**Index**

1. Problem Statement
2. Methodology
   * Shared Memory Sort
   * Hadoop single node setup
   * Hadoop multiple node (cluster) setup
   * Spark
3. Results & Conclusion
   * Table shared memory evaluation
   * Table 10 GB comparison
   * Table 100 GB comparison
   * Speed up chart
4. References
5. Screenshots
   * D2.xlarge 1 node
     1. 1GB Hadoop Start
     2. 1GB Hadoop End
     3. 1GB Hadoop valsort
     4. 1GB Spark Start
     5. 1GB Spark End
     6. 1GB Spark valsort
   * C3.large 1 node
     1. 10GB Shared Memory
     2. 10GB Shared Memory valsort
     3. 10GB Hadoop Start
     4. 10GB Hadoop End
     5. 10GB Hadoop valsort
     6. 10GB Spark Start
     7. 10GB Spark End
     8. 10GB Spark valsort
   * C3.large 16 node
     1. 100GB Hadoop Start
     2. 100GB Hadoop End
     3. 100GB Hadoop valsort
     4. 100GB Spark Start
     5. 100GB Spark End
     6. 100GB Spark valsort

**Problem Statement:**

This programming assignment covers the TeraSort application implemented in 3 different ways: Java, Hadoop, and Spark. In order to carry out the implementation on Amazon Web Service, the data for sorting will be generated on demand when the sorting needs to be performed.

**Methodology:**

Runtime environment setting:

OS used: Linux

Linux distribution:

Java version: 1.7

Hadoop version: 2.7

Spark version: 1.6

MPI version: NA

I have used an automation script which would login to each of the slave hosts and execute the commands for initial setup. The teach.sh script works like an instructor that gives commands to each of the host in the list. After executing commands on one host, the host is removed from list. The commands can be provided in a separate command file . The teach.sh script reads the hostname from the file “list”, and reads the command from “command”.

**Shared Memory sort:**

In This method I have implemented quicksort using java. By using quick sort we are able to achieve performance of the order O(n log n). Yet another advantage of using quick sort is that it sorts in place. We do not require external storage for sorting the data. Since the data we choose to sort is a huge volume it is advantageous to save space by the use of quick sort.

Furthermore, to improve performance, i have implemented multithreading. The objective here is to sort data chunk that is larger than memory. This requires splitting of input data into smaller chunks. Using multithreading we submit each thread to operate the sorting logic over a chunk.

For the merge, i created a arraylist which stores the first element of each sorted chunk. The minimum element is fetched out of this and written to the output file. The arraylist elemt is replaced by the next element of the same file. This way we read the data only once and write only once.

(1 Disk Read Write). This is a n way merge and has time complexity of O [n(log n)].

**Hadoop setup on local :**

I am using single node in pseudo-distributed mode. Installation steps for setup of cluster to run a MapReduce job locally-

1. Format the filesystem: bin/hdfs namenode -format
2. Start NameNode daemon and DataNode daemon: sbin/start-dfs.sh
3. Browse the web interface for the NameNode: <http://localhost:50070/>
4. Make the HDFS directories to execute MapReduce jobs: bin/hdfs dfs -mkdir /user/hduser
5. Copy the input files into the distributed filesystem: bin/hdfs dfs -put etc/hadoop input
6. Run a MapReduce job
7. We can examine the output by getting it to local filesystem by using : bin/hdfs dfs -get output output or we can view the files on HDFS by using :bin/hdfs dfs -cat output/\*
8. stop the daemons: sbin/stop-dfs.sh

**Hadoop single node setup on AWS:**

Hadoop’s Java configuration is driven by two types of important configuration files:

* Read-only default configuration - core-default.xml, hdfs-default.xml, yarn-default.xml and mapred-default.xml.
* Site-specific configuration - core-site.xml, hdfs-site.xml, yarn-site.xml and mapred-site.xml.

The Hadoop namenode uses fs.default.name to read hostname and port number.

**Hadoop Multiple node (cluster) setup:**

To configure the Hadoop cluster we configure the environment in which the Hadoop daemons execute as well as the configuration parameters for the Hadoop daemons.

HDFS daemons are NameNode, SecondaryNameNode, and DataNode. YARN damones are ResourceManager, NodeManager, and WebAppProxy. For MapReduce the MapReduce Job History Server will also be running.

The data passed to the Mapper is specified by an InputFormat. The InputFormat is specified in the driver code. It defines the location of the input data like a file or directory on HDFS. It also determines how to split the input data into input splits.

Description of the function of each file, and what modifications you had to make to go from 1 node to multiple nodes. Please answer the following questions:

1) What is a Master node? What is a Slaves node?

2) Why do we need to set unique available ports to those configuration files on a shared environment?

What errors or side-effects will show if we use same port number for each user?

3) How can we change the number of mappers and reducers from the configuration file?

**Results:**

**Shared Memory Sort**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size | 1 Thread | 2 Threads | 4 Threads | 8 Threads | Throughput |
| 10 GB (C3.large) | 42min 12 sec | 39min 09 sec | 41 min 17 sec | 46min 44sec | 3.9 MB/sec |
| 1 GB (D2.xlarge) | 8 min 22 sec | 7 min 52 sec | 6 min 28 sec | 6 mins 59 sec | 3.5 MB/sec |

8

7

6

5

4

3

2

1

1 2 4 8

threads

Time in mins

D2.xlarge

4 cores

C3.large

2 cores

50

48

46

44

42

40

38

36

34

.

.

0

1 2 4 8

threads

Time in mins

**Comparison** between the SharedMemory to the Hadoop and Spark for **1 node**

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Shared Memory | Hadoop | Spark |
| 10 GB (c3.large) | 37 mins 09 sec | 21mins | 16mins |
| 1 GB (D2.xlarge) | 6 mins 28 sec | 1 min 15 sec | 1 min 8 sec |

7

6:30

6

5:30

5

4:30

4

3:30

3

2:30

2

1:30

1

Time

min:sec

D2.xlarge

1GB

-Shared Memory

-Hadoop

-Spark

40

36

32

28

24

20

16

12

8

4

0

Time

min:sec

C3.large

10 GB

**Comparison** between the SharedMemory to the Hadoop and Spark for **1 node**

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Shared Memory (1 node) | Hadoop | Spark |
| 100 GB (c3.large) | (37 mins 09 sec)\*10 | 1 hr 26 mins | 25 mins |

-Hadoop

-Spark

100

90

80

70

60

70

60

50

40

30

20

10

0

Time

min

C3.large

100 GB

**Comparison** between the throughput of SharedMemory , Hadoop and Spark for **c3.large**

|  |  |  |  |
| --- | --- | --- | --- |
| Throughput | Shared Memory | Hadoop | Spark |
| 1 node | 3.9 MB/sec | 7.9 MB/sec | 10.4 MB/sec |
| 16 node | 3.9 MB/sec | 19.4 MB/sec | 66.6 MB/sec |

**Graph showing execution speedup 1 node and 16 nodes**

12

10

8

6

4

2

0

Throughput

C3.large

1node

-Shared Memory

-Hadoop

-Spark

70

60

50

40

30

20

10

0

C3.large

16 nodes

Throughput

**Conclusion:**

Justification for the difference in performance:

Hadoop has HDFS, Spark does not have it. Spark runs on top of Hadoop’s DFS. in Hadoop every read write is a disk operation, whereas in case of spark, Disk IO is minimised by utilising the memory space. This makes Sark perform faster than Hadoop.

After completion of Hadoop on 1 node first, we have to modify these configuration

files to execute multi-node run:

1. conf/master --
2. conf/slaves -- add the ip of all slave nodes so that data can be distributed to these.
3. conf/core-site.xml -- mention the url of the master node
4. conf/hdfs-site.xml -- mention the path for tmp files
5. conf/mapred-site.xml -- set the variables for number of workers/mappers/reducers

1) What is a Master node? What is a Slave node?

The master node is where the namenode runs. It controls the execution. It distributes the task to slave nodes. Slave nodes is where the data gets distributed and the execution is implemented. When there are several slaves the processing of data is distributed and executed parallely.

The job tracker runs on the master node whereas the task tracker runs on slave nodes.

2) Why do we need to set unique available ports to those configuration files on a shared environment? What errors or side-effects will show if we use same port number for each user?

The port defines where the result is transfered. If various user use same port the data could get over written and we would not get the expected output.

3) How can we change the number of mappers and reducers from the configuration File?

We can add this property in the mapred-site.xml. At startup the configuration properties are read from this file and accordingly the number of mapper/reducer deamons are launched.

Which seems to be best at 1 node scale? How about 16 nodes? Can you predict which would be best at 100 node scale? How about 1000 node scales?

Spark seems to scale best in all the tests. Even when we increase the number of slave nodes, the spark implementation is much faster. From the speedup graph we can see that when the number of nodes is increased we observe a multitude of speedup. Hence Spark would be best suited for all the distributed implementations.

**Comparison of results with those from the Sort Benchmark who used Hadoop and Spark:**

Winner Hadoop Throughput = 1.4 TB/min [64 GB memory, 12x3TB disks]

Personal implementation: 11.4 GB/min

Winner Spark Throughput = 4.2 TB/min [244GB memory, 8x800 GB SSD]

Personal implementation: 39.9 GB/min

There is a huge difference in the performance as compared to the benchmark. This can be attributed to the type of instance used. The c3.large instance has only 2 cores. Also the processing was done with memory size of 1GB which is very small as compared to the benchmark memory of 64 and 244 GB.

Secondly the number of slave nodes considered for this experiment was very very small ie 16 as compared to the 1200 nodes setup used for the benchmark.

**The CloudSort benchmark** offers a standardised procedure to measure and compare various implementations of sorting. Sorting is used as a standard benchmark test since the IOs in sorting are in close approximation with the workload of real world applications which are IO intensive. Different algorithms can be implemented over public clouds and the cost can be compared. This is a good method for comparison because the results are from the same kind of hardware so the anomaly reduces.

