection.

3. To collect similar key-value pairs (intermediate keys), the Mapper class takes the help of RawComparator class to sort the key-value pairs.

4. The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values (K2, {V2, V2, …}) before they are presented to the Reducer.

References:

https://arxiv.org/ftp/arxiv/papers/1506/1506.00449.pdf

https://www.tutorialspoint.com/map_reduce/map_reduce_algorithm.htm

Skeleton of the code had been referred from the links below.

http://santoshsorab.blogspot.com/2014/12/hadoop-java-map-reduce-sort-by-value.html

3. Spark TeraSort

Spark provides a faster and more general data processing platform. Spark lets you run programs up to 100x faster in memory, or 10x faster on disk, than Hadoop

1. Inputs the dataset from HDFS.
2. Map to {key, Value}.
3. Sort the data by key.
4. Map to {Key, Value}
5. Write the sorted partition to disk.
6. Merge the portioned files explicitly (not in the Scala script).

References:

http://spark.apache.org/docs/latest/rdd-programming-guide.html
https://sparkour.urizone.net/recipes/installing-ec2/
https://stackoverflow.com/questions/37730808/how-i-know-the-runtime-of-a-code-in-scala
https://www.tutorialspoint.com/apache_spark/apache_spark_core_programming.htm

4. MPI TeraSort

MPI primarily addresses the message-passing parallel programming model***:*** data is moved from the address space of one process to that of another process through cooperative operations on each process.

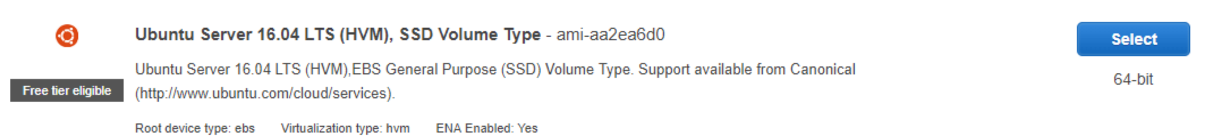
1. Passing ''messages" (data) amng processors on same node.
2. Divide the dataset into chunks by root node.
3. Each chunk is divided by number of processors and root node passes(scatters) the data.
4. Each processor gets a rank and sorts the piece of chunk
5. Root node gathers data and writes the sorted chunk to temporary sort file
6. After the entire data set is processed all the files are merged to get the final output sorted file.

References:

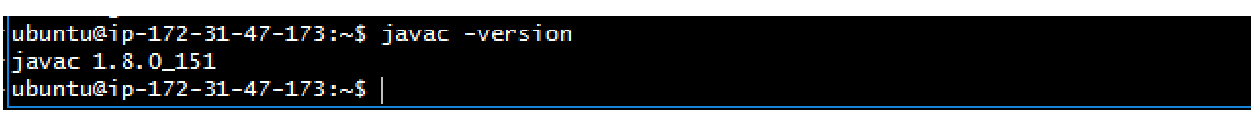
https://jetcracker.wordpress.com/2012/03/01/how-to-install-mpi-in-ubuntu/
http://mpitutorial.com/tutorials/mpi-hello-world/
http://supercomputingblog.com/category/mpi/

Version details of software used for Hadoop and spark installations:

<u>Linux version:</u> Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-aa2ea6d0 Amazon Machine Image.



<u>Java version:</u> javac 1.8.0_151



<u>Hadoop version:</u>

To run terasort on Hadoop we used Hadoop-2.8.2

http://apache.claz.org/hadoop/common/hadoop-2.8.2/hadoop-2.8.2.tar.gz

To run terasort on spark we used Hadoop -2.7.4

http://apache.claz.org/hadoop/common/hadoop-2.7.4/hadoop-2.7.4.tar.gz

Spark version:

To run Spark terasort we used Spark – 2.2.0

MPI Version:

3.2.1

Configuration 1 Virtual Cluster (1-node i3.large)

