

Cloud Computing Assignment 2

Note :

All the experiments have been performed using 1GB and 1TB datasets on 1 and 16 nodes.

Instance Type – d2.xlarge - Zone – US East (N Virginia)

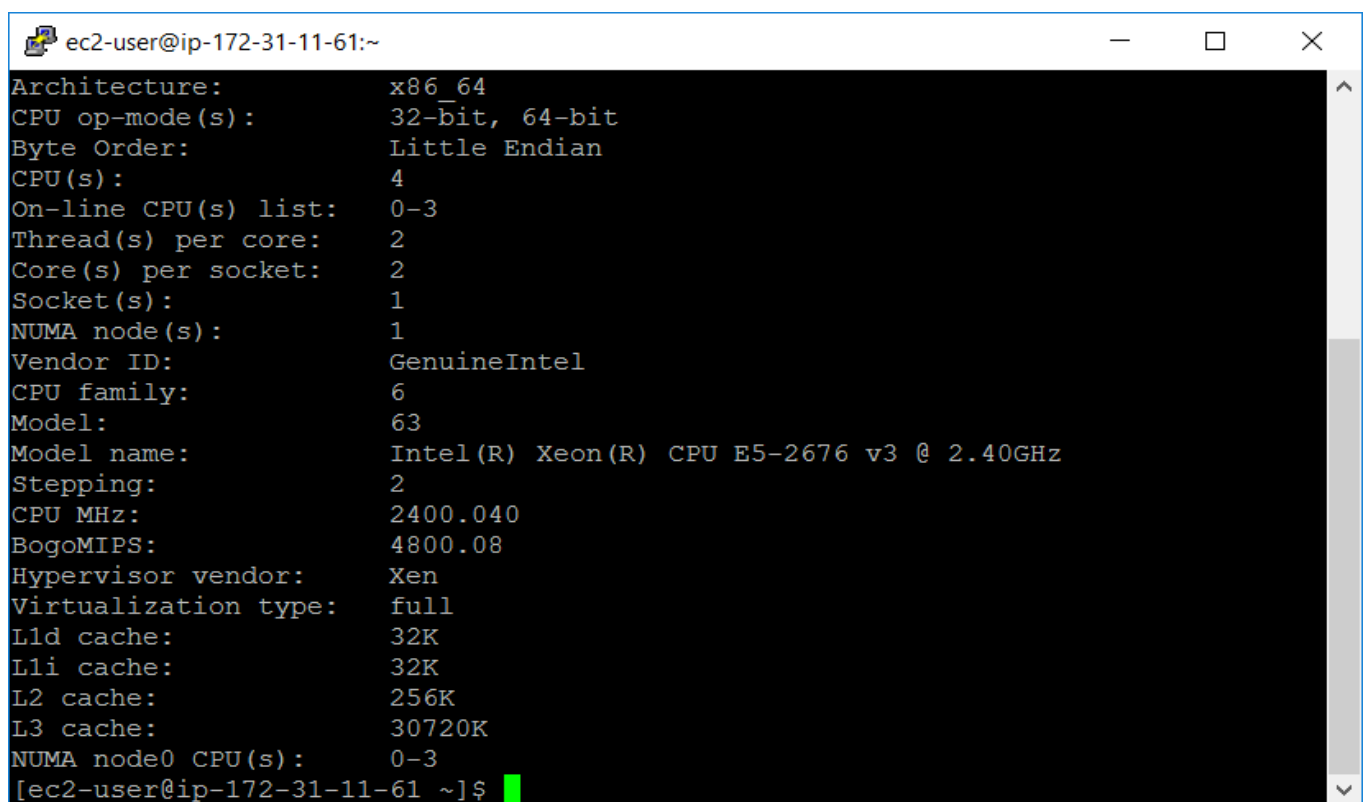
Hadoop Version – 2.7.2

Spark Version – 1.6.1

Java Version - 1.7.0_95

Operation System - Linux AMI 2016.03.0

The following image shows the system configuration for d2.xlarge instance.

