TeraSort Application for Shared-memory, Hadoop, and Spark

Sunny Changediya
A20353568

Content

Introduction:

TeraSort is an application used to sort large file of size usually greater than CPU memory. This application is tested on varying file sizes as 1GB, 100GB, and 1TB of data. The input datafile is generated using Gensort application with each record of 100 Bytes each in size. The TeraSort application is developed for Shared-Memory using Python multiprocessing for making advantage of multiple and parallel execution of processes, for Hadoop using Java and Hadoop Java libraries, and for Spark using Scala spark libraries.

Experiment Details:

The experiments are done on Amazon Aws EC2 spot instances with Ubuntu 14.01 as base AMI. Shared-Memory is implemented on single node, Hadoop & Spark on 16 cluster node of Amazon EC2 c3.large spot instance.

Configuration:
AMI: Ubuntu 14.01 64-bit
vCPU: 2
Disk: 2 * 16GB (SSD)

- ## Gensort:

The gensort program can be used to generate input records for the sort benchmarks. Gensort can be used to generate separate input partitions. This allows multiple instances of gensort to be run in parallel to generate the sort benchmark input.
The following code will generate input datafile of size 1GB and place data into file datafile.
Eg. gensort -a 10000000 datafile

- ## Shared-Memory:

Shared Memory is a Terasort application developed in Python and taking advantage of Python multiprocessing to sort and merge file records in parallel and concurrently. Shared Memory is designed and developed on same line with Map-Reduce analogy and architecture implementation. To achieve parallel and concurrent processing of input file and records, I have made use of Python multiprocessing which is substitute for threading. The files are sorted using Heap Sort algorithm using max-heap property in O(n Log n) time. The files are sorted by parallel child processes concurrently. In last phase the sorted input chunks are merged to produce single sorted datafile.

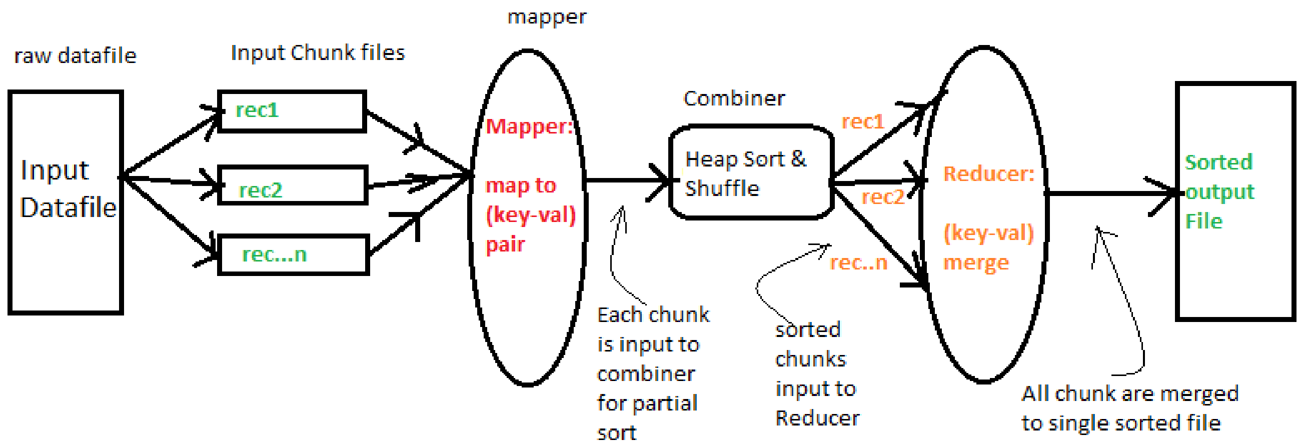


Fig:1. Shared memory architecture and working flow.

The design and working of shared memory is explained below.(refer Fig.1)

a) Shared Memory takes input as datafile generated from gensort application. The program accept number of child processes to be created as an argument. Since shared-memory uses multiprocessing, many child processes of parent process are created to achieve parallel processing.
b) It splits the input file into small file chunks usually of size equal to memory available into system, so that full CPU utilization can be achieved and processor kept busy without wasting resources. The file chunks are named based on increasing indexes to keep track of file numbers.
c) Then it removes original input datafile so as to minimize disk capacity. So at any time, disk contains data of size equal to input datafile only.
d) Heap Sort is implemented using max-heap property to sort file records based on key value pair. The key is 10 Byte field and value contains 90 Bytes making single record of size 100 Bytes. The max-heap is built using keys extracted from file and stored using max-heap property in a tree. Heap sort running time is O(n log n) with space complexity O(n).
e) The sorted records are overwritten to same source file chunk to maintain disk capacity and saves disk from overflowing of available capacity. This is where shared memory is more efficient in maintaining disk utilization. The sorted files are updated with index value to keep track while merging.
f) The sorted files chunk are then merged using multiprocessing and reduced to single sorted file.
g) The bottleneck happens in case of merging phase where program travels same sorted records many times, increasing running time and making it slow.

- ## Hadoop Terasort:

Hadoop is a parallel processing programming model used to process large chink of data in the format of key value pair. It uses MapReduce programing model to divide files into small chunks which acts as an input to mapper and to merge them internally by reducer. Hadoop uses YARN as an inbuilt standalone scheduler to schedule jobs and worker processing. YARN uses jobtracker to keep track of worker functions. The jobtracker schedules and tracks worker processes on many nodes for parallel and concurrent processing which is mainly executed by jobseekers. Jobseekers periodically sends data process status information to jobtracker so that jobtracker can track failed and processed nodes and schedule more worker processes on idle nodes. Hadoop also uses its own distributed file system called HDFS. It is a file system that manages storage across network of machines called distributed file system. Files in HDFS are written by single writer and are always done at the end of file. HDFS has a block size, which is minimum data that can be written or read. HDFS has block size of 64Mb or 128Mb based on system processor size. HDFS cluster has two types of nodes, NameNode and DataNode. NameNode is called as Master and used to manage file system name-space. It maintains file system tree and meta-data for files. While DataNode is called workers, and used to store and retrieve blocks of file records. They also report back to NameNode periodically.

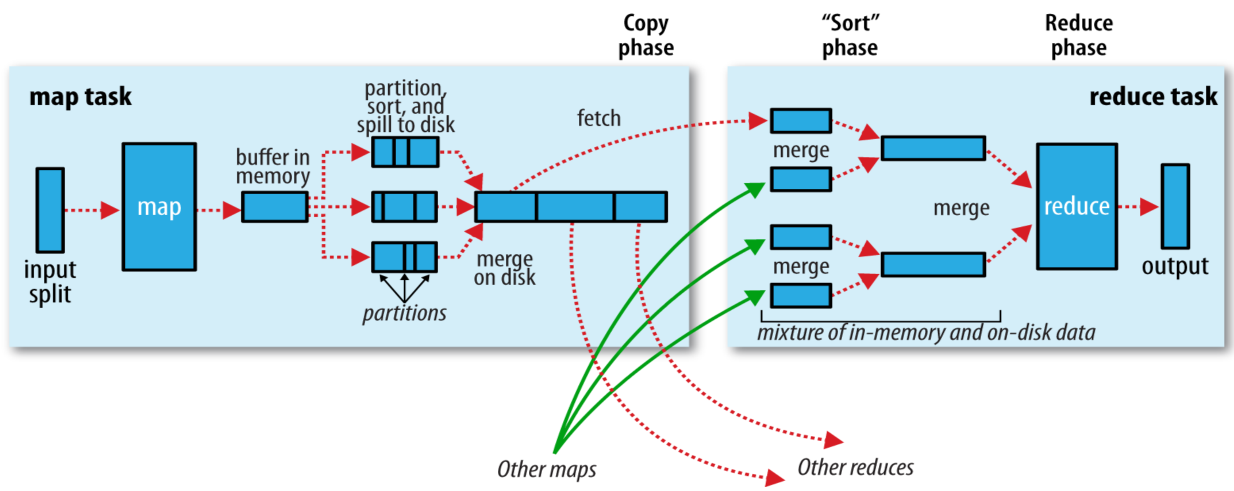


Fig.2: Hadoop MapReduce working flow for Terasort Hadoop

The working flow of Terasort Hadoop application is as follows:

a) Hadoop Terasort application is developed using Java Hadoop libraries to sort large input datafile.
b) The code includes three files with main class object at terasort.java, mapper for hadoop at terasortmapper.java, and reducer for hadoop at terasortReducer.java.
c) The input datafile is stored on HDFS file system for distributed data processing on many parallel worker nodes. The mapper function reads file chunks and processes them to form key-value pair.
d) The processed key-value pair were supplied as an input to Map() hadoop function, which further used as an input to combiner function.
e) Combiner() of hadoop is the model where actual sorting takes place. The data is sorted based on key-value pair and added as an input to Reducer() function on Hadoop, so that reducer can work around on merging step.
f) The reducer takes input as sorted key-value pair from combiner and merge them to single sorted file.
g) Hadoop involves many intermediate disk I/O like read, write to process large data. This is where hadoop perform bottleneck as more disk space is used.