1. Shared Memory
   Instance Running Sort:

```
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# vi SharedMemory.cpp
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# g++ SharedMemory.cpp -o SharedMemory -lpthread
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# vi SharedMemory.cpp
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# g++ SharedMemory.cpp -o SharedMemory -lpthread
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# ./SharedMemory
```

   Sort Complete:

```
root@ip-172-31-47-173: /pp/hadoop/share/hadoop/64
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# vi SharedMemory.cpp
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# g++ SharedMemory.cpp -o SharedMemory -lpthread
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# vi SharedMemory.cpp
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# g++ SharedMemory.cpp -o SharedMemory -lpthread
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# ./SharedMemory

Total time for sorting : 22320 secs
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64#
```

   Sort Validate:

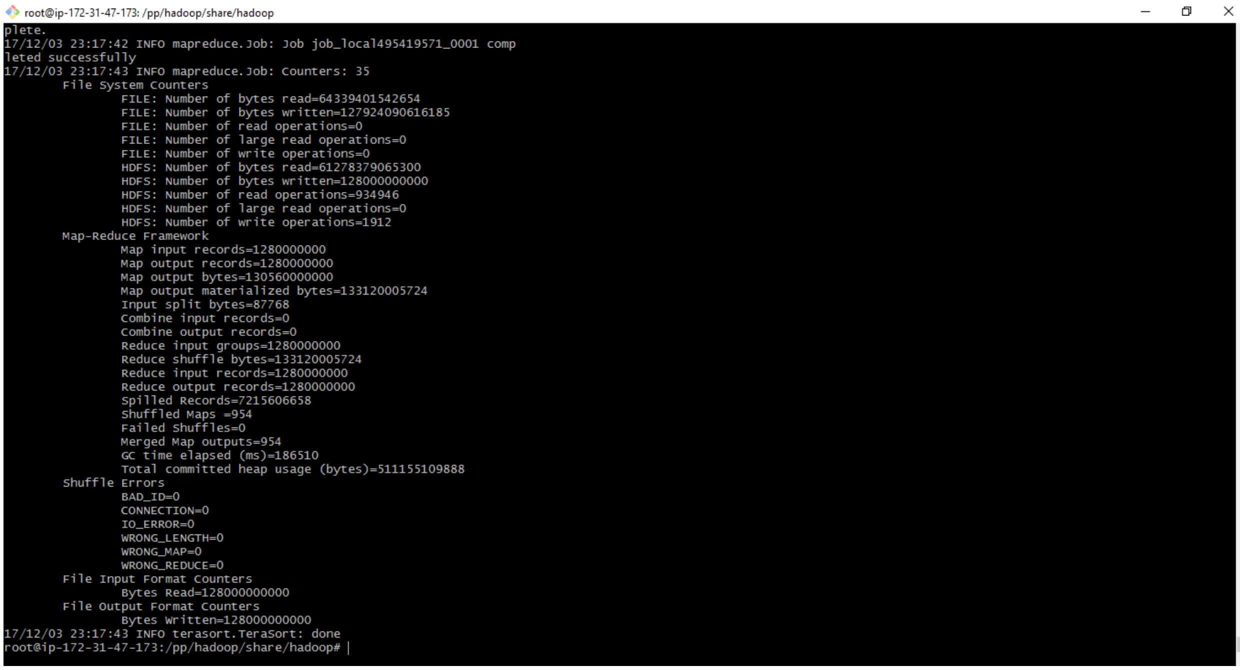
```
root@ip-172-31-47-173: /pp/hadoop/share/hadoop/64
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# ./valsort output1
Records: 1280000000
Checksum: 7341rq6jioe5y671
Duplicate keys: 0
SUCCESS - all records are in order
root@ip-172-31-47-173:/pp/hadoop/share/hadoop/64# |
```