**References:**

<http://www.algolist.net/Algorithms/Sorting/Quicksort>

<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html>

<http://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm>

<http://hortonworks.com/hadoop-tutorial/introducing-apache-hadoop-developers/>

<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html>

<http://spark.apache.org/docs/latest/index.html>

<http://spark.apache.org/docs/latest/programming-guide.html>

<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/raid-config.html>

<https://www.researchgate.net/profile/Umar_Farooq_Minhas/publication/282511568_Clash_of_the_titans/links/561806b208ae044edbad0a00.pdf>

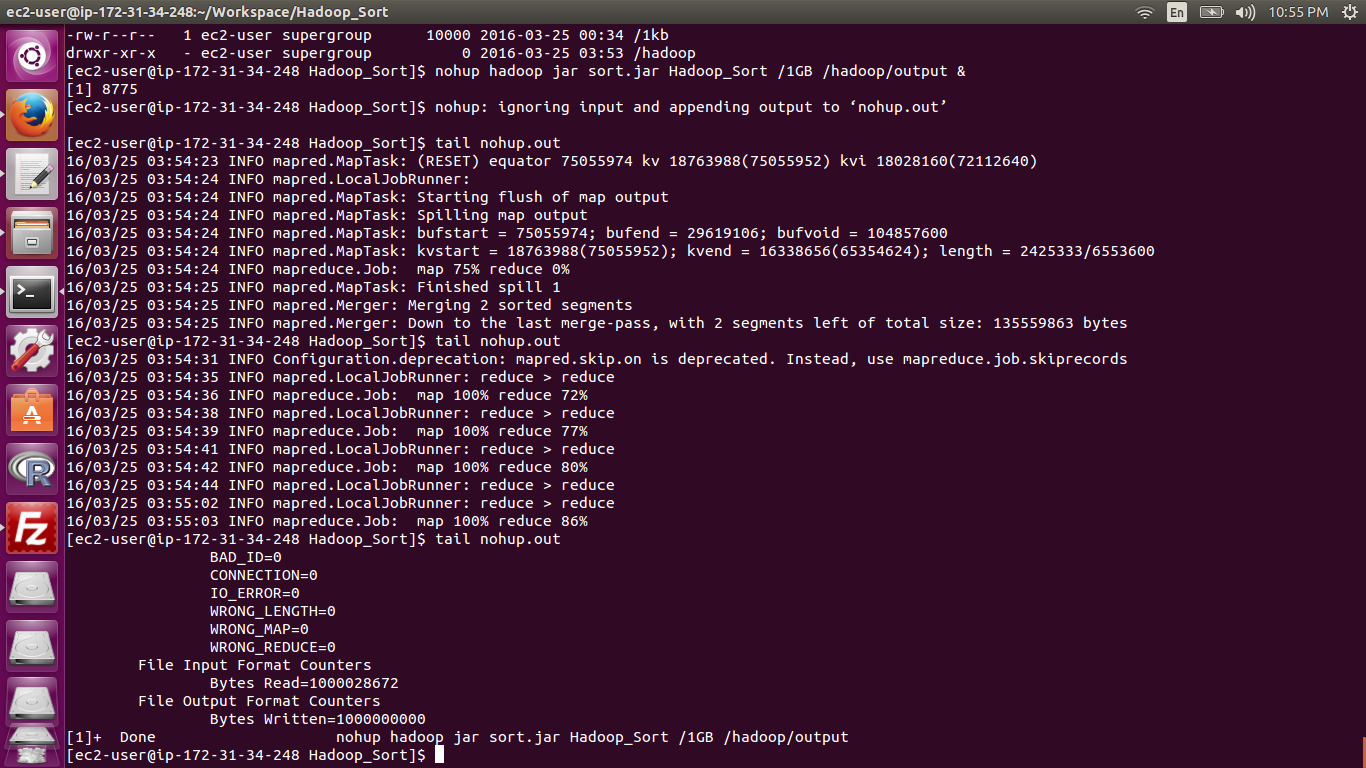
**Screenshots:**

**D2.xlarge**

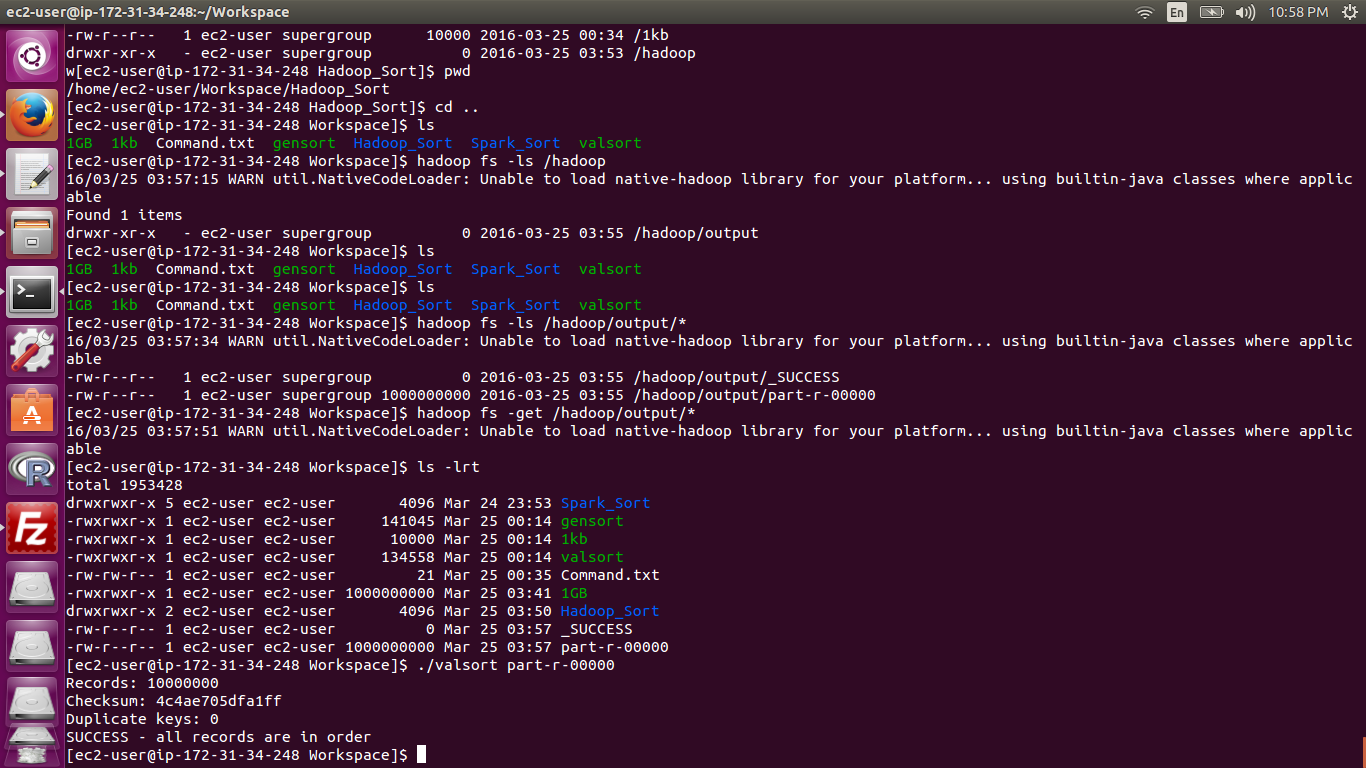
**1GB Hadoop Start**



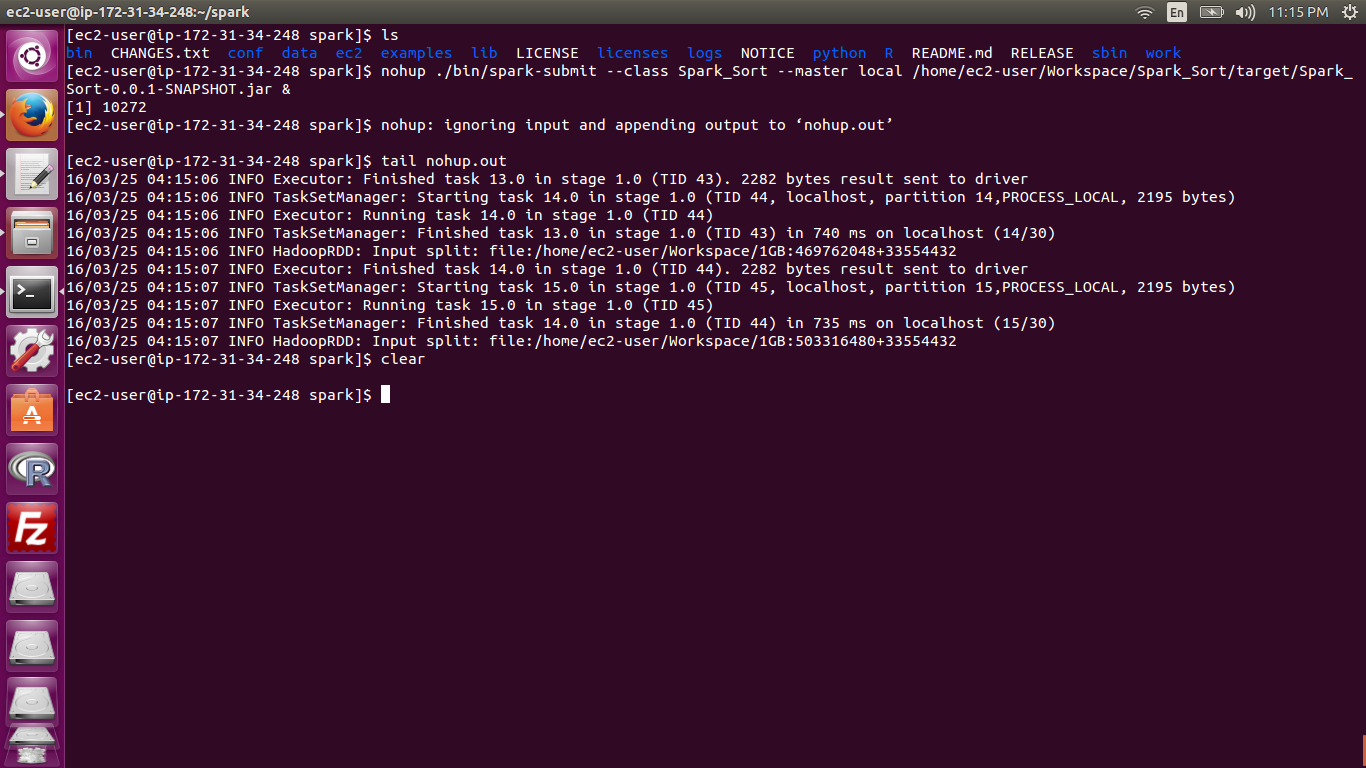
**1GB Hadoop End**



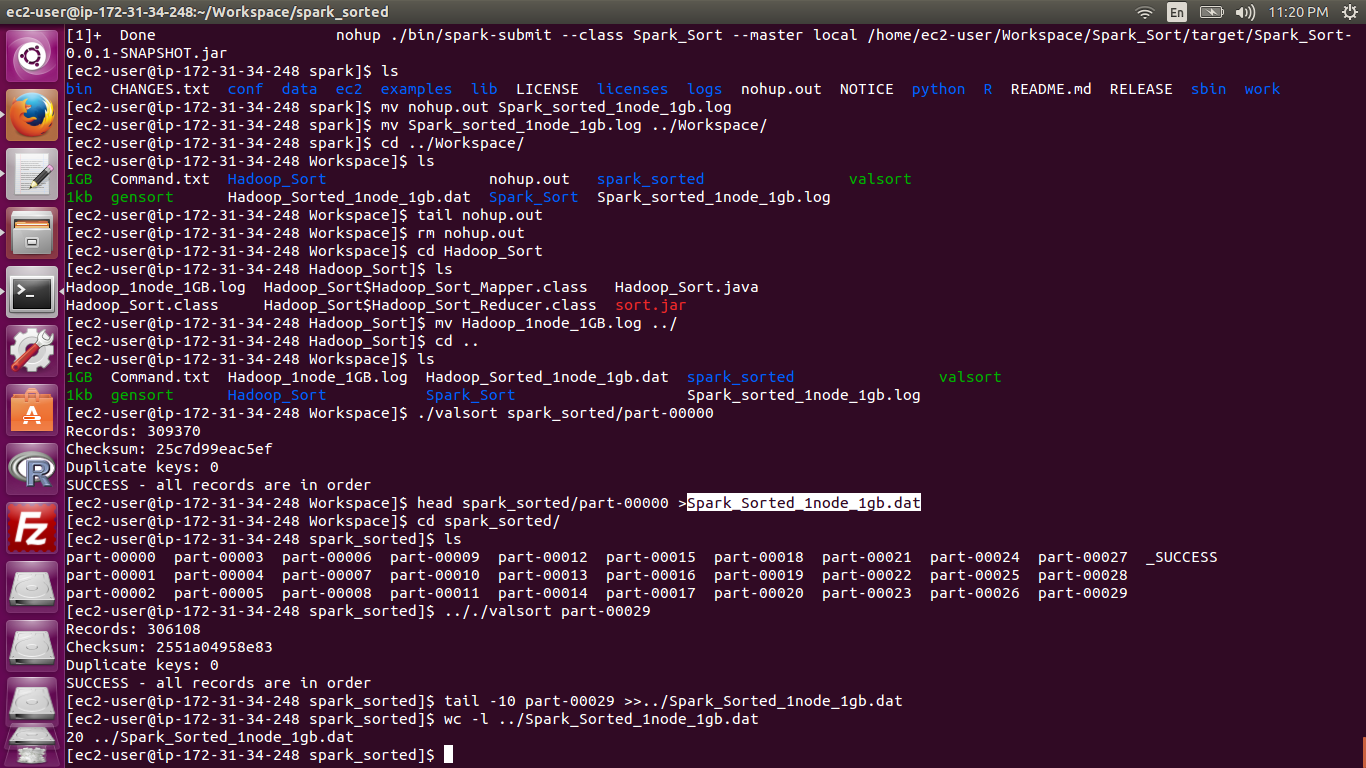
**1GB Hadoop valsort**



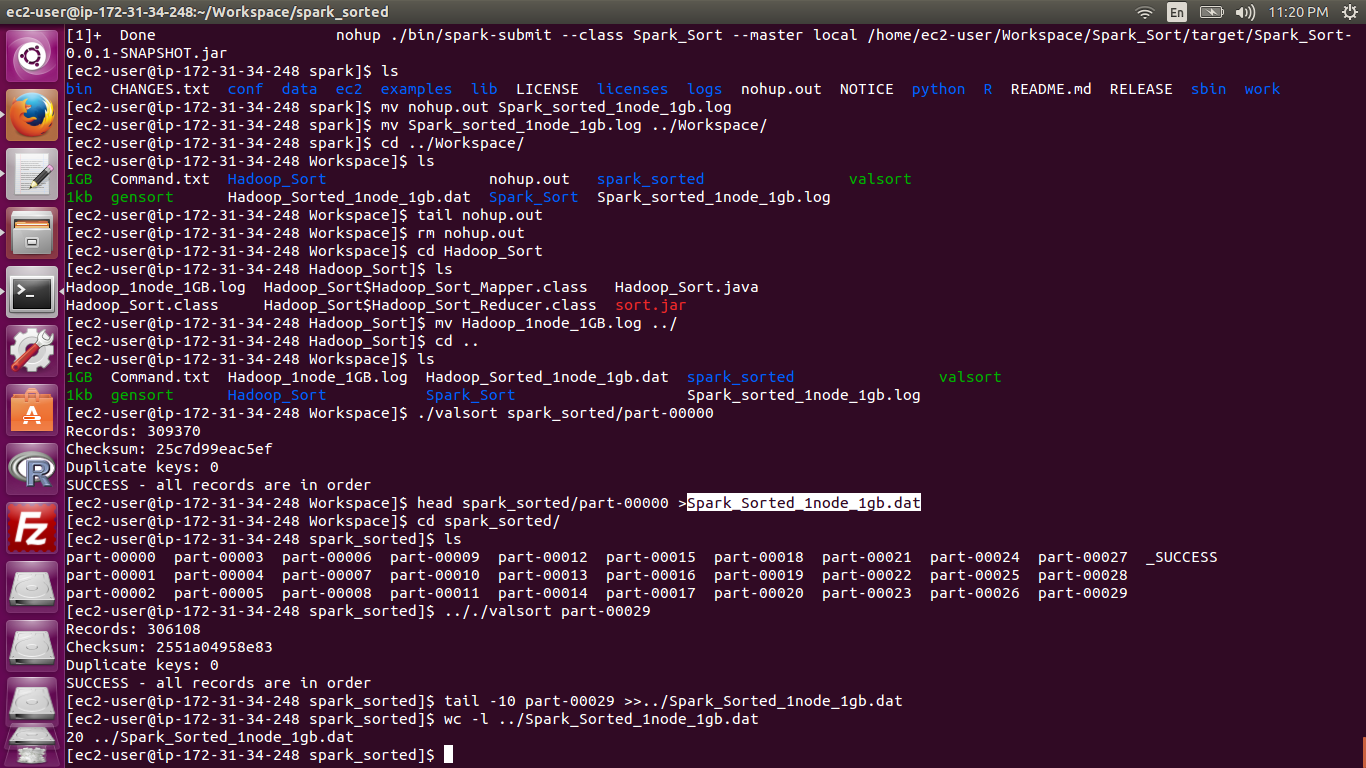