A terminal window titled 'ec2-user@ip-172-31-11-61:~' displays the output of the 'lscpu' command. The output lists various system parameters including architecture (x86_64), CPU op-mode(s) (32-bit, 64-bit), byte order (Little Endian), number of CPUs (4), online CPU list (0-3), threads per core (2), cores per socket (2), socket count (1), NUMA node (1), vendor ID (GenuineIntel), CPU family (6), model (63), model name (Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz), stepping (2), CPU MHz (2400.040), bogomips (4800.08), hypervisor vendor (Xen), virtualization type (full), L1d and L1i caches (32K), L2 cache (256K), L3 cache (30720K), and NUMA node0 CPU(s) (0-3). The prompt '[ec2-user@ip-172-31-11-61 ~]\$' is followed by a green cursor.

```
ec2-user@ip-172-31-11-61:~  
Architecture:          x86_64  
CPU op-mode(s):        32-bit, 64-bit  
Byte Order:            Little Endian  
CPU(s):                4  
On-line CPU(s) list:   0-3  
Thread(s) per core:    2  
Core(s) per socket:    2  
Socket(s):             1  
NUMA node(s):          1  
Vendor ID:             GenuineIntel  
CPU family:            6  
Model:                 63  
Model name:            Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz  
Stepping:               2  
CPU MHz:               2400.040  
BogoMIPS:              4800.08  
Hypervisor vendor:     Xen  
Virtualization type:   full  
L1d cache:             32K  
L1i cache:             32K  
L2 cache:              256K  
L3 cache:              30720K  
NUMA node0 CPU(s):     0-3  
[ec2-user@ip-172-31-11-61 ~]$
```

Hadoop (2.7.2):

Hadoop Cluster Installation Steps

Before starting the steps for hadoop cluster installation , please keep .pem file and the files from **2_Hadoop\Hadoop XML & Scripts** folder at /home/ec2-user/ location

Master Node

- 1) Provide permission to .pem file using following command

```
chmod 400 Parth.pem
```

- 2) Enter the following command to authorize the use of key for agents session.

```
eval `ssh-agent -s`  
ssh-add Parth.pem
```

- 3) Use the script hadoop_install.sh which will execute the following

- RAID 0 the underlying disk
- Install Hadoop

- 4) Set the following environment variable in .bashrc file

```
export CONF=/usr/local/hadoop/etc/hadoop  
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.95.x86_64  
export HADOOP_INSTALL=/usr/local/hadoop  
export PATH=$PATH:$HADOOP_INSTALL/bin  
export PATH=$PATH:$HADOOP_INSTALL/sbin  
export HADOOP_COMMON_HOME=$HADOOP_INSTALL  
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL  
export YARN_HOME=$HADOOP_INSTALL  
export HADOOP_PREFIX=/usr/local/hadoop  
export HADOOP_HDFS_HOME=$HADOOP_INSTALL  
export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop  
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

- 5) Execute command source ~/.bashrc

- 6) Update the following xml files at path **/usr/local/hadoop/etc/hadoop** with same configuration as provided in hadoopsetup folder.

- mapred-site.xml
- core-site.xml
- hdfs-site.xml
- yarn-site.xml

- 7) Create AMI of Master Node
- 8) Create 16 slaves instances using AMI of master node
- 9) Update the slaves file of Master at location **/usr/local/hadoop/etc/hadoop** and provide the IP's of all 16 running slaves.
- 10) Copy slaves file to following path **/home/ec2-user**
- 11) Copy slaves file and **configure_slaves.sh** on slave nodes and execute **configure_slaves.sh** on slaves nodes using following commands from master node.

Copy files on slave node

```
scp -i Parth.pem -r /home/ec2-user/slaves SLAVESIP:/home/ec2-user
```

```
scp -i Parth.pem -r /home/ec2-user/configure_slaves.sh SLAVESIP:/home/ec2-user
```

Running configure_slaves