2. Hadoop

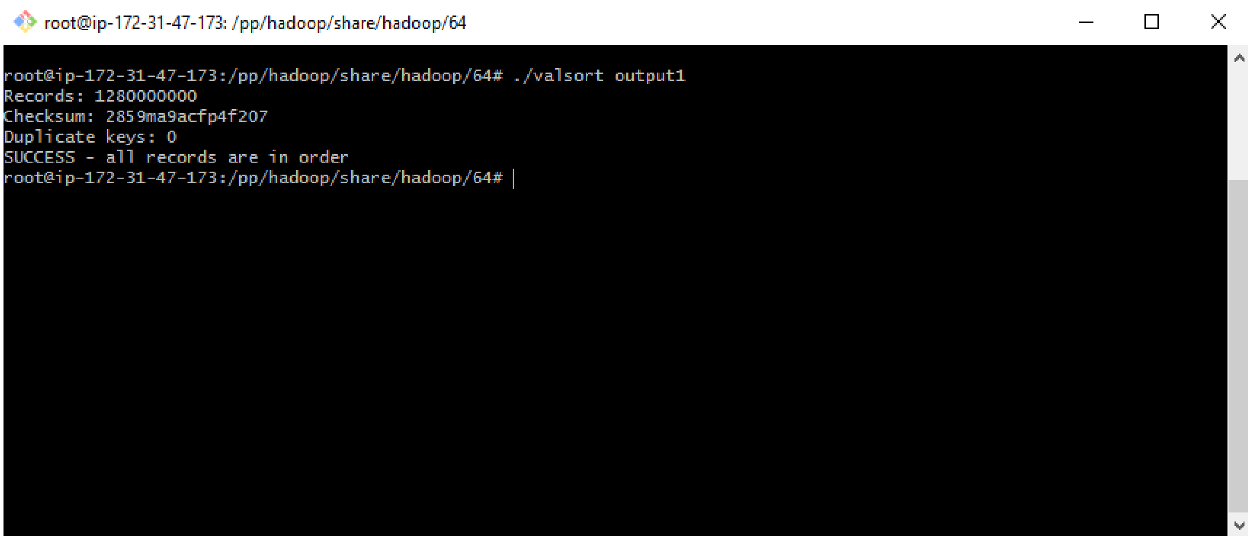
Instance Running Sort:

```
root@ip-172-31-47-173: /pp/hadoop/share/hadoop
17/12/03 22:17:44 INFO mapreduce.Job:  map 100% reduce 51%
17/12/03 22:17:50 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:17:56 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:02 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:08 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:14 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:20 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:26 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:32 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:38 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:44 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:50 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:56 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:18:56 INFO mapreduce.Job:  map 100% reduce 52%
17/12/03 22:19:02 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:08 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:14 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:20 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:26 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:32 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:38 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:44 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:50 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:19:56 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:02 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:03 INFO mapreduce.Job:  map 100% reduce 53%
17/12/03 22:20:08 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:14 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:20 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:26 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:32 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:38 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:44 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:50 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:20:56 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:02 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:08 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:14 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:15 INFO mapreduce.Job:  map 100% reduce 54%
17/12/03 22:21:18 INFO mapred.Merger: Merging 10 intermediate segments out of a total of 19
17/12/03 22:21:20 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:26 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:32 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:38 INFO mapred.LocalJobRunner: reduce > sort
17/12/03 22:21:44 INFO mapred.LocalJobRunner: reduce > sort
```

Sort Complete:



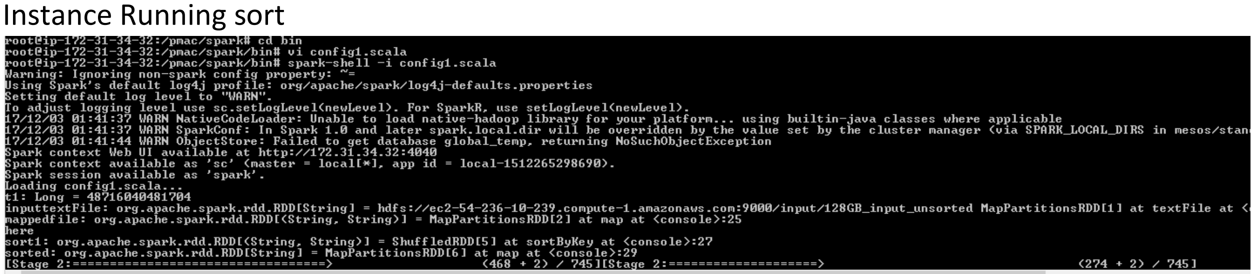
Sort Validate:



3. Spark

Screenshots:

Instance Running sort



Sort Complete

```
root@ip-172-31-34-32:/pmac# cd spark
root@ip-172-31-34-32:/pmac/spark# cd bin
root@ip-172-31-34-32:/pmac/spark/bin# vi config1.scala
root@ip-172-31-34-32:/pmac/spark/bin# spark-shell -i config1.scala
Warning: Ignoring non-spark config property: ~=
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/12/03 01:41:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/12/03 01:41:37 WARN SparkConf: In Spark 1.0 and later spark.local.dir will be overridden by the value set by the cluster manager (via SPARK_LOCAL_DIRS in mesos/stand
17/12/03 01:41:44 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Spark context Web UI available at http://172.31.34.32:4040
Spark context available as 'sc' (master = local[*], app id = local-1512265298690).
Spark session available as 'spark'.
Loading config1.scala...
t1: Long = 48716040481704
inputtextFile: org.apache.spark.rdd.RDD[String] = hdfs://ec2-54-236-10-239.compute-1.amazonaws.com:9000/input/128GB_input_unsorted MapPartitionsRDD[1] at textFile at <c
mappedfile: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[2] at map at <console>:25
here
sort1: org.apache.spark.rdd.RDD[(String, String)] = ShuffledRDD[5] at sortByKey at <console>:27
sorted: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[6] at map at <console>:29
res5: inputtextFile.type = hdfs://ec2-54-236-10-239.compute-1.amazonaws.com:9000/input/128GB_input_unsorted MapPartitionsRDD[1] at textFile at <console>:23 745]
duration: Double = 6556.892331831
Time taken to sort file :-6556.892331831seconds

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_151)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Sort Validate:

```
MINGW64:/c/Users/Pmac23/downloads
root@ip-172-31-34-32:/pmac/64# ./valsort output
Records: 1280000000
Checksum: 2769nb9azfp4f287
Duplicate keys: 0
SUCCESS - all records are in order
root@ip-172-31-34-32:/pmac/64#
```

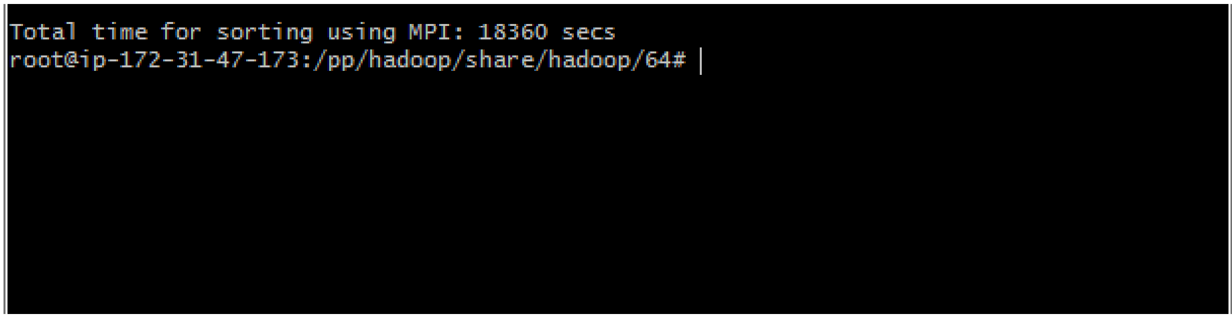
4. **MPI**

   Screenshots:

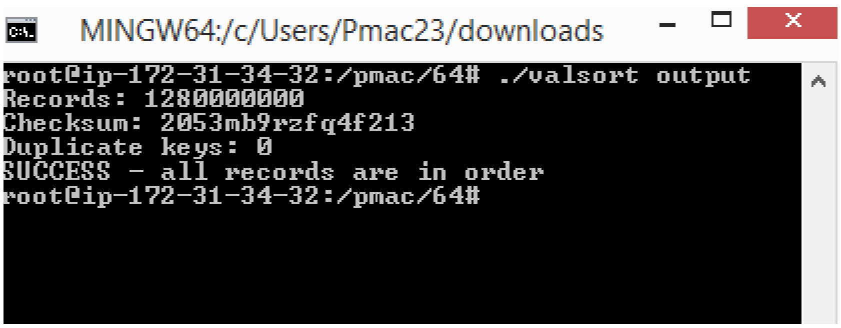
   Instance running sort