• Spark Terasort:

Spark is a cluster computing platform designed to be fast for general purpose. One of main feature spark offers is the ability to run computations in memory, but the system is also faster than Map-reduce for complex applications running on disk. Spark can run on Hadoop clusters and access hadoop data source including YARN and HDFS. Spark provides two main abstractions for parallel programming: resilient distributed database and parallel operations.

The main components of spark are as follows:

1. Spark Core: Provides basic functionality to Spark like task scheduling, Memory Management, Fault tolerance, storage systems, and defining RDD api's.
2. RDD: RDD is an immutable distributed collection of objects. Each RDD is split into multiple partitions. Once RDD's are created, they offer two types of operations a) Transformation & b) Action. Transformation contains set of instructions to be executed on call to Action on RDD. Actions on the other hand compute result based on RDD and may return to driver program or save it to HDFS. RDD can persist data into memory if needed by application for many time to save extra computation time.

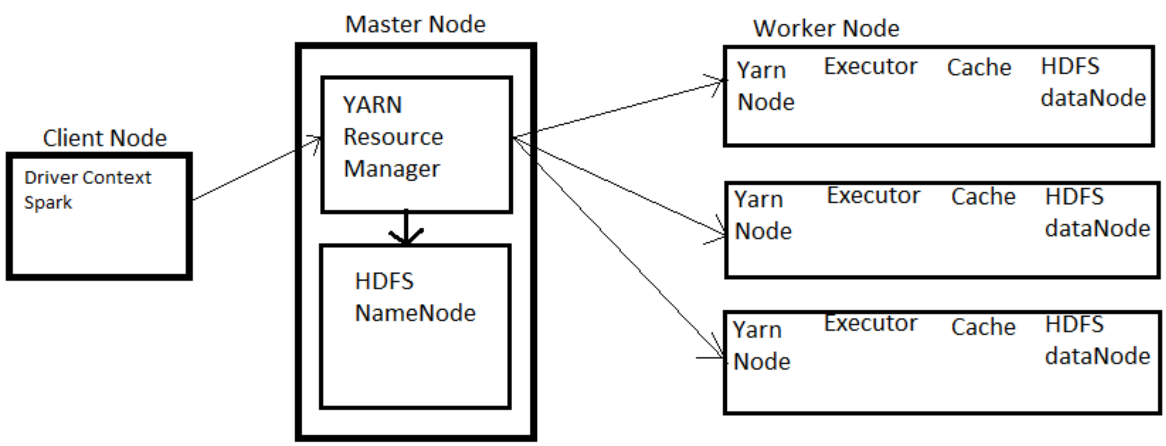


Fig.3: Execution flow of Terasort Spark using YARN Resource Manager

- The working flow of Terasort Spark is as follows:
  a) Terasort Spark application uses Scala code to make efficient use of spark libraries available for Scala.
  b) The Scala code is submitted to driver program in the form of jar so as to distribute it across network. The driver program is the main program of Scala which starts spark execution.
  c) To provide better abstraction of HDFS, hadoop, and Spark libraries, I have used YARN as a standalone resource manager instead of Spark's own standalone manager as shown in Fig.1.
  d) Spark driver calls YARN-cluster mode to execute Spark on cluster instances. Driver then runs inside YARN context which has fairly simple access to HDFS. Then in turn YARN manages the worker node execution and resource allocation. Here only YARN node manager can start executors for Spark.
  e) The code first creates a RDD which reads datafile into memory. It then uses map() to generate key-value pair to be used by executors to process file data. The action sortbykey() is called on RDD to sort the file based on keys. The executors perform the actual sorting operations. Each executor then arranges file based on keys such that keys with same ASCII values are clubbed

into same file for processing.

f) Each Spark executor outputs a separate file which is sorted by key type.

Virtual Cluster Setup:

for Virtual Cluster setup, I have used 16 Node spot instances of type c3.large for data size of 1Gb, 10GB, and 100GB. For data size of 1TB, I have used virtual cluster of spot instances of type d2.xlarge from Amazon AWS EC2 resources. Virtual cluster is used only for parallel programming models such as Hadoop & Spark. Shared memory is performed on single node instance for all data sizes 1Gb, 10GB, and 100GB, and 1TB on d2.xlarge and c3.large. The virtual cluster is setup by updating some packages for Ubuntu instance.

Following are the necessary steps to setup a virtual cluster:

1. Request required number of spot instances from Amazon EC2.
2. Setup required update and configure disk size. Convert SSD disks into raid array using steps given in file “raid-commands”. Follow the instructions to setup raid array of disks for storage. Do ssh to all instances and install openssh using “sudo apt-get install openssh”.
3. Generate the input datafile of records to work on. Execute the script “ksh generate_data.sh” on only master node instance and execute “./gensort -a 1000000 datafile” to generate file of size 1MB named datafile.
4. Shared-Memory: It does not require any setup and packages to be installed.
5. Hadoop: It requires changes into configuration files to be done. Follow the steps given in file “working_with_hadoop_stepwise” to configure & install hadoop on all instances.
6. Spark: Since we are using YARN as a resource manager for Spark master & executors, some configuration files for Spark needs to be changed as per “working_with_spark_stepwise”.
7. Once setup is done, we can execute Terasort application using all three models.

Running Applications:

Shared-Memory:

- Shared-memory is implemented in Python using multiprocessing instead of threading in python. Python-multiprocessing provides many child processes to be created to achieve parallel programming and run concurrent executions.
- To run execute follow the steps in file “README_Shared_Memory”.

Terasort Hadoop:

- Configure storage for Hadoop using RAID array of SSD Disks. Follow the steps in file “raid_commands”
- configure Hadoop on 16-Node instance using steps given in “README_Hadoop”. It contains steps as well as automated scripts to be executed to avoid manual work.
- Hadoop Configuration Files and Descriptions:

Config File name	Description
conf/master	Used to specify masterNode IP. It specifies who is going to be master.
conf/slaves	It specifies IP address of all slave nodes. Master uses it to specify work for slaves.

conf/core-site.xml	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS, MapReduce, and YARN.
Conf/hdfs-site.xml	Configuration settings for HDFS daemons: namenode, the secondary namenode, and the datanodes.
conf/mapred-site.xml	Configuration settings for MapReduce daemons: the job history server
Conf/yarn-site.xml	Configuration settings for YARN daemons: the resource manager, the web app proxy server, and the node managers

- **Master Node:**
  The master nodes in distributed Hadoop clusters host the various storage and processing management services like NameNode: which manages HDFS, Resource Manager: which keeps YARN configured, jobtracker: which handles cluster resource managers which is option for YARN. Master Node is used as a heart of Hadoop driver program. It manages workers, schedules job on workers, and check failure management on workers.

- **Slave Node:**
  Slave nodes are worker nodes scheduled by Master Node to perform actual task. They perform shuffle and sort for MapReduce program. They periodically inform master node of status of processing and failure. All slaves work in parallel and independent of other giving abstraction of concurrent execution. There are many slaves, but only single master node.

- **Number of Mapper:**
  The number of maps is usually driven by the number of DFS blocks in the input files. Although that causes people to adjust their DFS block size to adjust the number of maps. The “mapred.map.tasks” parameter is just a hint to the InputFormat for the number of maps. The default InputFormat behavior is to split the total number of bytes into the right number of fragments. However, in the default case the DFS block size of the input files is treated as an upper bound for input splits. The number of map tasks can also be increased manually using the JobConf's “conf.setNumMapTasks(int num)”. This can be used to increase the number of map tasks, but will not set the number below that which Hadoop determines via splitting the input data.

- **Number of reducer:**
  The ideal reducers should be the optimal value that gets them closest to: (* A multiple of the block size * A task time between 5 and 15 minutes * Creates the fewest files possible). The number of reduce tasks can also be increased manually using the JobConf's “conf.setNumReduceTasks(int num)”.

Terasort Spark:

- Terasort for Spark is coded using Scala program. First install “sbt” and “scala” package in ubuntu.
  1. sudo wget http://dl.bintray.com/sbt/debian/sbt-0.13.6.deb
  2. sudo dpkg -i sbt-0.13.6.deb
  3. sudo apt-get update
- add “build.sbt” provided with code and do “sbt package” in the directory where “build.sbt” and “terasort.scala” code are located.
- Execute steps in “working_with_Spark_stepwise”.

Performance:

The performance of Shared-Memory, Terasort Hadoop, and Terasort Spark are compared using various parameters like threading, disk size, CPU cores, and number of cluster instances.

A) Shared-Memory: Shared-memory is implemented using python-multiprocessing.