```
ssh -i Parth.pem SLAVESIP '/home/ec2-user/configure_slaves.sh'
```

- 12) On the Mater node execute the following command to format name node

```
hadoop namenode -format
```

- 13) Execute the script **createcluster.sh** on Master Node.
- 14) Use **Jps** command and you will see the following running proces on the Master Node
 - NameNode
 - Secondary NameNode
 - Resource Manager

15) Please check the cluster report using following command

Hadoop dfsadmin -report and you will see 16 live Datanodes.

```
ec2-user@ip-172-31-8-8:/usr/local/hadoop
Configured Capacity: 95259891859456 (86.64 TB)
Present Capacity: 90457722191072 (82.27 TB)
DFS Remaining: 90457721798656 (82.27 TB)
DFS Used: 393216 (384 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0

-----
Live datanodes (16):

Name: 172.31.14.59:50010 (ip-172-31-14-59.ec2.internal)
Hostname: ip-172-31-14-59.ec2.internal
Decommission Status : Normal
Configured Capacity: 5953743241216 (5.41 TB)
DFS Used: 24576 (24 KB)
Non DFS Used: 300135604224 (279.52 GB)
DFS Remaining: 5653607612416 (5.14 TB)
DFS Used%: 0.00%
DFS Remaining%: 94.96%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Fri Mar 18 04:37:31 UTC 2016

Name: 172.31.9.84:50010 (ip-172-31-9-84.ec2.internal)
Hostname: ip-172-31-9-84.ec2.internal
Decommission Status : Normal
Configured Capacity: 5953743241216 (5.41 TB)
DFS Used: 24576 (24 KB)
Non DFS Used: 300135604224 (279.52 GB)
DFS Remaining: 5653607612416 (5.14 TB)
DFS Used%: 0.00%
DFS Remaining%: 94.96%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Fri Mar 18 04:37:31 UTC 2016
```

Executing Sample Java Program

1) Create Directory in hdfs

```
hadoop fs -mkdir /user
hadoop fs -mkdir /user/ec2-user
```

2) Create Directory for input file

```
hadoop fs -mkdir input
```

3) Place input.txt file in input folder

```
hadoop fs -put input.txt input
```

4) Compile java program

```
hadoop hadoop com.sun.tools.javac.Main Sample.java
```

5) Creating jar from .class file

```
jar cf Sample.jar *.class (all the required class files)
```

6) Executing jar

```
hadoop jar Sample.jar Sample argument1 argument2
```

The following are the configuration files used for setting up hadoop 2.7.2 cluster.

- 1) core-site.xml
- 2) hdfs-site.xml
- 3) mapred-site.xml
- 4) slaves
- 5) yarn-site.xml

Following is the description of each file , I have highlighted the properties that needs to be changed during Installation.

1) Core-site.xml -

The core-site.xml informs the hadoop daemon on which machine Namenode is running in the cluster. I have used the property **hadoop.tmp.dir** to inform hadoop framework about where to store the intermediate temporary results i.e. in cluster setup I have used the mounted raid 0 disk to store the intermediate results by providing appropriate configuration in core-site.xml.

```
<property>  
  <name>hadoop.tmp.dir</name>  
  <value>/mnt/raid/temporary_dir</value>  
</property>
```

2) hdfs-site.xml -

The hdfs-site.xml contains the configuration setting for hdfs daemons for Namenode, Datanode and Secondary Namenode. We can manipulate the no of replication of blocks created for the input file by using **dfs.replication**, by defaults the replication factor is 3, for our experiment I have kept the replication factor as 1. We can also set the directories for storing data blocks in Namenode and Datanode using properties **dfs.namenode.name.dir** and **dfs.datanode.data.dir** respectively. I have used the mounted raid 0 disk for storing data blocks in Namenode and Datanode