**1GB Spark Start**



**1GB Spark End**

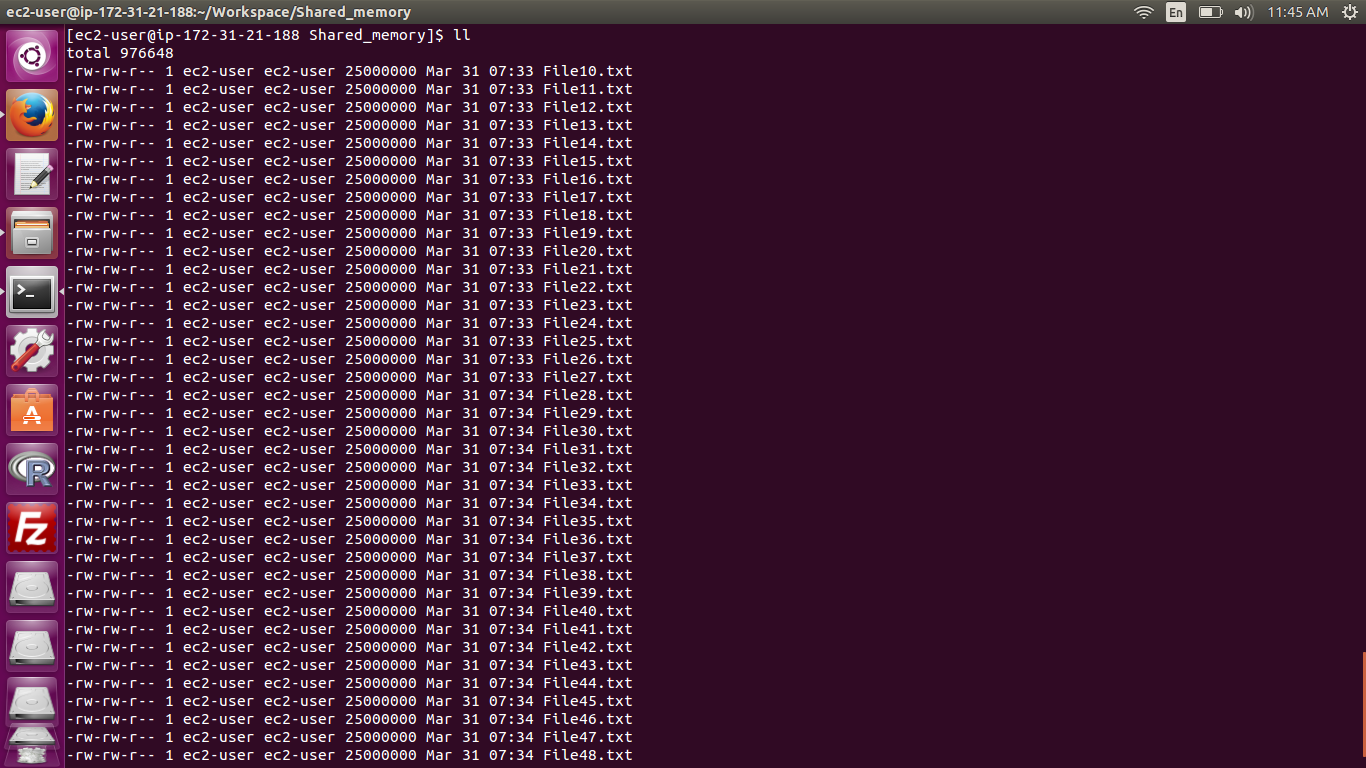


**1GB Spark valsort**

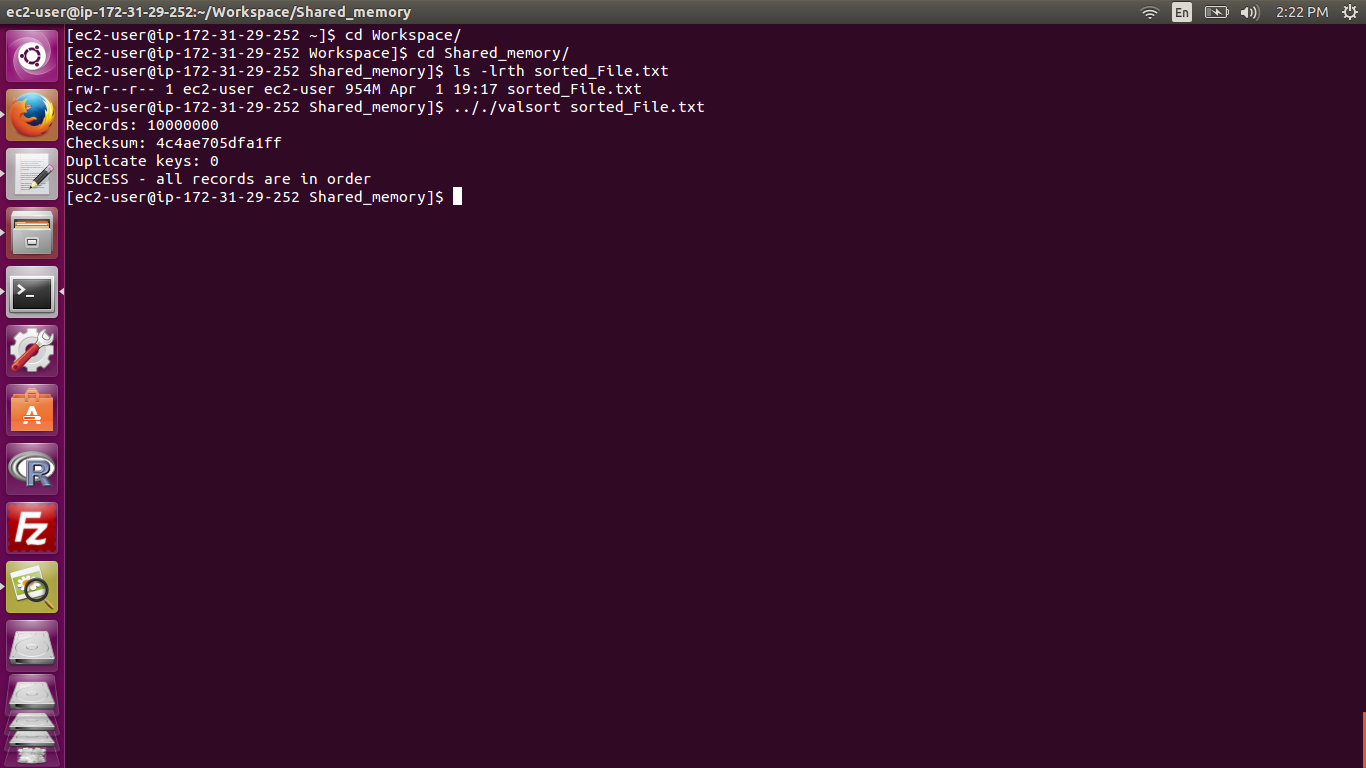


**c3.large**

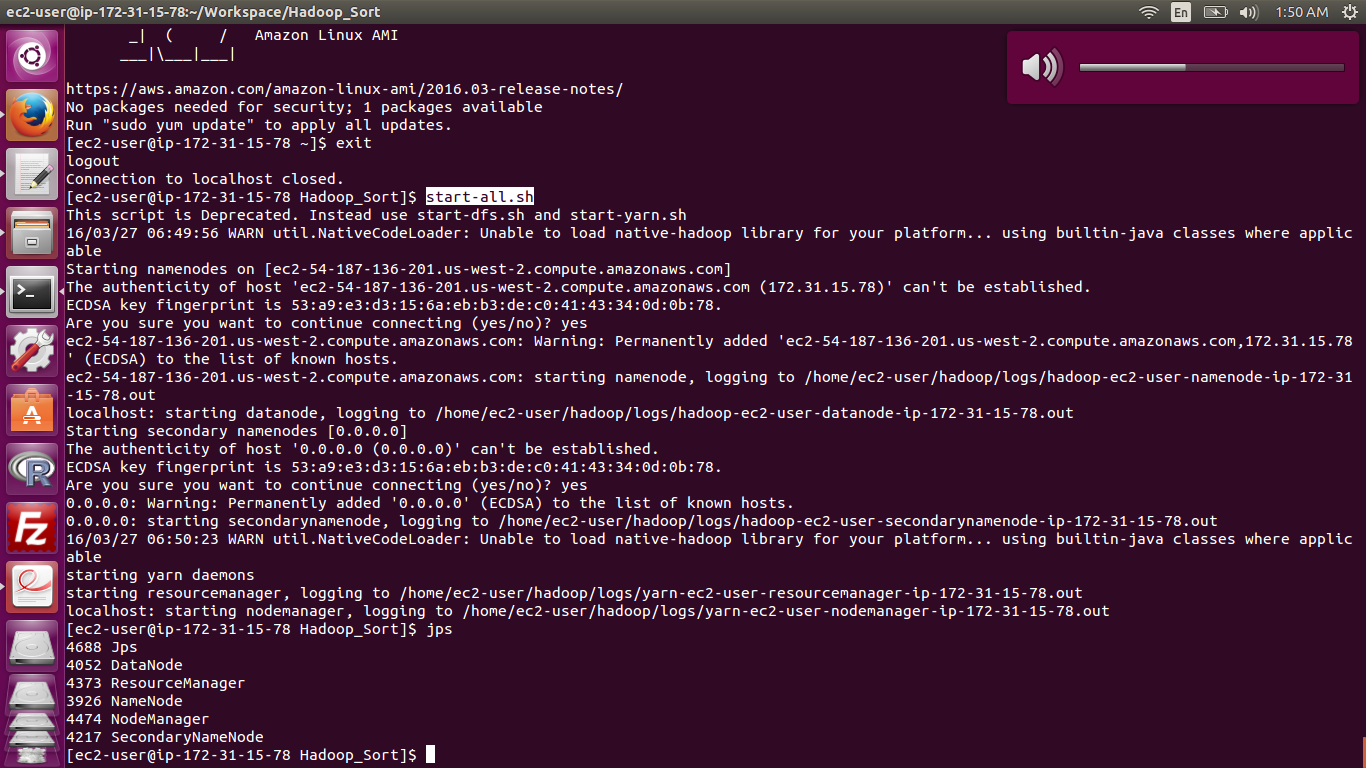
**10GB Shared Memory**



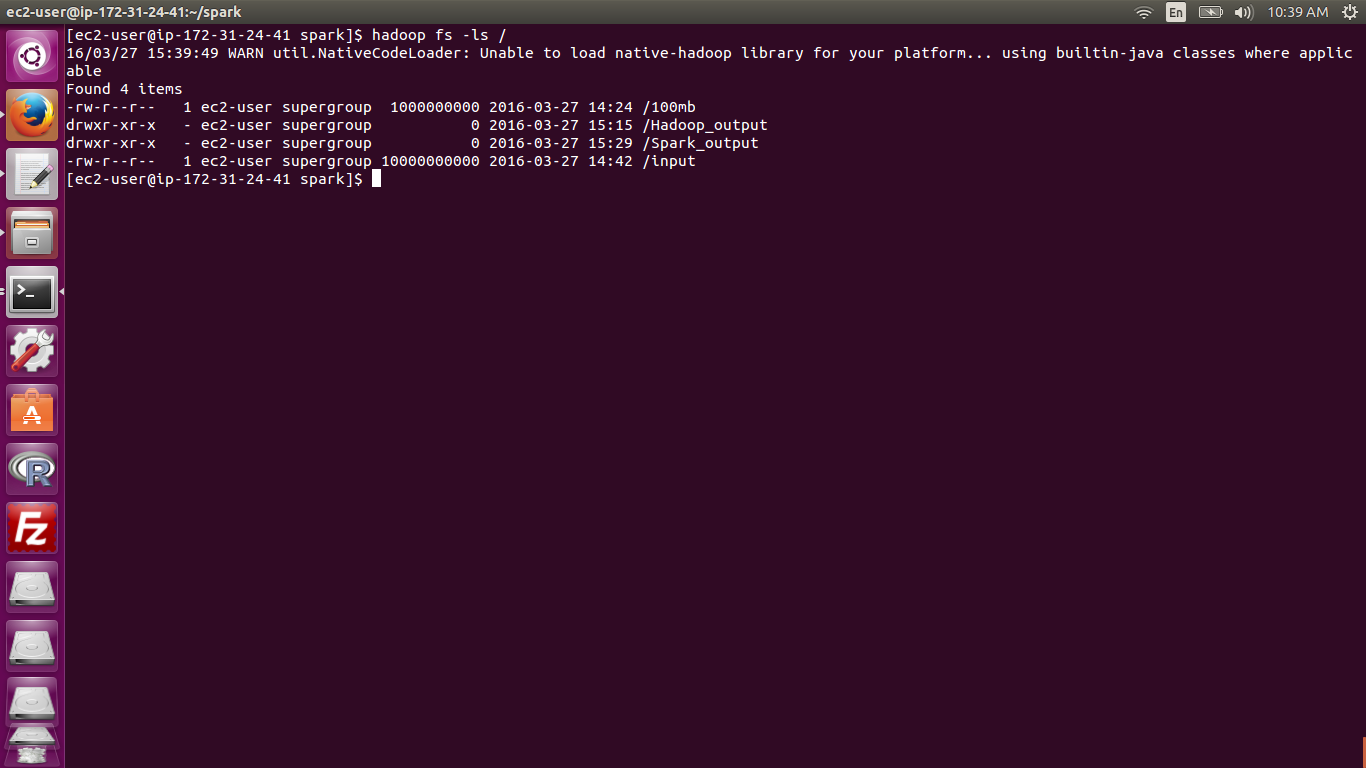