- Shared-Memory on 1-node C3.large instance using threads for data size 1GB, 10GB, and 100GB
- Shared-Memory on 1-node d2.xlarge instance using 8-threads for data size 1TB took 23.34 hours.
- Shared-memory using multiprocessing threads on 1-Node on data size 1Gb, and 10GB shows variations in execution time to process sorting of data file based on size. It seems that multiprocessing using threads increases efficiency of program run and improves running time. Since CPU cores are limited, there is not much variation in running time for same data size using 4-thread and 8-Thread. So it shown that thread switching takes considerable time as execution time has not deviated much from origin in both cases. Running time is shown in seconds and data size ranges in GB's.

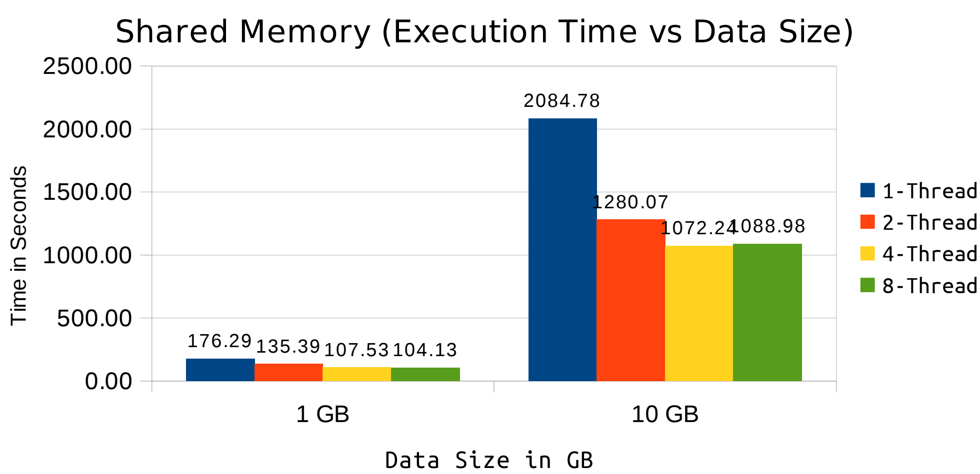


Fig. 5: Shared-memory on 1-node C3.large instance using threads for data size 1GB, 10GB, 100GB.

- Shared-memory using multiprocessing threads on 1-Node on data size 1Gb, 10GB, and 100GB shows variations in execution time to process sorting of data file based on size. It seems that multiprocessing using threads increases efficiency of program run and improves running time.

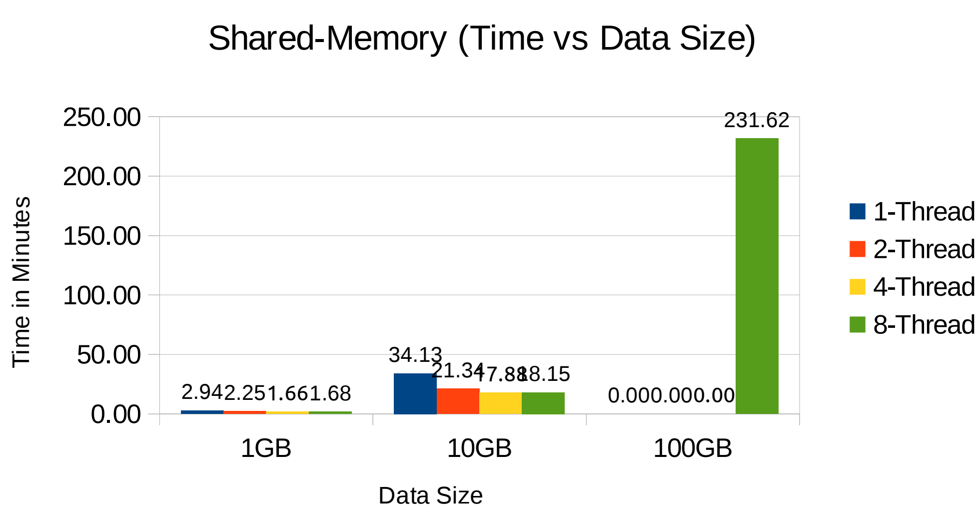


Fig. 6: Shared-memory on 1-node instance using threads for data size 1GB, 10GB, 100GB.

- ## Shared-memory Execution for varying data size:

a) Data Size: 1GB, Time taken: 164.69 sec. On 1-node c3.large:

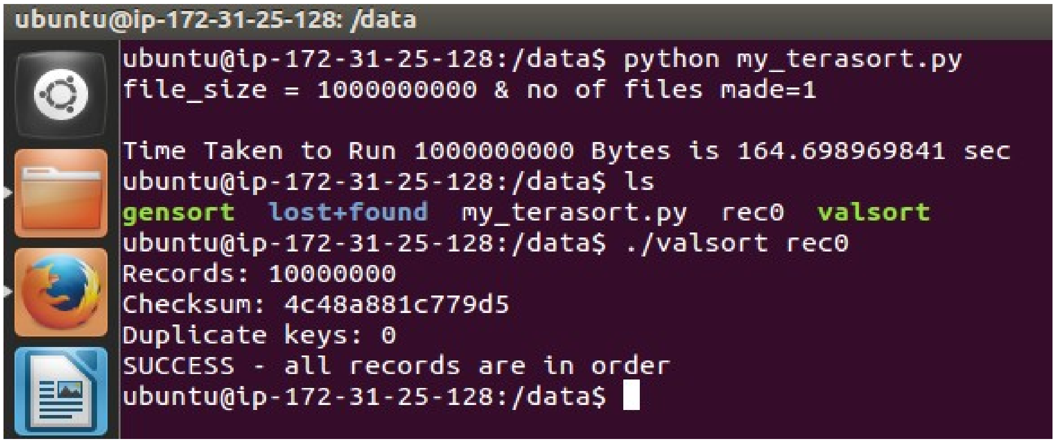


fig: shows shared memory execution and valsort checksum for 1GB file size

b) Data Size: 10GB, Time taken: 1072.240 sec. On 1-node c3.large:

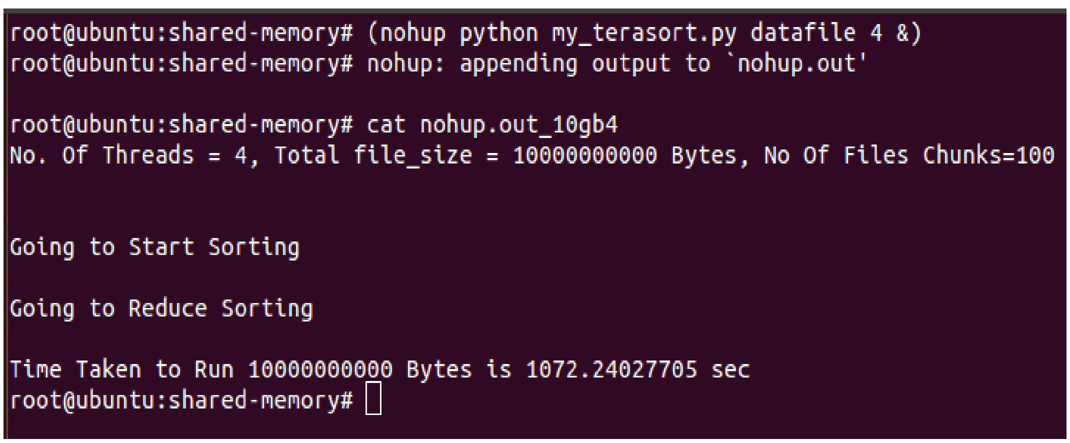


fig: shows shared memory execution and valsort checksum for 10GB file size. Due to massive read-writes, shared memory shows considerable time in sorting file.

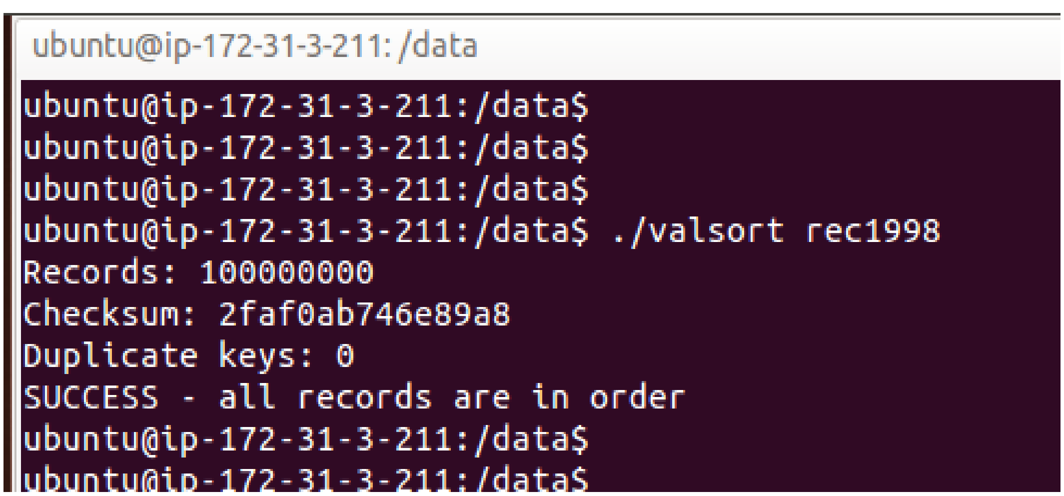


Fig. Valsort Comparison for checksum on above records:

c) Data Size: 100GB, Time taken: 13897.162 sec. On 1-node c3.large:

```
root@ubuntu:shared-memory# (nohup python my_terasort.py datafile 4 &)
root@ubuntu:shared-memory# nohup: appending output to `nohup.out'

root@ubuntu:shared-memory# cat nohup.out_100gb
No. Of Threads = 8, Total file_size = 100000000000 Bytes, No Of Files Chunks=250


Going to Start Sorting

Going to Reduce Sorting

Time Taken to Run 100000000000 Bytes is 13897.162122 sec
root@ubuntu:shared-memory#
```

fig: shows shared memory execution and valsort checksum for 100GB file size

d) Data Size: 1TB, Time taken: 84204.345 sec. On 1-node d2.xlarge:

```
root@ubuntu:shared-memory# (nohup python my_terasort.py datafile 8 &)
root@ubuntu:shared-memory# nohup: appending output to `nohup.out'

root@ubuntu:shared-memory# cat nohup.out
No. Of Threads = 8, Total file_size = 1000000000000 Bytes, No Of Files Chunks=5000


Going to Start Sorting

Going to Reduce Sorting

Time Taken to Run 1000000000000 Bytes is 84204.3444990 sec
root@ubuntu:shared-memory#
```

fig: shows shared memory execution and valsort checksum for 1TB file size

(a) shows shared-memory run on 1Gb data with valsort checksum. (b) – (d) shows shared-memory for 10Gb, 100GB and 1Tb data. The valsort checksum has been added into to verify shared memory data sorted as per keys.

Shared-Memory vs Terasort Hadoop vs Terasort Spark:

- Shared-memory, Terasort Hadoop, and Terasort Spark are executed on 1-node C3.large instance for data size 10GB, and 100GB to perform analysis on execution time varying according to data size.
- Following graph shows stark running time variation between Shared-memory, hadoop, and spark. While shared-memory, and Hadoop took same running time for small set of data size of 10GB, Spark executed much faster than both applications. It shows memory read-write are bottleneck to Hadoop performance. While Spark uses RDD to store machine instructions in memory and only on action call, actual execution takes place. It shows Spark is 10x faster than Hadoop in every single case.

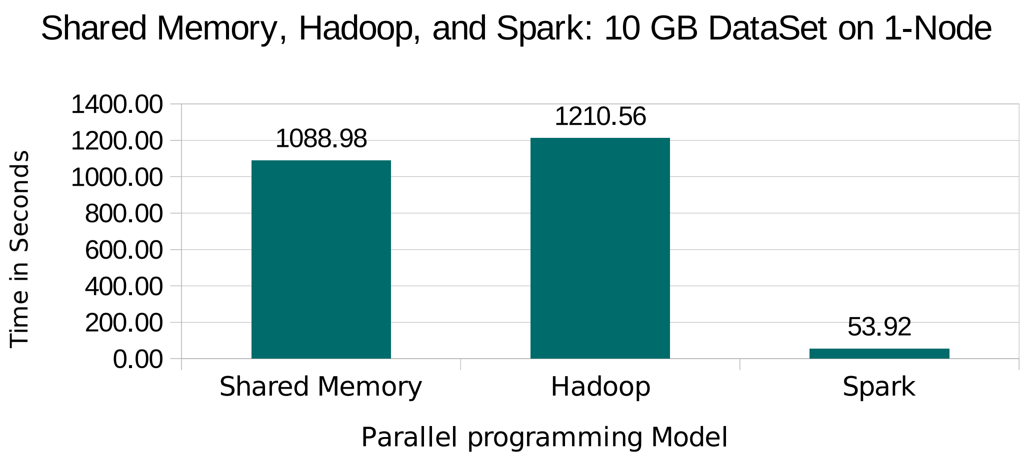


Fig: Above Graph shows that on 10GB dataset, executed on single node, spark runs 10x faster than Hadoop and shared memory due to RDD data object and partitioning tuning in RDD. It reduces considerable disk red-write as compared to hadoop & Shared-Memory.

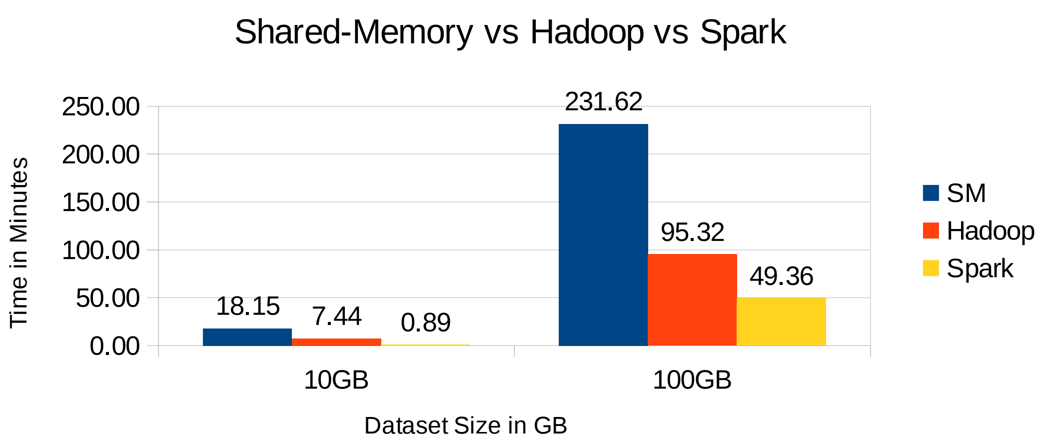


Fig: As shown, on dataset of size 10GB and 100GB, Shared-memory and Hadoop takes huge time in sorting file of records as compared to Spark. Spark is 10x times more faster than Hadoop and Shared-memory. The time shown is in minutes.

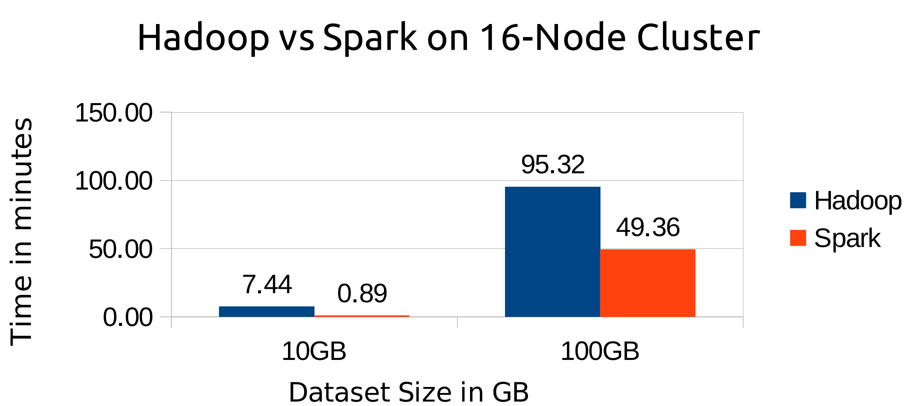


Fig. Above graph shows comparison of Hadoop vs Spark on 16-Node cluster setup for varying data sizes. Spark is 10x faster in performance due to RDD's and tuning in partitions.