```
<configuration>  
  <property>  
    <name>dfs.replication</name>  
    <value>1</value>  
  </property>  
  <property>  
    <name>dfs.namenode.name.dir</name>  
    <value>file:///mnt/raid/temporary_dir/namenode</value>  
  </property>  
  <property>  
    <name>dfs.datanode.data.dir</name>  
    <value>file:///mnt/raid/temporary_dir/datanode</value>  
  </property>  
</configuration>
```

3) Mapred-site.xml

The mapred-site.xml contains the configuration settings for MapReduce framework using property **mapreduce.framework.name**. We have to configure jobtracker's address as Name node using property **mapreduce.jobtracker.address**. We can also configure the amount of memory to request from the scheduler for each map task and each reduce task using properties **mapreduce.map.memory.mb** and **mapreduce.reduce.memory.mb** respectively. I have used the following configuration to request from scheduler for performing map task and reduce task.

```
<property>
  <name>mapreduce.map.memory.mb</name>
  <value>2560</value>
</property>
<property>
  <name>mapreduce.reduce.memory.mb</name>
  <value>5120</value>
</property>
```

4) Slaves

The slaves file contains IP address of each DataNode one per line. As we were working with 16 nodes, the slaves file contained 16 different IPs of Data nodes.

5) Yarn-site.xml

The property **yarn.nodemanager.aux-services** notifies the Nodemanager that there is auxiliary services called `mapreduce_shuffle` which the Nodemanager needs to implement. We can also set the amount of physical memory, in MB, that can be allocated for containers using **yarn.nodemanager.resource.memory-mb** property. We can also set address of the scheduler interface and other configurations related to resource manager.

```
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>28000</value>
</property>
```

As we move from **1 node to 16**, we will need to update the slave file, provide appropriate memory for map task and reduce task in mapred-site.xml using **mapreduce.map.memory.mb** and **mapreduce.reduce.memory.mb** properties respectively and provide sufficient memory to containers by properly configuring **yarn.nodemanager.resource.memory-mb** property in yarn-site.xml.

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

Ans :

Master node (Name Node) : Master node in hadoop ecosystem also know as NameNode manages entired hdfs storage across Data Nodes . Master node contains the Metadata about mapping of data blocks with the DataNode . The Master Node is single point of failure in hadoop cluster. The Master node also contains Resource Manager which communicates with Node Manager of DataNodes to assign tasks and to get notified when task is completed.

Slave (Data Node) : Slave node in hadoop ecosystem also know as DataNode does all the work of storing data and running computations. Slave nodes consist of Node manager that communicates and receives instructions from the master node. Slaves nodes performs exctly as instructed by Mater 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?

Ans: The communication in cluster across nodes happens through unique ports , if we set similar ports we will not able to successfully start hadoop cluster. There will be communication failure in case of similar ports. The error which can appear in case of similar port will be or Port Binding error while starting hadoop cluster.

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

Ans: We can change the number of mappers and reducer by using mapred-site.xml file.We will need to change the property name mapreduce.job.maps and mapreduce.job.reduces for mappers and reducers respectively.

```
<name> mapreduce.job.maps</name>
<value>2</value>
<name> mapreduce.job.reduces</name>
<value>4</value>
```

Spark – 1.6.1

Steps for Spark Cluster Installation

- 1) First we will need to export AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY

For example

```
export AWS_ACCESS_KEY_ID=AKIAI6EPPDHR7TYS77SA  
export AWS_SECRET_ACCESS_KEY=iGahLeV8HZXkYo6Z6Y0uPrmMRCFre0aZFikwC
```

- 2) Provide permission to .pem file using following command

```
chmod 400 Parth.pem
```

- 3) Enter the following command to authorize the use of key for agents session.