```
mpiexec@ip-172-31-34-32] Press Ctrl-C again to force abort
root@ip-172-31-34-32:/pmac/mpi# cd ..
root@ip-172-31-34-32:/pmac# clear

root@ip-172-31-34-32:/pmac# df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            7.5G     0  7.5G   0% /dev
tmpfs           1.5G  8.9M  1.5G   1% /run
/dev/xvda1      7.7G  2.0G  5.8G  26% /
tmpfs           7.5G     0  7.5G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           7.5G     0  7.5G   0% /sys/fs/cgroup
tmpfs           1.5G     0  1.5G   0% /run/user/1000
/dev/nvme0n1    436G  122G  292G  30% /pmac
/dev/xvdf       394G   73M  374G   1% /pmac1
tmpfs           1.5G     0  1.5G   0% /run/user/0
root@ip-172-31-34-32:/pmac# cd mpi
root@ip-172-31-34-32:/pmac/mpi# vi sort1.c
root@ip-172-31-34-32:/pmac/mpi# mpicc sort1.c -o terasort
root@ip-172-31-34-32:/pmac/mpi# mpirun -np 2 ./terasort 2 12GB_Input

NUMBER OF PROCESSES: 2


NUMBER OF PROCESSES: 2
```

Sort Complete

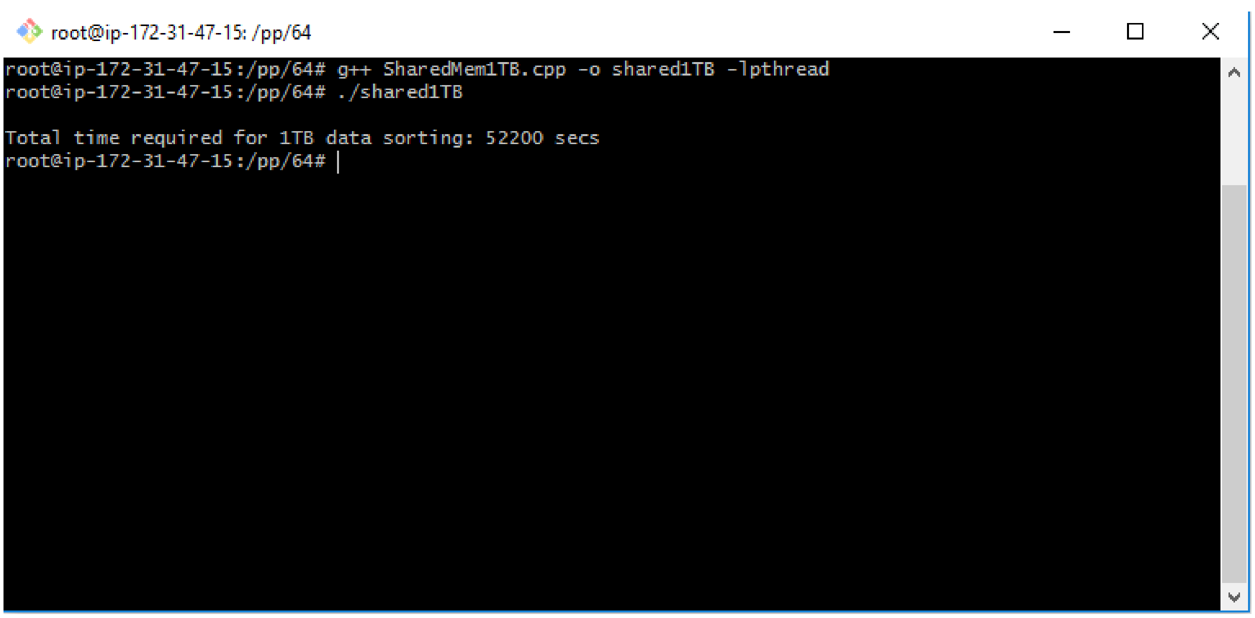


Sort Validate



Configuration 2 -Virtual Cluster (1-node i3.4xlarge)

1. **Shared Memory**



Sort Complete

root@ip-172-31-47-15: /pp/64