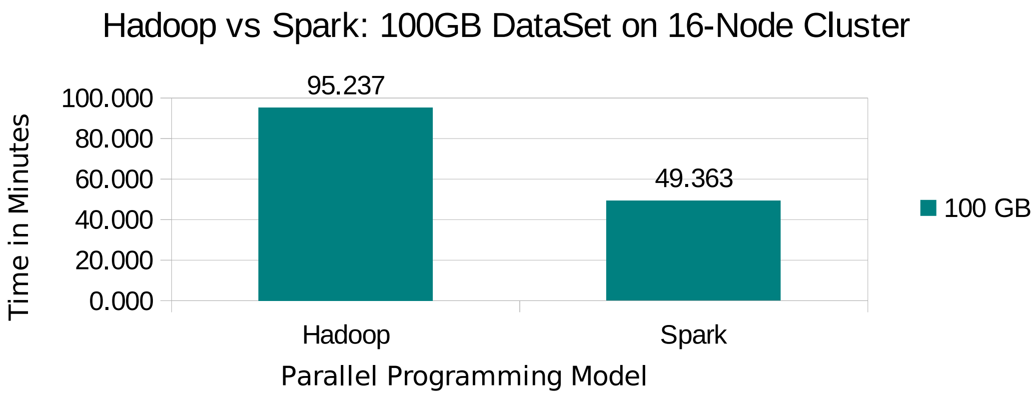
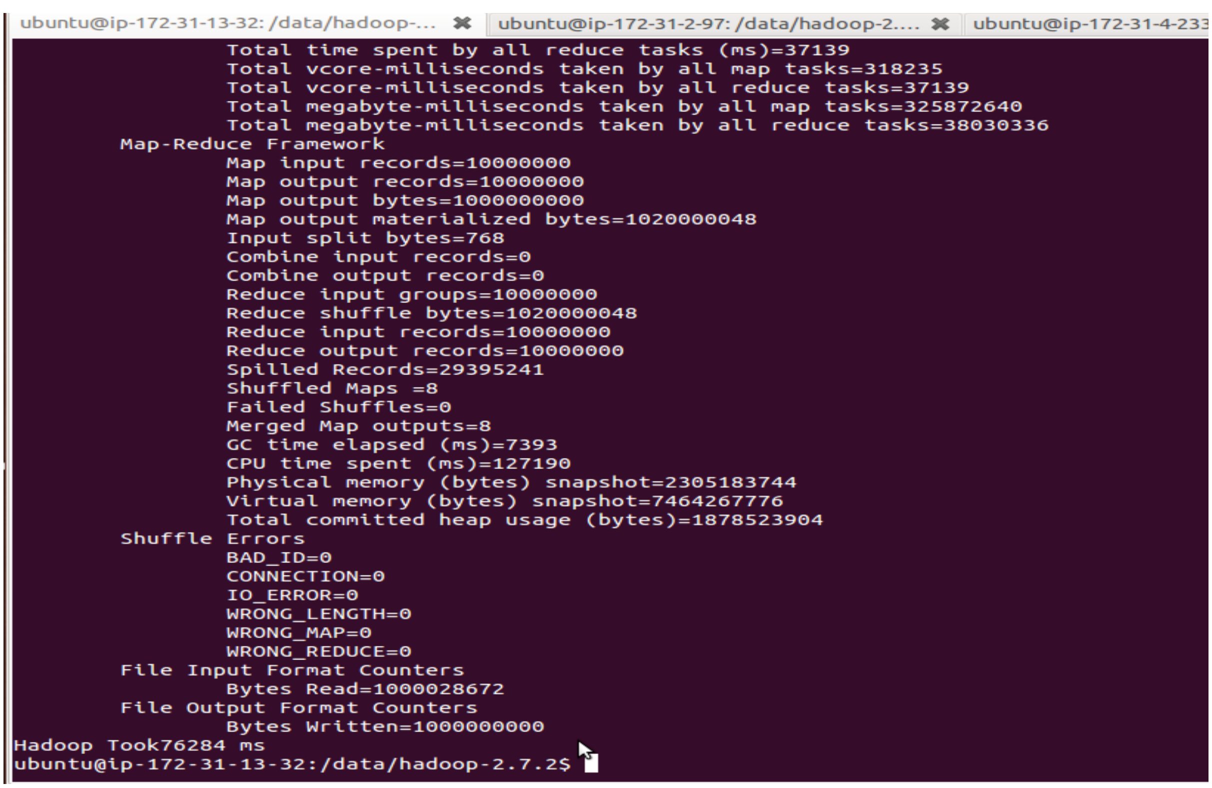


Fig: Graph shows variation of Hadoop & Spark on 100GB dataset using 16-node cluster. The time shown is in minutes. For large dataset, spark shown considerable improvement in performance. On cluster network, Spark uses YARN scheduler from Hadoop to perform execution.

Terasort Hadoop Executions:

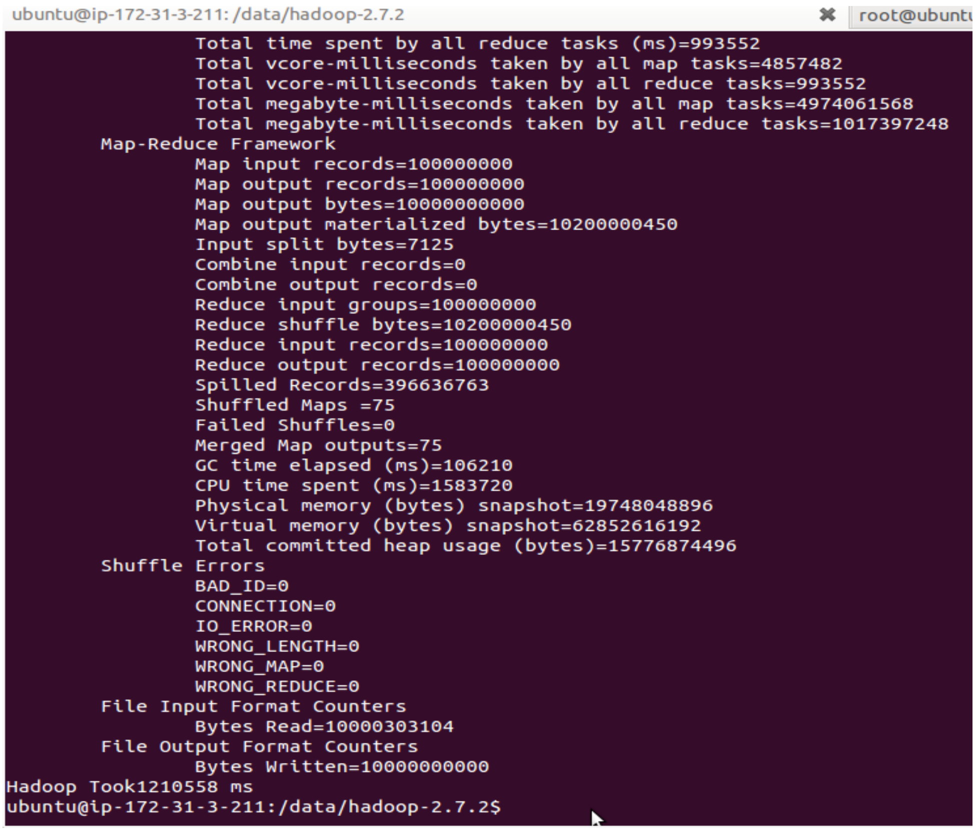
a) Terasort Hadoop 1GB 1-Node C3.large performance:



```
ubuntu@ip-172-31-13-32: /data/hadoop-...   ubuntu@ip-172-31-2-97: /data/hadoop-2....   ubuntu@ip-172-31-4-233
                Total time spent by all reduce tasks (ms)=37139
                Total vcore-milliseconds taken by all map tasks=318235
                Total vcore-milliseconds taken by all reduce tasks=37139
                Total megabyte-milliseconds taken by all map tasks=325872640
                Total megabyte-milliseconds taken by all reduce tasks=38030336
        Map-Reduce Framework
                Map input records=10000000
                Map output records=10000000
                Map output bytes=1000000000
                Map output materialized bytes=1020000048
                Input split bytes=768
                Combine input records=0
                Combine output records=0
                Reduce input groups=10000000
                Reduce shuffle bytes=1020000048
                Reduce input records=10000000
                Reduce output records=10000000
                Spilled Records=29395241
                Shuffled Maps =8
                Failed Shuffles=0
                Merged Map outputs=8
                GC time elapsed (ms)=7393
                CPU time spent (ms)=127190
                Physical memory (bytes) snapshot=2305183744
                Virtual memory (bytes) snapshot=7464267776
                Total committed heap usage (bytes)=1878523904
        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=1000028672
        File Output Format Counters
                Bytes Written=1000000000
Hadoop Took76284 ms
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
```

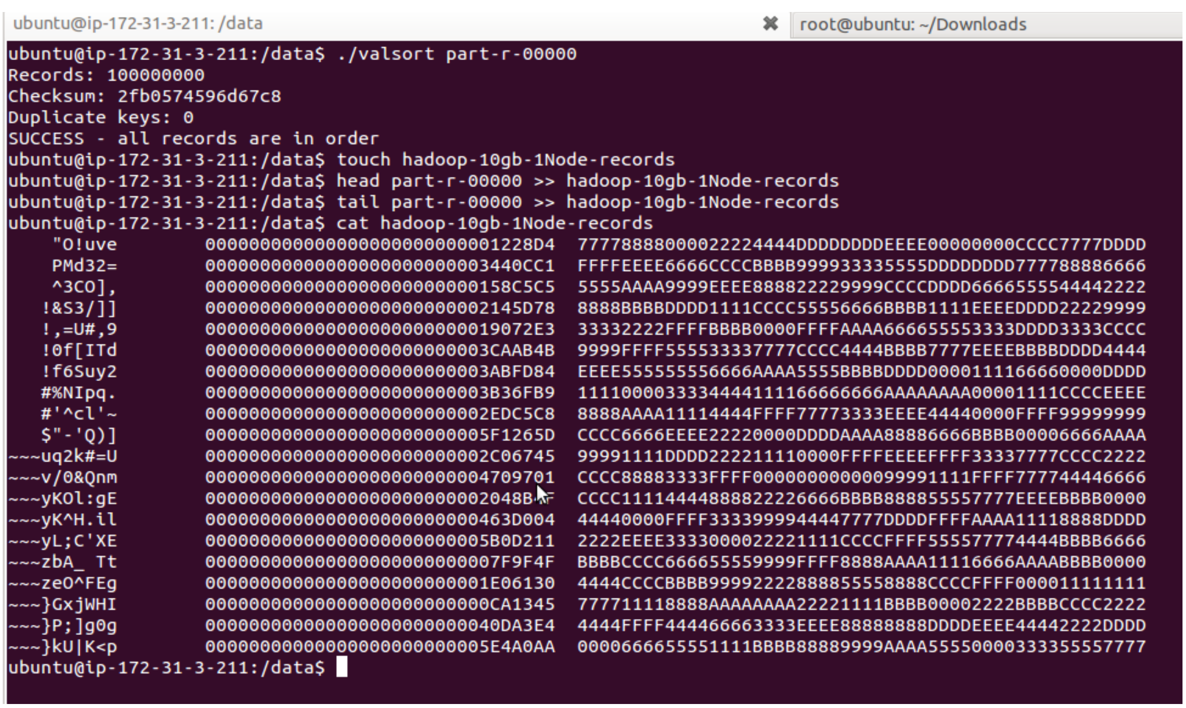
fig: shows Terasort Hadoop execution for 1-node cluster on 1GB file size