```
eval `ssh-agent -s`  
ssh-add Parth.pem
```

- 4) Download Spark and navigate to Spark → EC2 folder and execute the following command

```
./spark-ec2 -k Parth -i /home/ec2-user/Parth.pem -s16 -t d2.xlarge -r us-east-1 --  
spot-price=0.70 launch Spark
```

- 5) After executing the previous command Spark cluster will be set up using 1 Master and 16 Slaves.

- 6) Login to the master node using following command

```
./spark-ec2 --key-pair=Parth --identity-file=/home/ec2-user/Parth.pem login Spark
```

- 7) Navigate to /root/spark/sbin and execute ./stop-all.sh

- 8) Navigate to /root/ephemeral-hdfs/bin and execute ./stop-all.sh

- 9) Make changes only in following xml files as provided in 3_Spark\Spark XML folder.

- 10) At path /root/ephemeral-hdfs/conf make changes to hdfs-site.xml and core-site.xml as provided in **3_Spark\Spark XML\hdfs-conf** folder.

11) At path `/root/spark/conf` make changes to `spark-env.sh` and `core-site.xml` as provided in **3_Spark\Spark XML\spark conf** folder.

12) Navigate to `/root/spark-ec2` and run following 2 commands to copy the changes made to all slave files

```
copy-dir /root/ephemeral-hdfs/conf
```

```
copy-dir /root/spark/conf
```

13) Login in to slave nodes and execute the following command to raid0 the underlying disks

```
sudo mdadm --create --verbose /dev/md0 --level=0 --name=createraid --raid-devices=2 /dev/xvdb /dev/xvdc
```

```
sudo mkfs.ext4 -L createraid /dev/md0
```

```
sudo mkdir -p /mnt/raid
```

```
sudo mount LABEL=createraid /mnt/raid
```

```
sudo mkdir /mnt/raid/temporary_dir
```

14) Navigate to `/root/ephemeral-hdfs/bin` and execute following command