```
root@ip-172-31-47-15:/pp/hadoop/share/hadoop/64# ./valsort output1
Records: 10000000000
Checksum: 7489ea9acip9f192
Duplicate keys: 0
SUCCESS - all records are in order
root@ip-172-31-47-15:/pp/64#
```

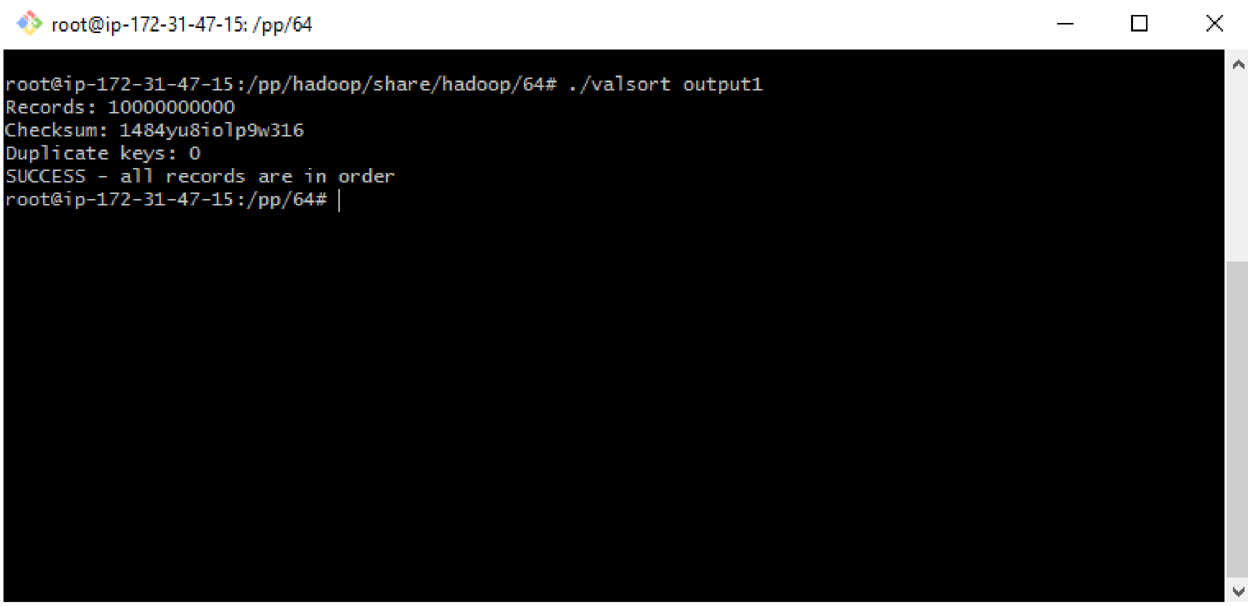
Sort Validate

2. **Hadoop**

root@ip-172-31-47-15: /pp/hadoop/share/hadoop/mapreduce