b) Terasort Hadoop 10GB 1-Node C3.large performance:



```
ubuntu@ip-172-31-3-211: /data/hadoop-2.7.2                                  root@ubuntu
                Total time spent by all reduce tasks (ms)=993552
                Total vcore-milliseconds taken by all map tasks=4857482
                Total vcore-milliseconds taken by all reduce tasks=993552
                Total megabyte-milliseconds taken by all map tasks=4974061568
                Total megabyte-milliseconds taken by all reduce tasks=1017397248
        Map-Reduce Framework
                Map input records=100000000
                Map output records=100000000
                Map output bytes=10000000000
                Map output materialized bytes=10200000450
                Input split bytes=7125
                Combine input records=0
                Combine output records=0
                Reduce input groups=100000000
                Reduce shuffle bytes=10200000450
                Reduce input records=100000000
                Reduce output records=100000000
                Spilled Records=396636763
                Shuffled Maps =75
                Failed Shuffles=0
                Merged Map outputs=75
                GC time elapsed (ms)=106210
                CPU time spent (ms)=1583720
                Physical memory (bytes) snapshot=19748048896
                Virtual memory (bytes) snapshot=62852616192
                Total committed heap usage (bytes)=15776874496
        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=10000303104
        File Output Format Counters
                Bytes Written=10000000000
Hadoop Took1210558 ms
ubuntu@ip-172-31-3-211:/data/hadoop-2.7.2$
```

fig: shows Terasort Hadoop execution for 1-node cluster on 10GB file size. From previous graph, since read-writes increased to considerable amount, it shows slow execution time



```
ubuntu@ip-172-31-3-211: /data                                  root@ubuntu: ~/Downloads
ubuntu@ip-172-31-3-211:/data$ ./valsort part-r-00000
Records: 100000000
Checksum: 2fb0574596d67c8
Duplicate keys: 0
SUCCESS - all records are in order
ubuntu@ip-172-31-3-211:/data$ touch hadoop-10gb-1Node-records
ubuntu@ip-172-31-3-211:/data$ head part-r-00000 >> hadoop-10gb-1Node-records
ubuntu@ip-172-31-3-211:/data$ tail part-r-00000 >> hadoop-10gb-1Node-records
ubuntu@ip-172-31-3-211:/data$ cat hadoop-10gb-1Node-records
    "O!uve     000000000000000000000000001228D4  77778888000022224444DDDDDDDDEEEE00000000CCCC7777DDDD
    PMd32=     00000000000000000000000003440CC1  FFFFEEEE6666CCCCBBBB999933335555DDDDDDDD777788886666
    ^3CO],     0000000000000000000000000158C5C5  5555AAAA9999EEEE888822229999CCCCDDDD6666555544442222
   !&S3/]]     00000000000000000000000002145D78  8888BBBBDDDD1111CCCC55556666BBBB1111EEEEDDDD22229999
   !,=U#,9     000000000000000000000000019072E3  33332222FFFFBBBB0000FFFFAAAA666655553333DDDD3333CCCC
   !0f[ITd     00000000000000000000000003CAAB4B  9999FFFF555533337777CCCC4444BBBB7777EEEEBBBBDDDD4444
   !f6Suy2     00000000000000000000000003ABFD84  EEEE55555555666 6AAAA5555BBBBDDDD000011116666 0000DDDD
   #%NIpq.     00000000000000000000000003B36FB9  11110000333344441111666666666AAAAAAAA00001111CCCCEEEE
   #'^cl'~     00000000000000000000000002EDC5C8  8888AAAA11114444FFFF77773333EEEE44440000FFFF99999999
   $"-'Q)]     00000000000000000000000005F1265D  CCCC6666EEEE22220000DDDDAAAA88886666BBBB00006666AAAA
~~~uq2k#=U     00000000000000000000000002C06745  99991111DDDD22221111 0000FFFFEEEEFFFF33337777CCCC2222
~~~v/0&Qnm     00000000000000000000000004709701  CCCC88883333FFFF000000000000 99991111FFFF777744446666
~~~yKOl:gE     00000000000000000000000002048B[illegible]  CCCC11114444888822226666BBBB888855557777EEEEBBBB0000
~~~yK^H.il     0000000000000000000000000463D004  44440000FFFF333399994444 7777DDDDFFFFAAAA11118888DDDD
~~~yL;C'XE     00000000000000000000000005B0D211  2222EEEE3333000022221111CCCCFFFF555577774444BBBB6666
~~~zbA_ Tt     00000000000000000000000007F9F4F   BBBBCCCC666655559999FFFF8888AAAA11116666AAAABBBB0000
~~~zeO^FEg     00000000000000000000000001E06130  4444CCCCBBBB999922228888555588888CCCCFFFF000011111111
~~~}GxjWHI     000000000000000000000000000CA1345  77771111 8888AAAAAAAA22221111BBBB00002222BBBBCCCC2222
~~~}P;]g0g     00000000000000000000000040DA3E4   4444FFFF444466663333EEEE88888888DDDDEEEE44442222DDDD
~~~}kU|K<p     00000000000000000000000005E4A0AA  0000666655551111BBBB88889999AAAA5555000033335555 7777
ubuntu@ip-172-31-3-211:/data$
```

Fig. Terasort Hadoop Valsort & All Sorted Records for 10GB Dataset Size:

c) Terasort Hadoop 100GB 16-Node C3.large performance:

```
ubuntu@ip-172-31-13-32: /data/hadoop-...    ubuntu@ip-172-31-2-97: /data/hadoop-2....    ubuntu@ip-172-31-4-233: /data/hadoop-...
                CPU time spent (ms)=15886670
                Physical memory (bytes) snapshot=199453720576
                Virtual memory (bytes) snapshot=620857204736
                Total committed heap usage (bytes)=155982495744
        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=100003047424
        File Output Format Counters
                Bytes Written=100000000000
Hadoop Took5714212 ms
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$ hadoop fs -tail /user/output/part-r-00000
00000006666111111111EEEE
~~~~#iay1X        00000000000000000000000025D35EDF  6666AAAA55559999777700002222333388888FFFF999922220000
~~~~+@p){@        000000000000000000000000085426F4  77773333555511111111110000CCCC55559999AAAA7777DDDDDDDD
~~~~,R^_?n        0000000000000000000000001034E347  11111111999900001111888888AAAA55554444EEEE999933338888
~~~~.Ey`^)        000000000000000000000000016F0E66B  CCCC6666DDDD2222DDDD1111888889999EEEEEEEEEEEEBBBB4444
~~~~4!kA7x        0000000000000000000000001F1A1E26  EEEE777711117777BBBB1111EEEE88884444DDDDDDDDEEEEBBBB
~~~~8Ii/!@        0000000000000000000000001F05932F  11119999BBBB444477770000111114444CCCCAAAA6666DDDD0000
~~~~<I'5>F        00000000000000000000000008CB2293  88883333BBBB11116666999988885555888888888822228888CCCC
~~~~G-)m^)        0000000000000000000000013397F73  DDDDFFFFBBBBCCCCFFFF44446666AAAA111133333333AAAACCCC
~~~~c+I&cP        000000000000000000000000074BDF64  88880000555550000DDDD22227777AAAA000033332222AAAADDDD
~~~~hb&5X*        00000000000000000000000032C0E06B  7777BBBBBBBB9999EEEEAAAAAAAA0000CCCCDDDD4444BBBB4444
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
ubuntu@ip-172-31-13-32:/data/hadoop-2.7.2$
```

Terasort Spark Execution:

REST URL: spark://ec2-52-38-9-184.us-west-2.compute.amazonaws.com:6066 *(cluster mode)*
Alive Workers: 16
Cores in use: 32 Total, 0 Used
Memory in use: 42.7 GB Total, 0.0 B Used
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160329035526-172.31.10.245-36273	172.31.10.245:36273	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035526-172.31.14.75-39499	172.31.14.75:39499	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035526-172.31.3.192-51406	172.31.3.192:51406	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035526-172.31.6.39-45364	172.31.6.39:45364	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.0.118-44414	172.31.0.118:44414	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.1.123-45555	172.31.1.123:45555	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.1.49-37996	172.31.1.49:37996	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.11.169-53464	172.31.11.169:53464	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.13.220-50661	172.31.13.220:50661	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.14.121-55083	172.31.14.121:55083	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.2.106-49433	172.31.2.106:49433	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.4.142-44121	172.31.4.142:44121	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.7.96-33147	172.31.7.96:33147	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.8.168-40203	172.31.8.168:40203	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.8.178-50076	172.31.8.178:50076	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)
worker-20160329035527-172.31.8.60-60285	172.31.8.60:60285	ALIVE	2 (0 Used)	2.7 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160329060120-0000	Sort	32	2.4 GB	2016/03/29 06:01:20	root	FINISHED	10 min

Fig. Spark cluster setup with c3.large 16-node spot instances and execution for 100Gb dataset

a) Terasort Spark 10GB 1-Node C3.large performance:

```
ubuntu@ip-172-31-3-211: /data/spark-1.6.0-bin-hadoop2.6          root@ubuntu: ~/Downloads
16/03/27 08:00:01 INFO ShuffleBlockFetcherIterator: Getting 298 non-empty blocks out of 298 blocks
16/03/27 08:00:01 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 0 ms
16/03/27 08:00:02 INFO FileOutputCommitter: Saved output of task 'attempt_201603270749_0002_m_000295_891' to file:/data/output/_temporary/0/ta
sk_201603270749_0002_m_000295
16/03/27 08:00:02 INFO SparkHadoopMapRedUtil: attempt_201603270749_0002_m_000295_891: Committed
16/03/27 08:00:02 INFO Executor: Finished task 295.0 in stage 2.0 (TID 891). 2080 bytes result sent to driver
16/03/27 08:00:02 INFO TaskSetManager: Starting task 297.0 in stage 2.0 (TID 893, localhost, partition 297,NODE_LOCAL, 1956 bytes)
16/03/27 08:00:02 INFO TaskSetManager: Finished task 295.0 in stage 2.0 (TID 891) in 2593 ms on localhost (296/298)
16/03/27 08:00:02 INFO Executor: Running task 297.0 in stage 2.0 (TID 893)
16/03/27 08:00:02 INFO ShuffleBlockFetcherIterator: Getting 298 non-empty blocks out of 298 blocks
16/03/27 08:00:02 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 0 ms
16/03/27 08:00:03 INFO FileOutputCommitter: Saved output of task 'attempt_201603270749_0002_m_000296_892' to file:/data/output/_temporary/0/ta
sk_201603270749_0002_m_000296
16/03/27 08:00:03 INFO SparkHadoopMapRedUtil: attempt_201603270749_0002_m_000296_892: Committed
16/03/27 08:00:03 INFO Executor: Finished task 296.0 in stage 2.0 (TID 892). 2080 bytes result sent to driver
16/03/27 08:00:03 INFO TaskSetManager: Finished task 296.0 in stage 2.0 (TID 892) in 1583 ms on localhost (297/298)
16/03/27 08:00:03 INFO FileOutputCommitter: Saved output of task 'attempt_201603270749_0002_m_000297_893' to file:/data/output/_temporary/0/ta
sk_201603270749_0002_m_000297
16/03/27 08:00:03 INFO SparkHadoopMapRedUtil: attempt_201603270749_0002_m_000297_893: Committed
16/03/27 08:00:03 INFO Executor: Finished task 297.0 in stage 2.0 (TID 893). 2080 bytes result sent to driver
16/03/27 08:00:03 INFO TaskSetManager: Finished task 297.0 in stage 2.0 (TID 893) in 1164 ms on localhost (298/298)
16/03/27 08:00:03 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
16/03/27 08:00:03 INFO DAGScheduler: ResultStage 2 (saveAsTextFile at terasort.scala:26) finished in 346.912 s
16/03/27 08:00:03 INFO DAGScheduler: Job 1 finished: saveAsTextFile at terasort.scala:26, took 627.316088 s
Terasort run time53917 msecs
16/03/27 08:00:04 INFO SparkContext: Invoking stop() from shutdown hook
16/03/27 08:00:04 INFO SparkUI: Stopped Spark web UI at http://172.31.3.211:4040
16/03/27 08:00:04 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
16/03/27 08:00:04 INFO MemoryStore: MemoryStore cleared
16/03/27 08:00:04 INFO BlockManager: BlockManager stopped
16/03/27 08:00:04 INFO BlockManagerMaster: BlockManagerMaster stopped
16/03/27 08:00:04 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
16/03/27 08:00:04 INFO SparkContext: Successfully stopped SparkContext
16/03/27 08:00:04 INFO ShutdownHookManager: Shutdown hook called
16/03/27 08:00:04 INFO ShutdownHookManager: Deleting directory /tmp/spark-923a5a1c-f907-4e65-87a2-13403b3c485f/httpd-453cf9ff-e537-4118-891c-3
979601e6f5c
16/03/27 08:00:04 INFO ShutdownHookManager: Deleting directory /tmp/spark-923a5a1c-f907-4e65-87a2-13403b3c485f
16/03/27 08:00:04 INFO RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
ubuntu@ip-172-31-3-211:/data/spark-1.6.0-bin-hadoop2.6$
```

Fig. Shown execution time for spark application on 1Gb dataset. Execution time is comparable to hadoop for small datasets.

Terasort Spark 10GB 1-Node top-tail sorted records.

```
ubuntu@ip-172-31-3-211: /data/output          root@ubuntu: ~/Downloads
part-00014  part-00042  part-00070  part-00098  part-00126  part-00154  part-00182  part-00210  part-00238  part-00266  part-00294
part-00015  part-00043  part-00071  part-00099  part-00127  part-00155  part-00183  part-00211  part-00239  part-00267  part-00295
part-00016  part-00044  part-00072  part-00100  part-00128  part-00156  part-00184  part-00212  part-00240  part-00268  part-00296
part-00017  part-00045  part-00073  part-00101  part-00129  part-00157  part-00185  part-00213  part-00241  part-00269  part-00297
part-00018  part-00046  part-00074  part-00102  part-00130  part-00158  part-00186  part-00214  part-00242  part-00270  _SUCCESS
part-00019  part-00047  part-00075  part-00103  part-00131  part-00159  part-00187  part-00215  part-00243  part-00271
part-00020  part-00048  part-00076  part-00104  part-00132  part-00160  part-00188  part-00216  part-00244  part-00272
part-00021  part-00049  part-00077  part-00105  part-00133  part-00161  part-00189  part-00217  part-00245  part-00273
part-00022  part-00050  part-00078  part-00106  part-00134  part-00162  part-00190  part-00218  part-00246  part-00274
part-00023  part-00051  part-00079  part-00107  part-00135  part-00163  part-00191  part-00219  part-00247  part-00275
part-00024  part-00052  part-00080  part-00108  part-00136  part-00164  part-00192  part-00220  part-00248  part-00276
part-00025  part-00053  part-00081  part-00109  part-00137  part-00165  part-00193  part-00221  part-00249  part-00277
part-00026  part-00054  part-00082  part-00110  part-00138  part-00166  part-00194  part-00222  part-00250  part-00278
part-00027  part-00055  part-00083  part-00111  part-00139  part-00167  part-00195  part-00223  part-00251  part-00279
ubuntu@ip-172-31-3-211:/data/output$ touch spark-10gb-1Node-records
ubuntu@ip-172-31-3-211:/data/output$ head part-00000 >> spark-10gb-1Node-records
ubuntu@ip-172-31-3-211:/data/output$ tail part-00297 >> spark-10gb-1Node-records
ubuntu@ip-172-31-3-211:/data/output$ cat spark-10gb-1Node-records
    "O!uve  000000000000000000000000001228D4  7777888800002222444DDDDDDDDEEEE00000000CCCC7777DDDD
    PMd32=  00000000000000000000000003440CC1  FFFFEEEE6666CCCCBBBB999933335555DDDDDDDD777788886666
    ^3CO],  0000000000000000000000000158C5C5  5555AAAA9999EEEE888822229999CCCCDDDD6666555544442222
   !&S3/]]  00000000000000000000000002145D78  8888BBBBDDDD1111CCCC55556666BBBB1111EEEEDDDD22229999
   !,=U#,9  000000000000000000000000019072E3  33332222FFFFBBBB0000FFFFAAAA6666555533333DDDD3333CCCC
   !0f[ITd  00000000000000000000000003CAAB4B  9999FFFF555533337777CCCC4444BBBB7777EEEEBBBBDDDD4444
   !f6Suy2  00000000000000000000000003ABFD84  EEEE555555556666AAAA5555BBBBDDDD0000111166660000DDDD
   #%NIpq.  00000000000000000000000003B36FB9  11110000333344441111666666666AAAAAAAA00001111CCCCEEEE
   #'^cl'~  00000000000000000000000002EDC5C8  8888AAAA11114444FFFF77773333EEEE44440000FFFF99999999
   $"-'Q)]  00000000000000000000000005F1265D  CCCC6666EEEE22220000DDDDAAAA88886666BBBB00006666AAAA
~~~uq2k#=U  00000000000000000000000002C06745  99991111DDDD222211110000FFFFEEEEFFFF33337777CCCC2222
~~~v/0&Qnm  00000000000000000000000004709701  CCCC88883333FFFF000000000000099991111FFFF777744446666
~~~yKOl:gE  00000000000000000000000002048B4F  CCCC111144448888222266666BBBB888855557777EEEEBBBB0000
~~~yK^H.il  0000000000000000000000000463D004  44440000FFFF33339999444477777DDDDFFFFAAAA11118888DDDD
~~~yL;C'XE  00000000000000000000000005B0D211  2222EEEE3333000022221111CCCCFFFF555577774444BBBB6666
~~~zbA_ Tt  000000000000000000000000007F9F4F  BBBBCCCC666655559999FFFF8888AAAA11116666AAAABBBB0000
~~~zeO^FEg  00000000000000000000000001E06130  4444CCCCBBBB99992222888855558888CCCCFFFF000011111111
~~~}GxjWHI  000000000000000000000000000CA1345  777711118888AAAAAAAA22221111BBBB00002222BBBBCCCC2222
~~~}P;]g0g  000000000000000000000000040DA3E4  4444FFFF444466663333EEEE88888888DDDDEEEE44442222DDDD
~~~}kU|K<p  00000000000000000000000005E4A0AA  0000666655551111BBBB88889999AAAA55550000333355557777
ubuntu@ip-172-31-3-211:/data/output$ clear
```