**10GB Shared Memory valsort**



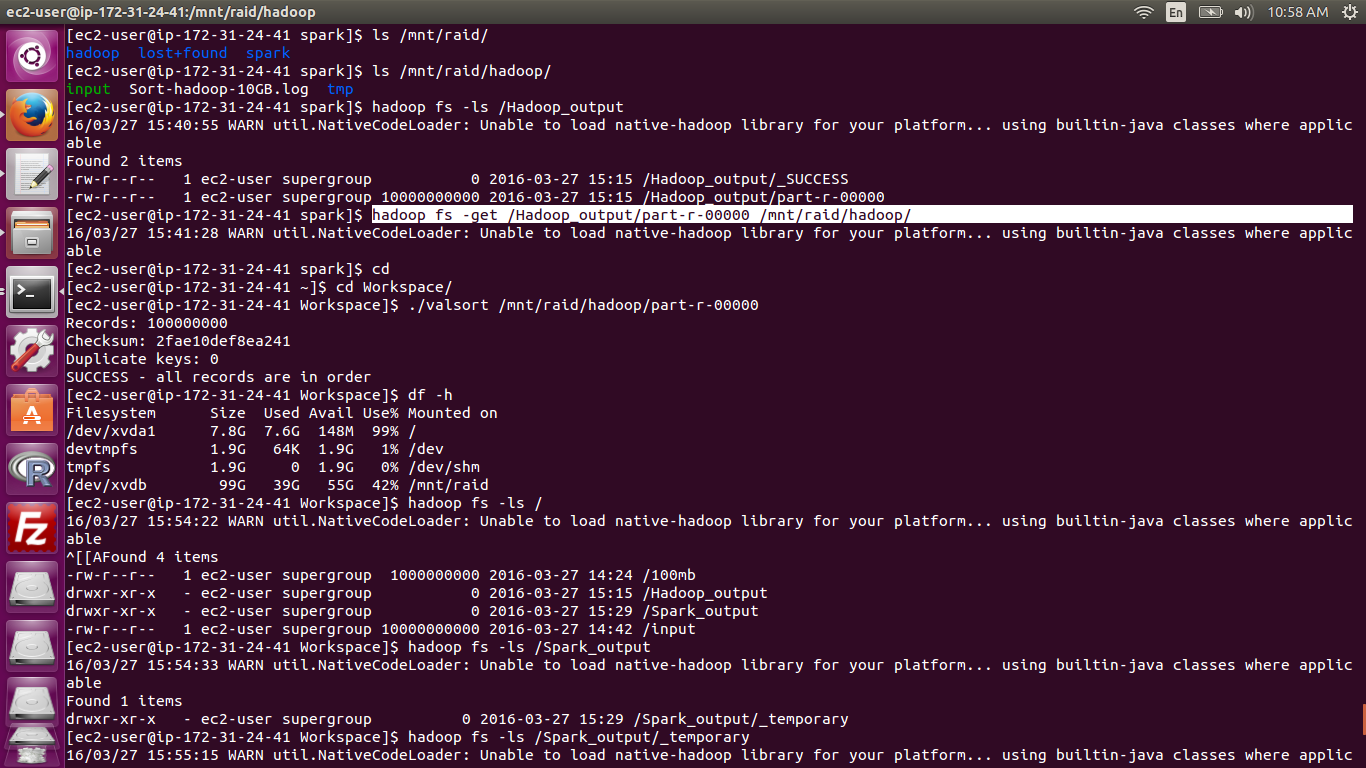
**10GB Hadoop Start**



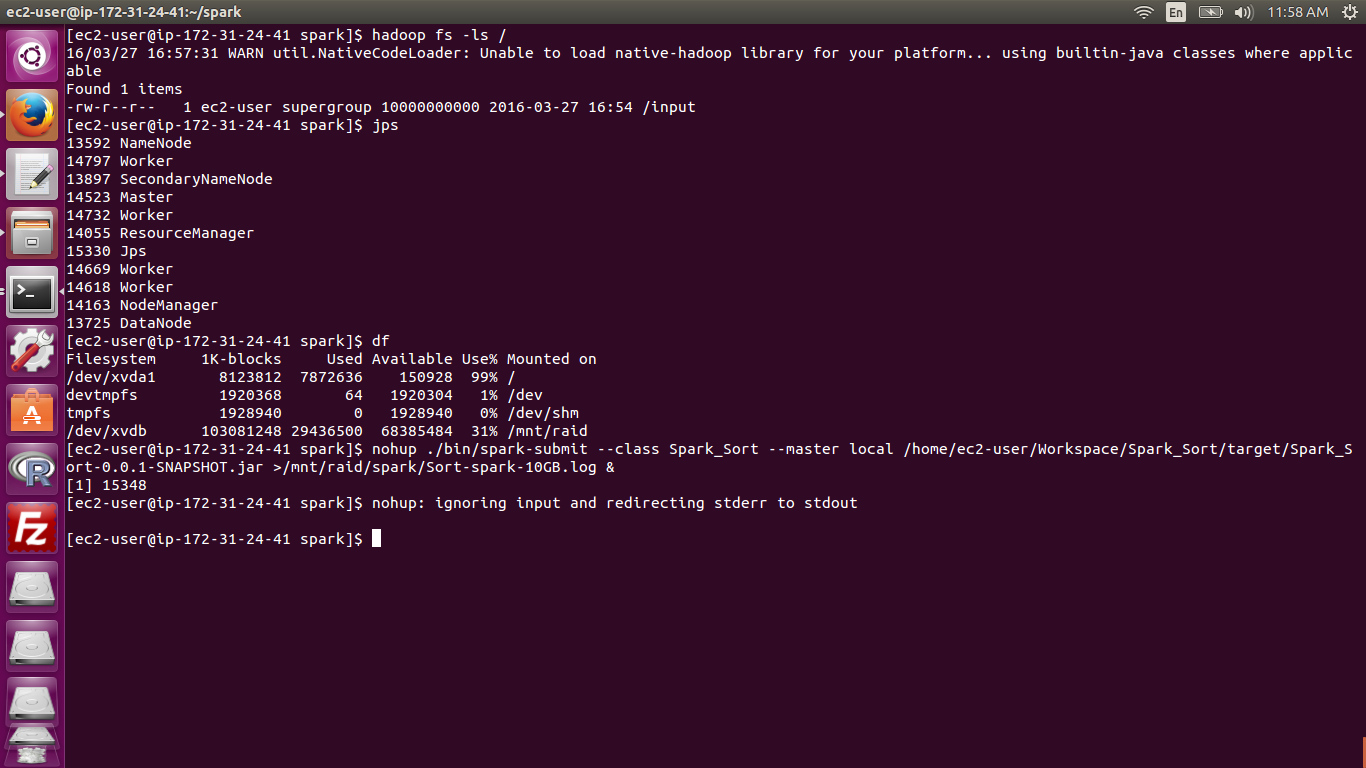
**10GB Hadoop End**



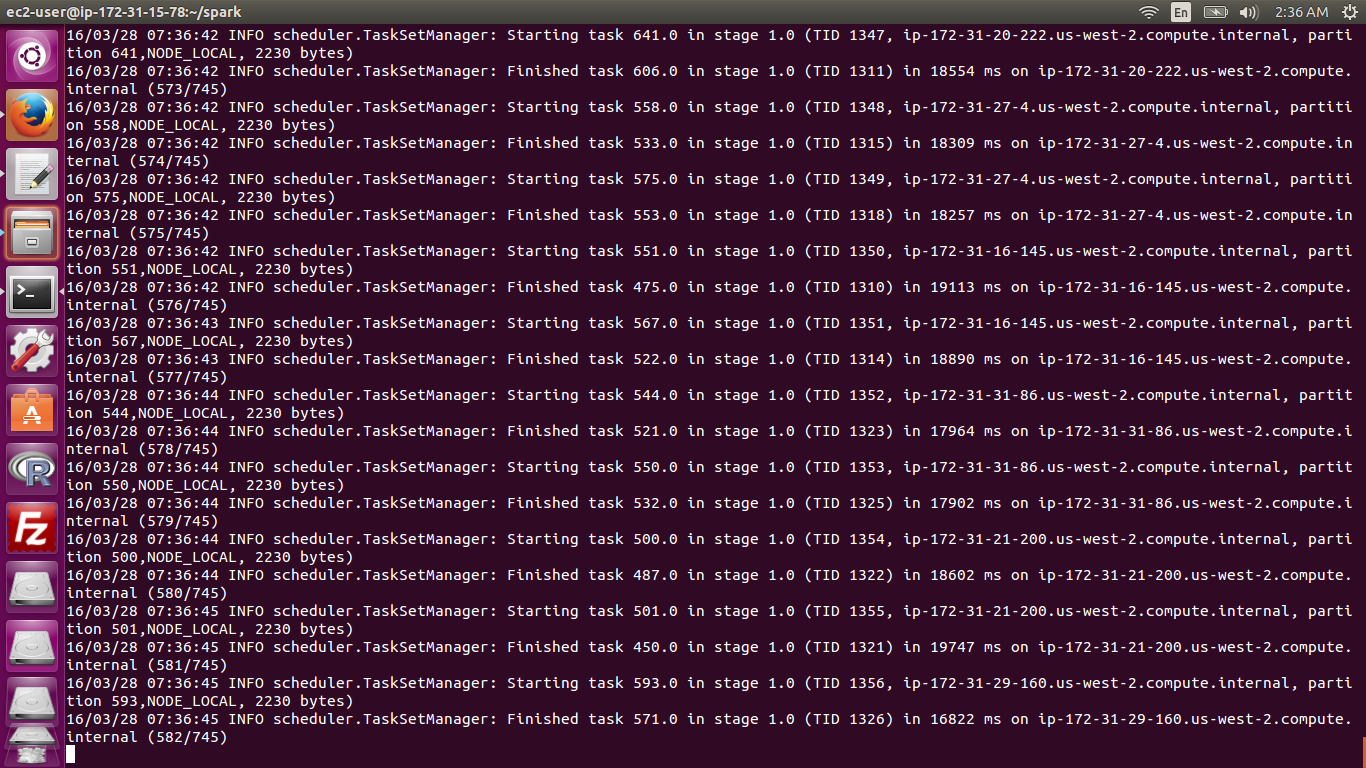
**10GB Hadoop valsort**

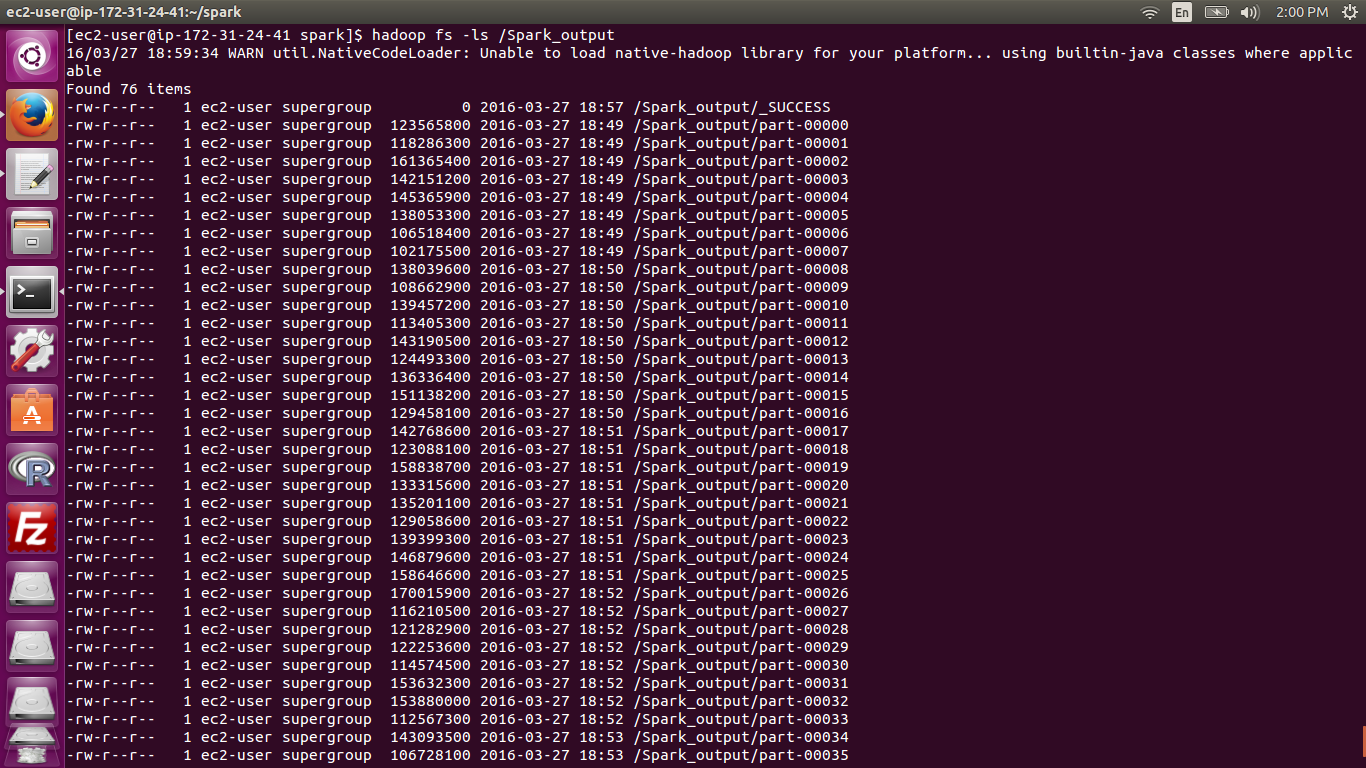


**10GB Spark Start**

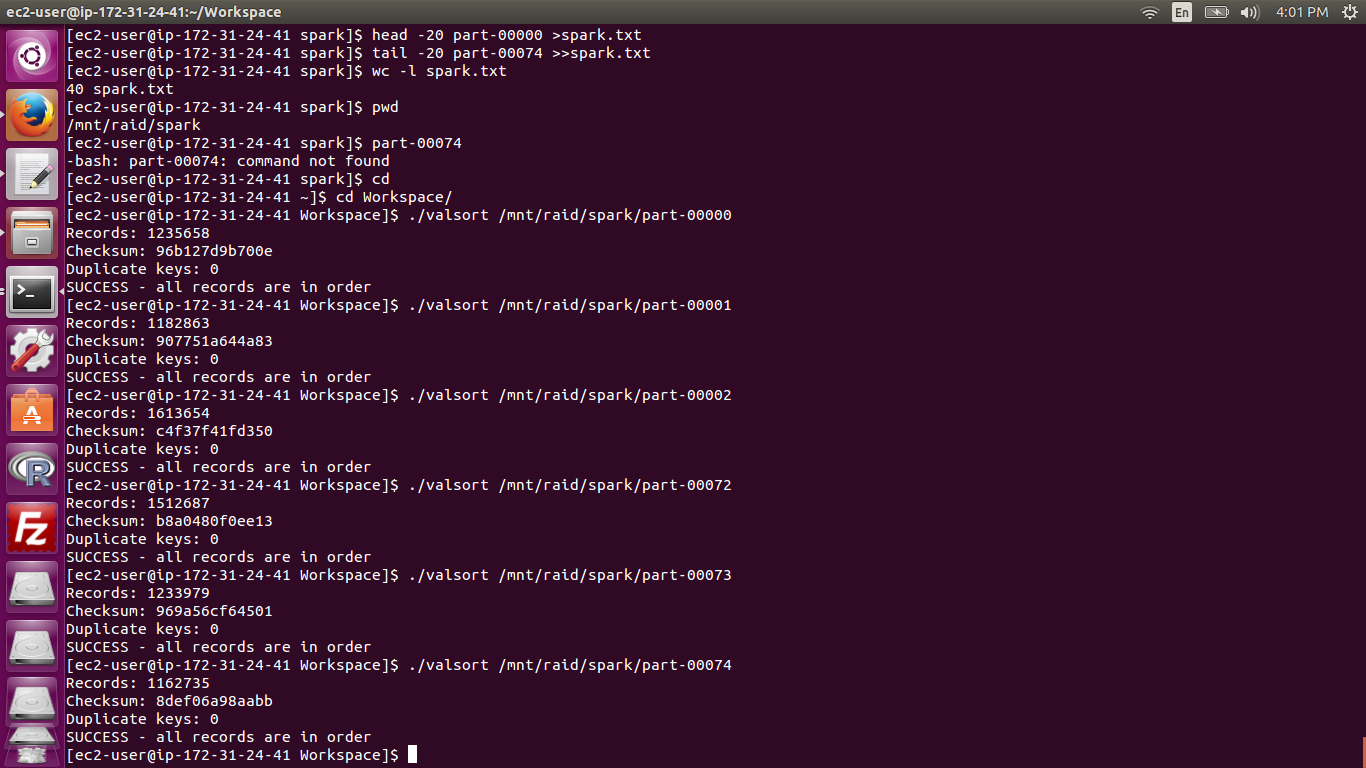