```
                Reduce shuffle bytes=1040000048
                Reduce input records=10000000000
                Reduce output records=10000000000
                Spilled Records=29395200041
                Shuffled Maps =8
                Failed Shuffles=0
                Merged Map outputs=8
                GC time elapsed (ms)=256
                Total committed heap usage (bytes)=4414504900060
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1000000000000
        File Output Format Counters
                Bytes Written=1000000000000
17/12/04 04:54:40 INFO terasort.TeraSort: done
root@ip-172-31-47-15:/pp/hadoop/share/hadoop/mapreduce#

root@ip-172-31-47-15:/pp/hadoop/share/hadoop/mapreduce#
```

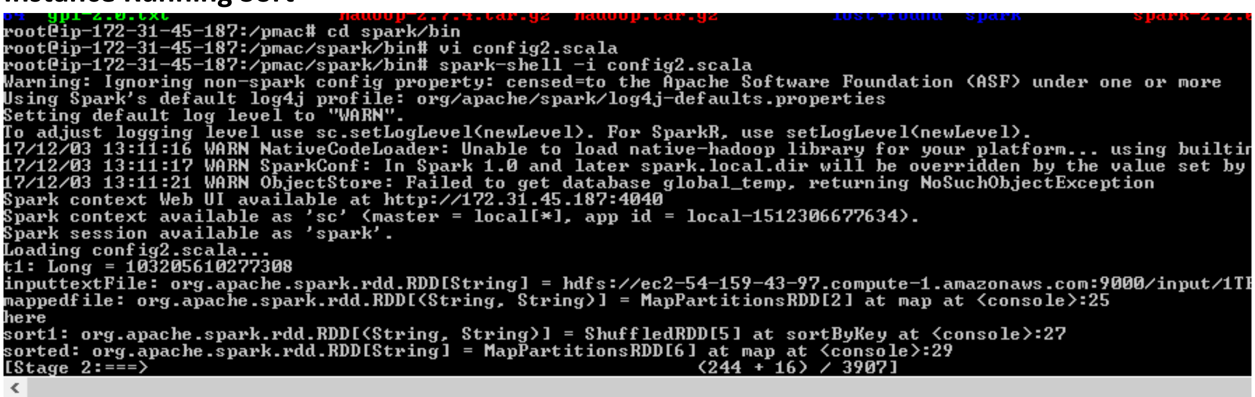
Sort Complete



Sort Validate

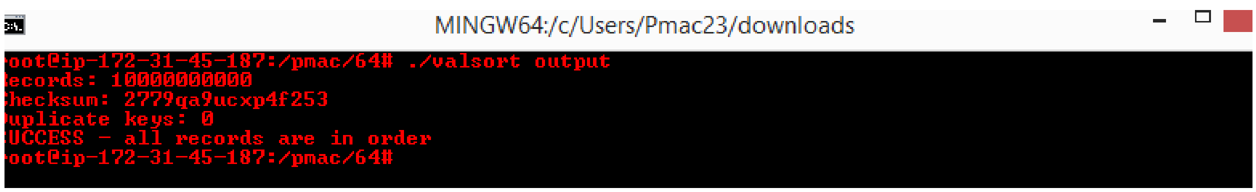
3. **Spark**

Instance Running Sort



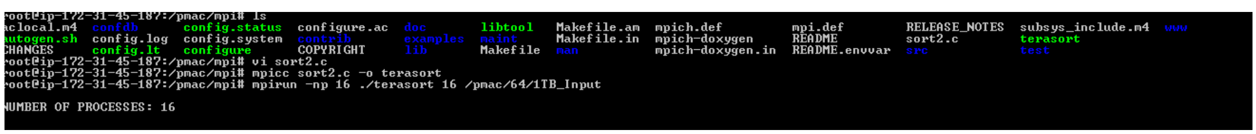
Sort Complete

Sort Validate

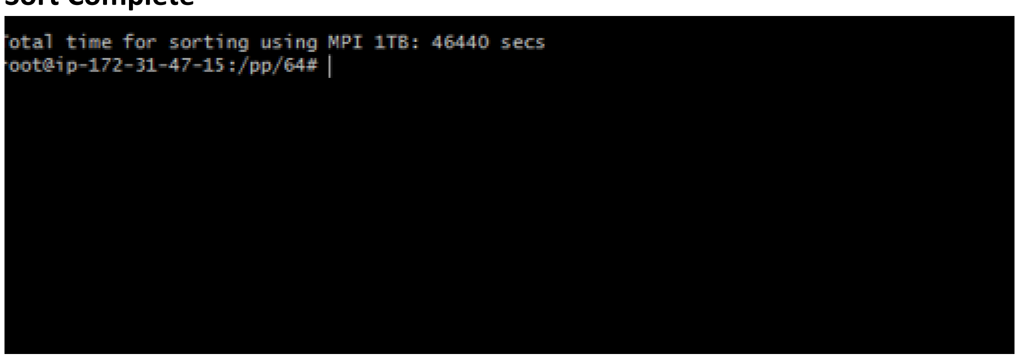


4. MPI

Instance Running Sort



Sort Complete



Sort Validate

Conclusions:

From the output and the time required for the system to perform terasorting it can be inferred that sparks perform is always better as compared to hadoop and shared memoy. Also, the implementation complexity of spark is less when compared to hadoop and shared memory. Spark also has a winning factor over others since it works inplace. However, if terasorting has to be done on one node then shared memory external sorting paradigm can also be considered as an option since the iplementation cost of shared memory is relatively less.

Which seems to be best at 1 node scale ?

When the parameter to compare is only performance then spark performs better. However, when it comes to 1 node scale a major factor to be considered here is also implementation cost. Moreover, shared memory also has relatively better performance and has low implementation cost since it does not required any major configurations and installations of master and slaves communication.

How about 8 nodes ?