Fig. Shown execution time for spark application on 10Gb dataset. Execution time is 10x faster than hadoop for large datasets on cluster instances.

b) Terasort Spark 100GB 16-Node C3.large performance:

```
16/03/27 03:05:13 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:14 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:15 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:16 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:17 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:18 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:19 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:20 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:21 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:22 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:23 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:24 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:25 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:26 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:27 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:28 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:29 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:30 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:31 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:32 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:33 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:34 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:35 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:36 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:37 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:38 INFO yarn.Client: Application report for application_1459040781286_0002 (state: RUNNING)
16/03/27 03:05:39 INFO yarn.Client: Application report for application_1459040781286_0002 (state: FINISHED)
16/03/27 03:05:39 INFO yarn.Client:
         client token: N/A
         diagnostics: N/A
         ApplicationMaster host: 172.31.9.219
         ApplicationMaster RPC port: 0
         queue: default
         start time: 1459043587934
         final status: SUCCEEDED
         tracking URL: http://masterNode:8088/proxy/application_1459040781286_0002/
         user: ubuntu
16/03/27 03:05:39 INFO util.ShutdownHookManager: Shutdown hook called
16/03/27 03:05:39 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-9af3bae1-a0ff-4143-98ca-6c56dadd780b
Time taken 2961780 ms
```

fig: Terasort Spark 100GB 16-Node: as shown, YARN resource manager is used instead of standalone. Instead of spark standalone cluster manager, we are using YARN to benefit from HDFS to distribute data across network

```
ubuntu@ip-172-31-2-146: /data/output | ubuntu@ip-172-31-9-219: /data/hadoop-... | ubuntu@ip-172-31-15-157: /data/hadoop... | root@ubuntu: ~/Downloads
part-00042  part-00110  part-00178  part-00246  part-00314  part-00382  part-00450  part-00518  part-00586  part-00654  part-00722
part-00043  part-00111  part-00179  part-00247  part-00315  part-00383  part-00451  part-00519  part-00587  part-00655  part-00723
part-00044  part-00112  part-00180  part-00248  part-00316  part-00384  part-00452  part-00520  part-00588  part-00656  part-00724
part-00045  part-00113  part-00181  part-00249  part-00317  part-00385  part-00453  part-00521  part-00589  part-00657  part-00725
part-00046  part-00114  part-00182  part-00250  part-00318  part-00386  part-00454  part-00522  part-00590  part-00658  part-00726
part-00047  part-00115  part-00183  part-00251  part-00319  part-00387  part-00455  part-00523  part-00591  part-00659  part-00727
part-00048  part-00116  part-00184  part-00252  part-00320  part-00388  part-00456  part-00524  part-00592  part-00660  part-00728
part-00049  part-00117  part-00185  part-00253  part-00321  part-00389  part-00457  part-00525  part-00593  part-00661  part-00729
part-00050  part-00118  part-00186  part-00254  part-00322  part-00390  part-00458  part-00526  part-00594  part-00662  part-00730
part-00051  part-00119  part-00187  part-00255  part-00323  part-00391  part-00459  part-00527  part-00595  part-00663  part-00731
part-00052  part-00120  part-00188  part-00256  part-00324  part-00392  part-00460  part-00528  part-00596  part-00664  part-00732
part-00053  part-00121  part-00189  part-00257  part-00325  part-00393  part-00461  part-00529  part-00597  part-00665  part-00733
part-00054  part-00122  part-00190  part-00258  part-00326  part-00394  part-00462  part-00530  part-00598  part-00666  part-00734
part-00055  part-00123  part-00191  part-00259  part-00327  part-00395  part-00463  part-00531  part-00599  part-00667  part-00735
part-00056  part-00124  part-00192  part-00260  part-00328  part-00396  part-00464  part-00532  part-00600  part-00668  part-00736
part-00057  part-00125  part-00193  part-00261  part-00329  part-00397  part-00465  part-00533  part-00601  part-00669  part-00737
part-00058  part-00126  part-00194  part-00262  part-00330  part-00398  part-00466  part-00534  part-00602  part-00670  part-00738
part-00059  part-00127  part-00195  part-00263  part-00331  part-00399  part-00467  part-00535  part-00603  part-00671  part-00739
part-00060  part-00128  part-00196  part-00264  part-00332  part-00400  part-00468  part-00536  part-00604  part-00672  part-00740
part-00061  part-00129  part-00197  part-00265  part-00333  part-00401  part-00469  part-00537  part-00605  part-00673  part-00741
part-00062  part-00130  part-00198  part-00266  part-00334  part-00402  part-00470  part-00538  part-00606  part-00674  part-00742
part-00063  part-00131  part-00199  part-00267  part-00335  part-00403  part-00471  part-00539  part-00607  part-00675  part-00743
part-00064  part-00132  part-00200  part-00268  part-00336  part-00404  part-00472  part-00540  part-00608  part-00676  part-00744
part-00065  part-00133  part-00201  part-00269  part-00337  part-00405  part-00473  part-00541  part-00609  part-00677  spark-100gb-records
part-00066  part-00134  part-00202  part-00270  part-00338  part-00406  part-00474  part-00542  part-00610  part-00678  _SUCCESS
part-00067  part-00135  part-00203  part-00271  part-00339  part-00407  part-00475  part-00543  part-00611  part-00679  valsort
ubuntu@ip-172-31-2-146:/data/output$
ubuntu@ip-172-31-2-146:/data/output$
ubuntu@ip-172-31-2-146:/data/output$ ./valsort part-00000
Records: 1103383
Checksum: 86d5823c3388a
Duplicate keys: 0
SUCCESS - all records are in order
ubuntu@ip-172-31-2-146:/data/output$ ./valsort part-00744
Records: 1476079
Checksum: b40f74e4a20df
Duplicate keys: 0
SUCCESS - all records are in order
ubuntu@ip-172-31-2-146:/data/output$
```

fig: Terasort Spark 100GB 16-Node Valsort checksum & Records:

Conclusion:

1. On small dataset of size 1GB to 5Gb, Shared-memory and Hadoop seems to take same running time to sort data. In shared-memory, working with more thread multiprocessing, doesn't benefit unless CPU cores are limited. So on average, Shared-Memory follows same analogy as MapReduce by Hadoop. So the time to sort file size up to 10Gb takes same amount of time on both Shared-Memory and Hadoop Terasort application. This is due to bottleneck generated by large disk read-writes.
2. Dataset of size 10GB and 100GB on Spark and hadoop application varies due to reduced disk read-write in spark application. As shown in graph, for both 10Gb & 100Gb dataset, Spark executes and sorts file on average 10x faster than Hadoop for same dataset. This is great improvement in Spark due to RDD partitioning and tuning in RDD partitioning. Also Spark stores RDD's into memory which greatly increases the performances.
3. On both 1-Node & 16-Node cluster with 10Gb, and 100GB dataset, Spark has shown tremendous improvements in sorting datafile, which is 10x faster than Hadoop or Shared-Memory application. This leads to conclusion that, on increasing the cluster size and dataset size, like 1Tb, 10TB, or 100TB hadoop will involve much larger disk read-write operations than Spark application. So Spark is best and fastest parallel programming model for large datasets and varying cluster sizes.