**10GB Spark End**



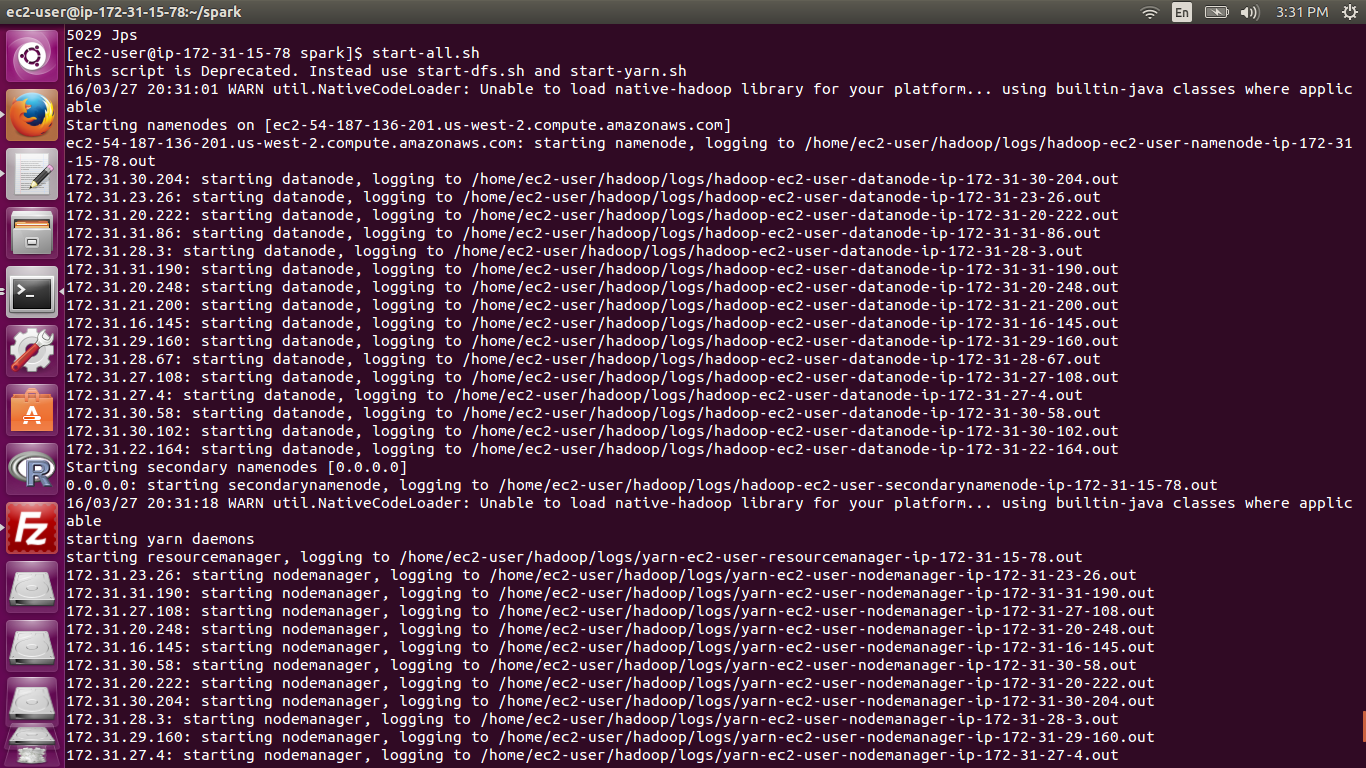


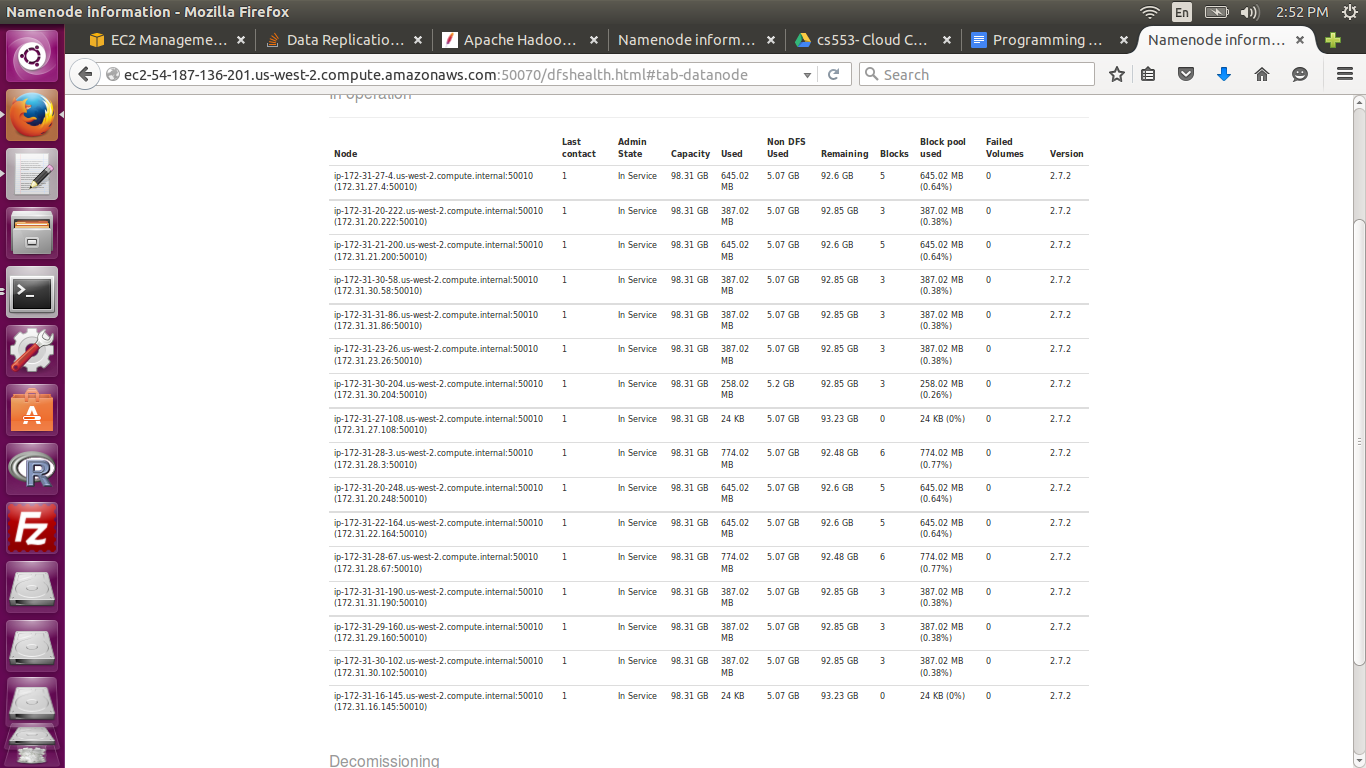
**10GB Spark valsort**



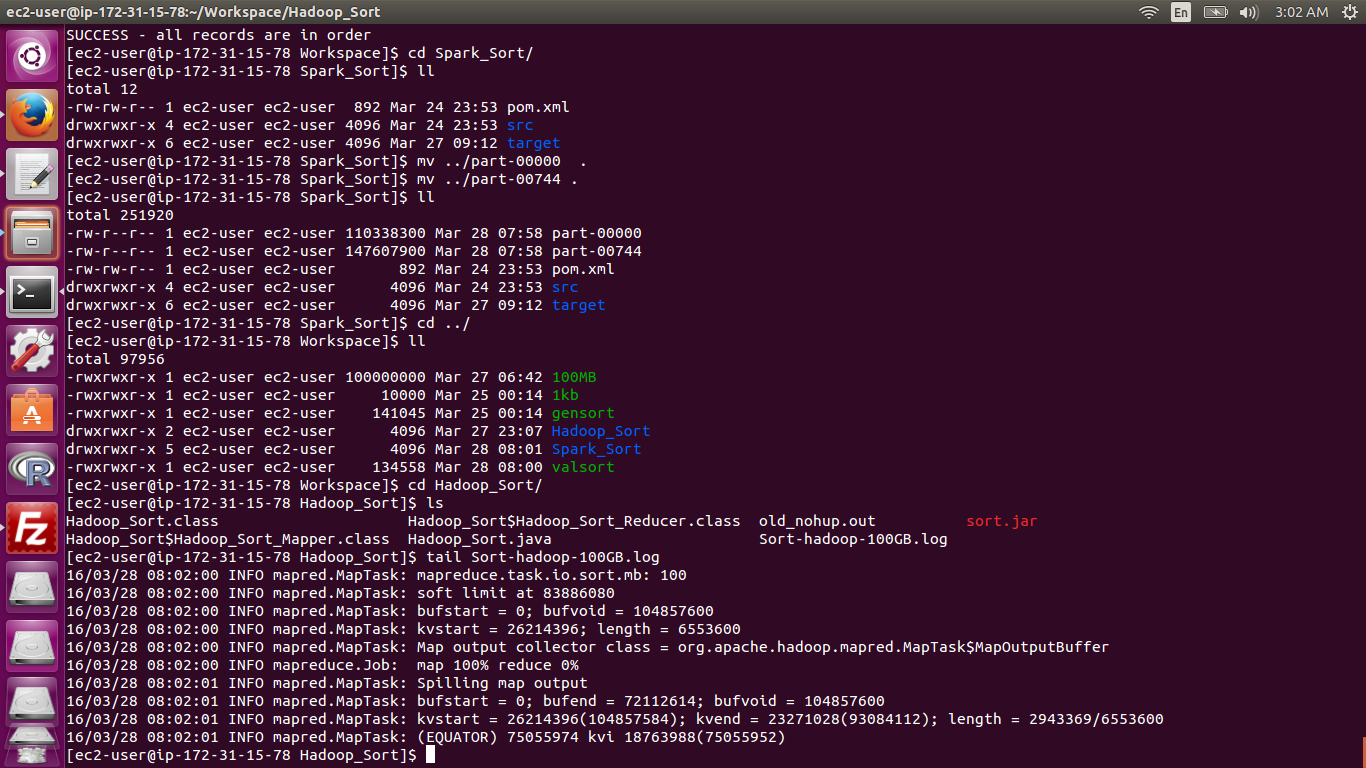
**C3.large**

**100GB Hadoop Start**

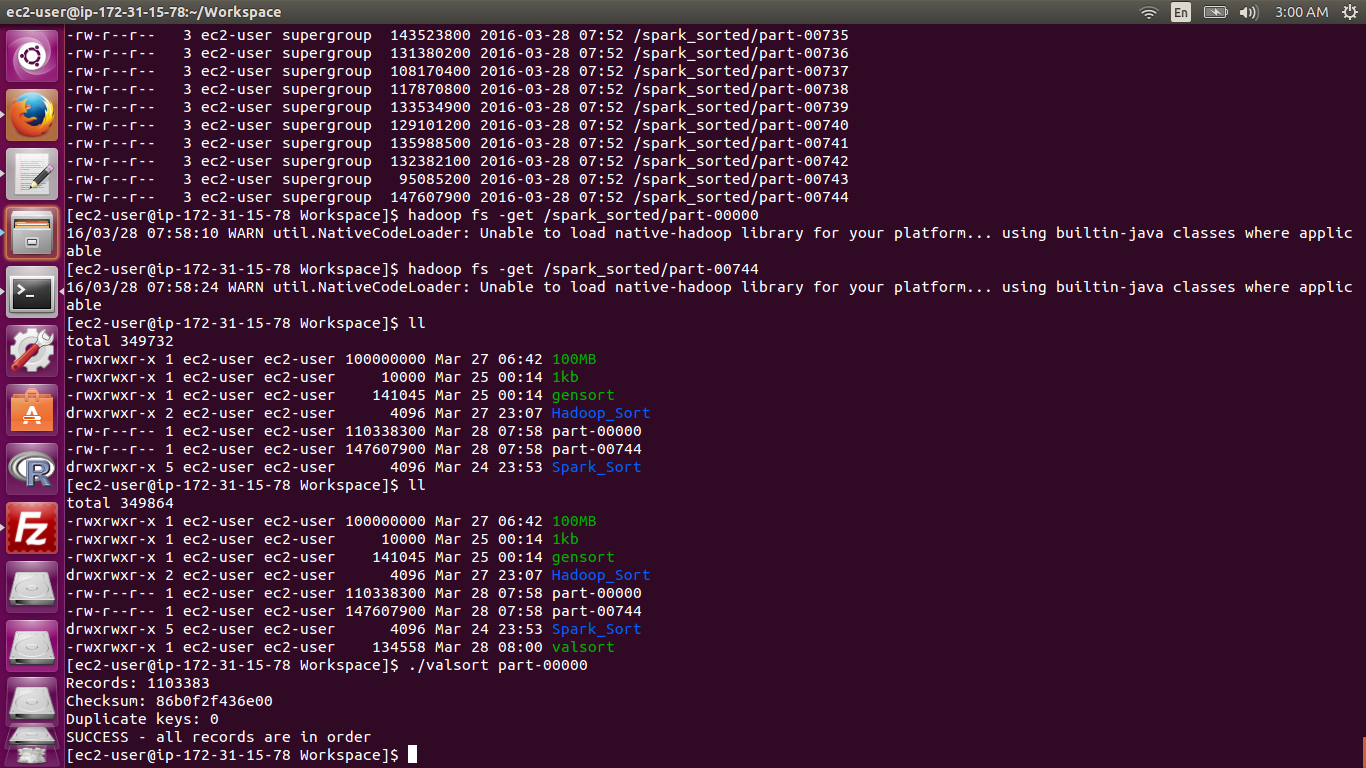




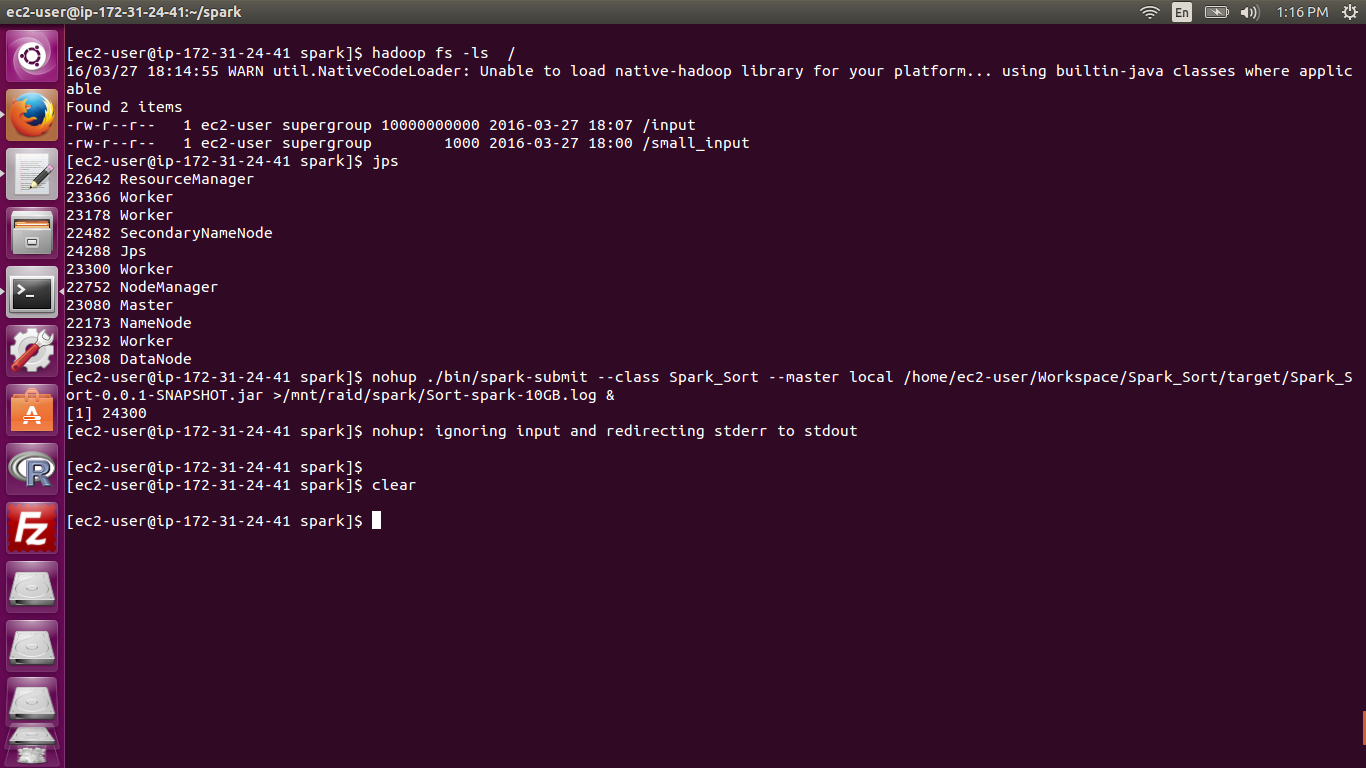
**100GB Hadoop End**

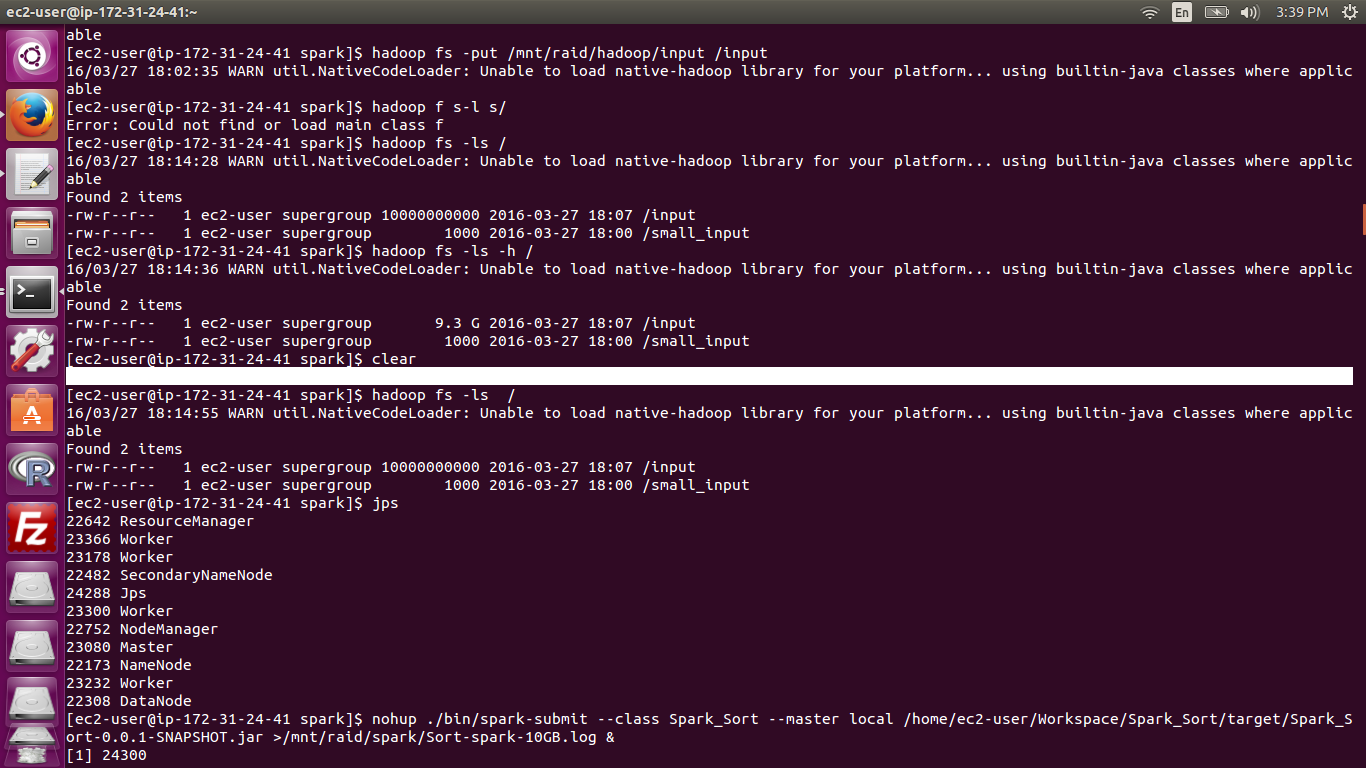