```
./hadoop namenode -format
```

15) Navigate to `/root/ephemeral-hdfs/bin` and execute `./start-all.sh`

16) Navigate to `/root/spark/sbin` and execute `./start-all.sh`

17) Spark cluster will be started now

Executing Sample Scala Program

- 1) Placing input file in hdfs

Navigate to root and execute following command

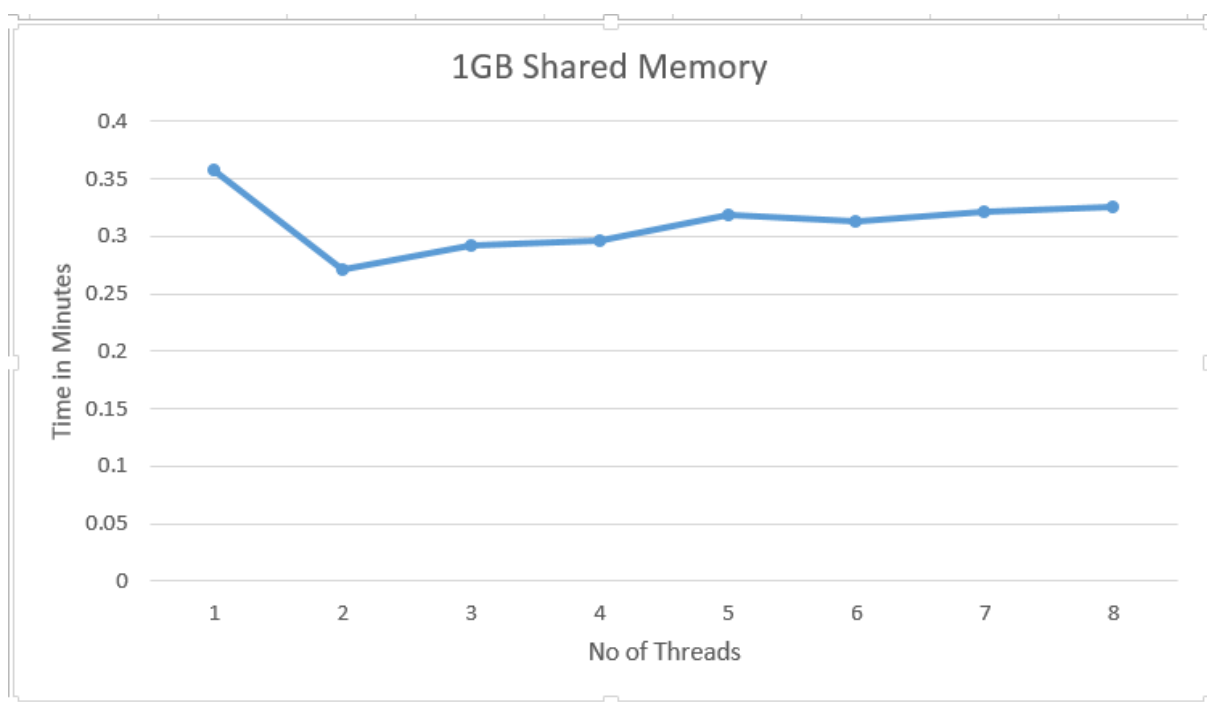
ephemeral-hdfs/bin/hadoop fs -put input.txt input.txt

- 2) Navigate to /root/spark/bin and execute command . /spark-shell to run spark-shell
- 3) In console type :paste and copy your source code and press enter and ctrl + d
- 4) The program execution will be started.

Shared Memory Experiments

- The Shared Memory Tera Sort Experiments were performed on single node using 1GB and 1TB data sets.
- The following table give the time taken for performing experiments on 1GB data sets using 1-8 threads.