When it is 8 nodes spark should be a better option. This is because spark has better architecture organization as compared to Hadoop and shared memory since it uses RDD's (Resilient Distributed Datasets). It can be much faster when it is compared with the performance of other methodologies when we have to process large amount of data.

Can you predict which would be best at 100 node scale?

Spark does in memory computing which will avoid space complexity and due to it's architecture it is very well suited for distributed file system computing. As the number of nodes increases the main memory at every node will also increase. This will provide better performance and failure handling.

How about 1000 node scales ?

Here also we would perform spark to run on 1000 scale. As explained in the answer above. We have seen how the performance of spark is increasing with higher configuration. Due to it's in memory computing capability and increase in the number of nodes which will give us more higher performance.

what can you learn from the CloudSort benchmark ?

With large amount of data to process and access disk becomes a bottleneck and hence we need to use the external sorting technique to reduce the bottleneck of the disk. CloudSort benchmark works on this methodology which uses the resources available in the cloud to sort data. With the help of external sorting on the cloud we can sort huge amount of data.

Experiment (instance/dataset)	Shared Memory TeraSort	Hadoop TeraSort	Spark TeraSort	MPI TeraSort
Compute Time (sec) [1xi3.large 128GB]	22320	15300	6556	18360
Data Read (GB) [1xi3.large 128GB]	128	128	128	128
Data Write (GB) [1xi3.large 128GB]	128	128	128	128
I/O Throughput (MB/sec) [1xi3.large 128GB]	11.7	16.7	33.4	13.9
Compute Time (sec) [1xi3.4xlarge 1TB]	52200	32887	29831	46440
Data Read (GB) [1xi3.4xlarge 1TB]	1000	1000	1000	1000
Data Write (GB) [1xi3.4xlarge 1TB]	1000	1000	1000	1000
I/O Throughput (MB/sec) [1xi3.4xlarge 1TB]	38.3	60.8	67.04	43.06
Compute Time (sec) [8xi3.large 1TB]	N/A			
Data Read (GB) [8xi3.large 1TB]	N/A			
Data Write (GB) [8xi3.large 1TB]	N/A			
I/O Throughput (MB/sec) [8xi3.large 1TB]	N/A			
Speedup (weak scale)	12.4	28	23	15
Efficiency (weak scale)	11.4%	22.3%	36.8%	18.2%