**100GB Hadoop valsort**

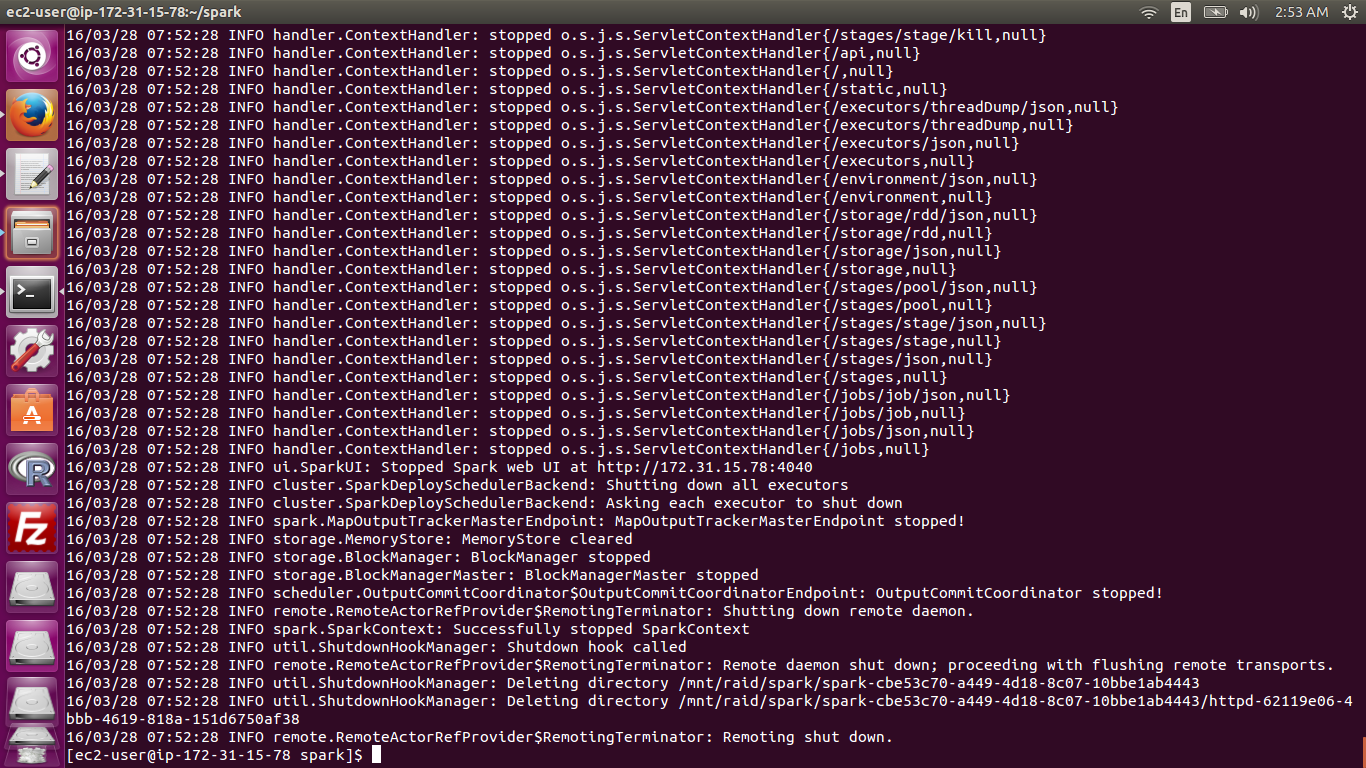


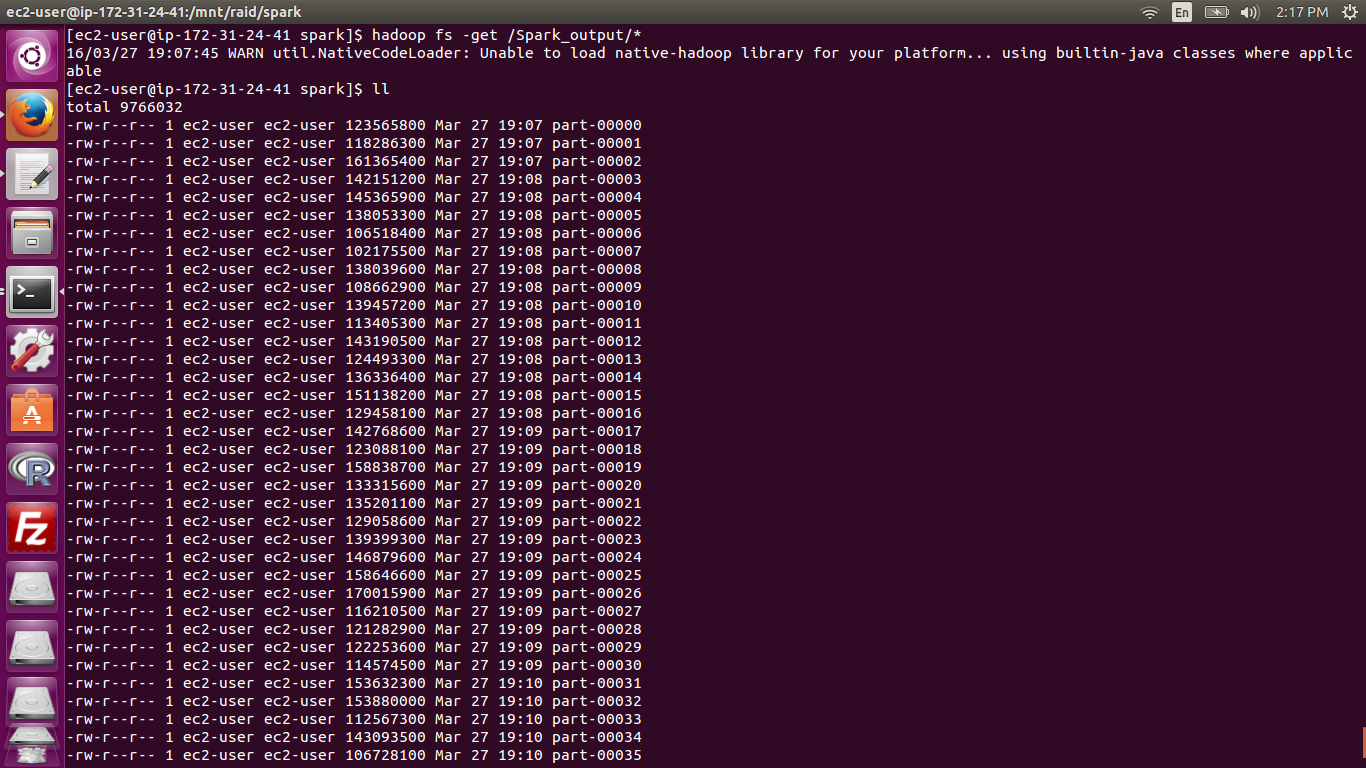
**100GB Spark Start**

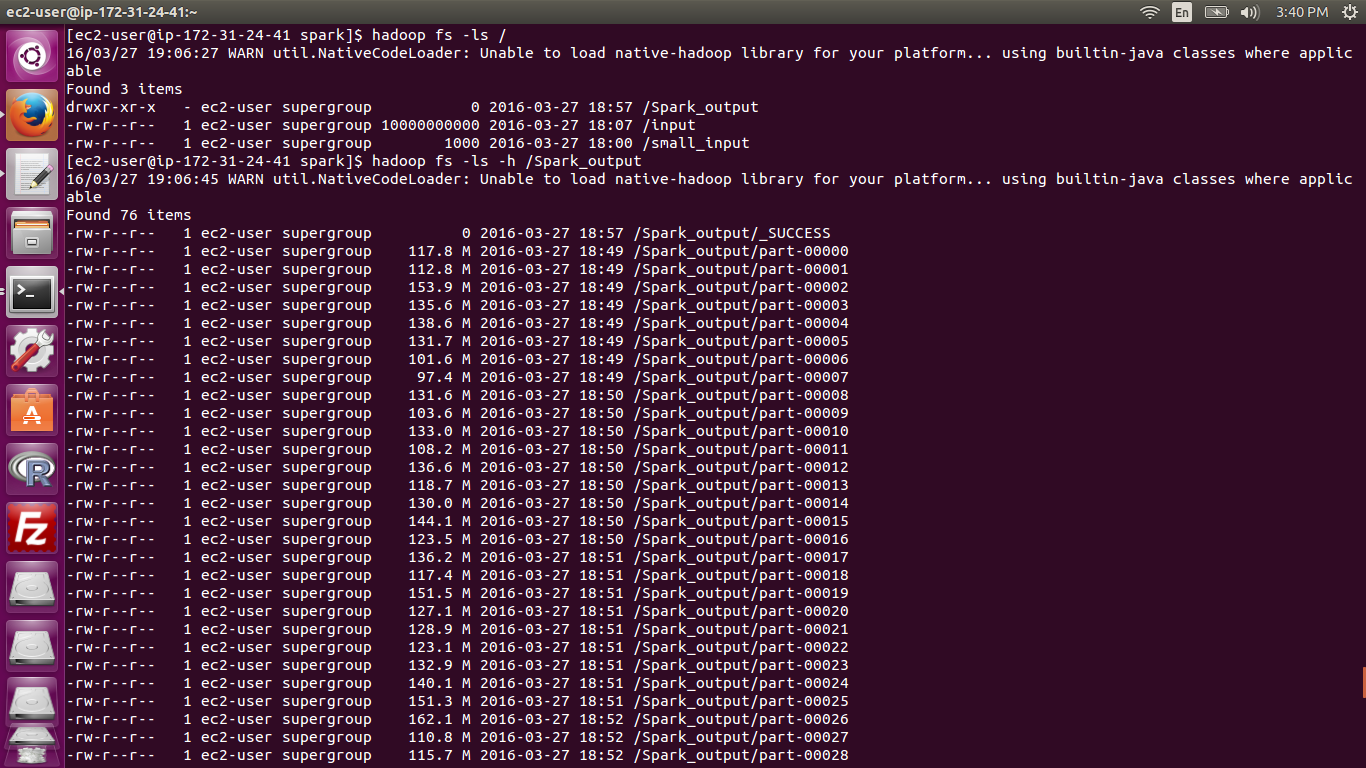




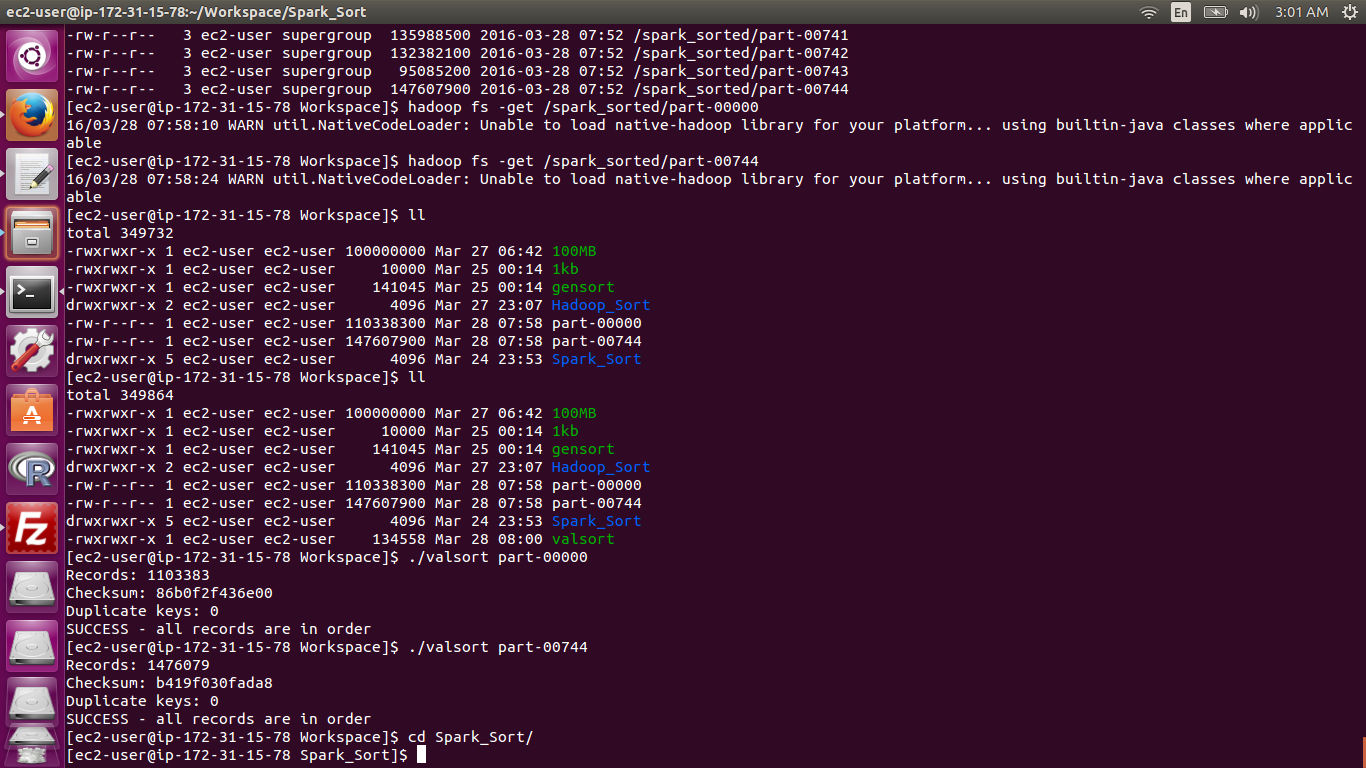
**100GB Spark End**







**100GB Spark valsort**



**Cluster IPs**