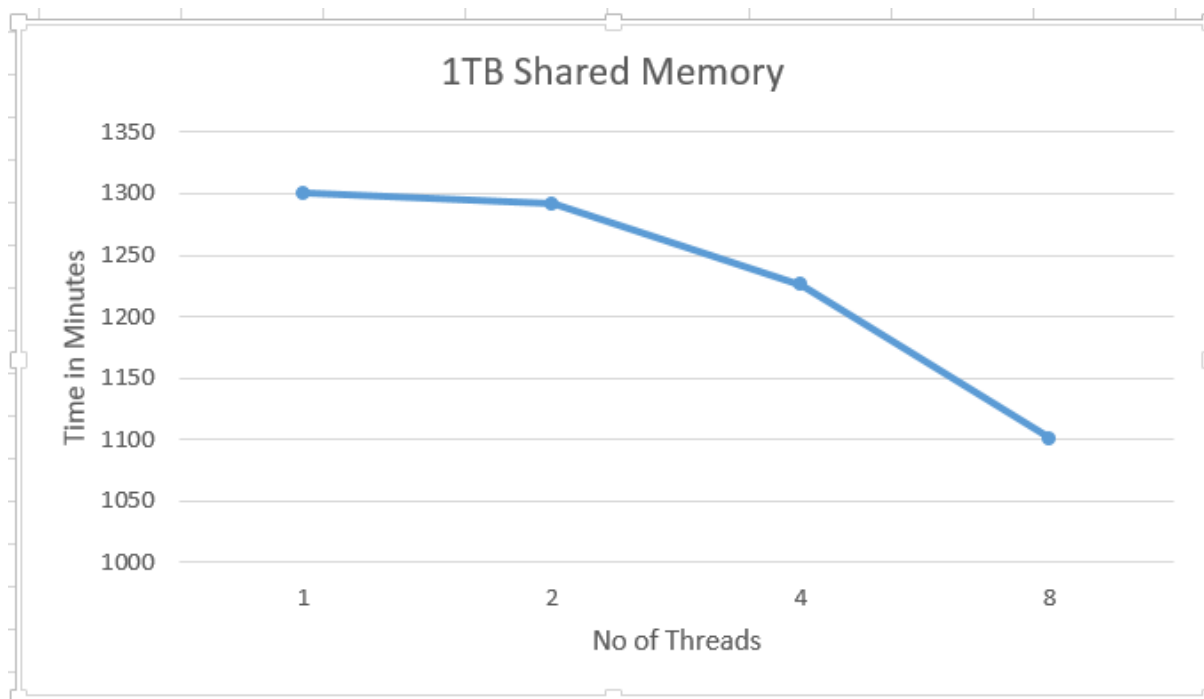
	Time taken (Minutes)							
	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
1GB	0.357	0.271	0.292	0.296	0.318	0.313	0.321	0.325



- From the above graph we can observe that for sorting 1GB of data it works best with 3 and 4 threads which we can assume that as there are 4 underlying virtual cores the best performance can be achieved using 4 threads.
- However in the above graph we can see threads 2 also takes relatively less time .
- What we can conclude is as the data set is relatively less in size we are not able to obtain absolutely perfect results.

- The following table give the time taken for performing experiments on 1TB data sets using 1-2-4 and 8 threads.

	Time taken (Minutes)			
	1 Thread	2 Threads	4 Threads	8 Threads
1TB	1300.48	1291.81	1225.57	1100.81



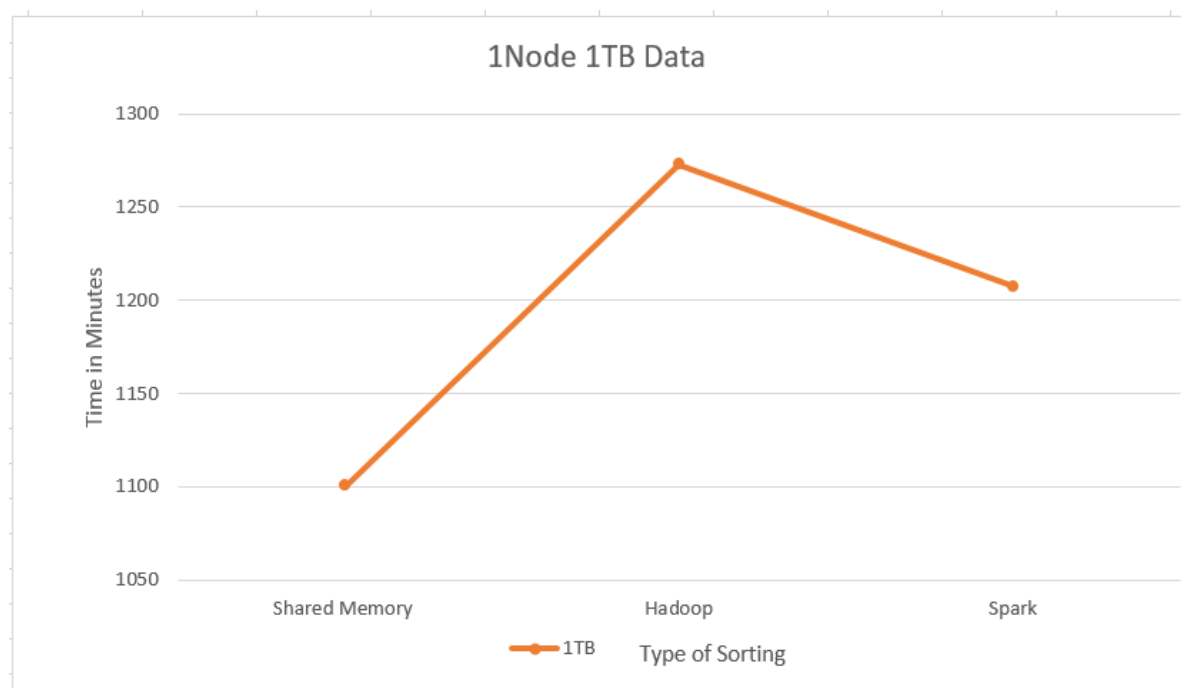
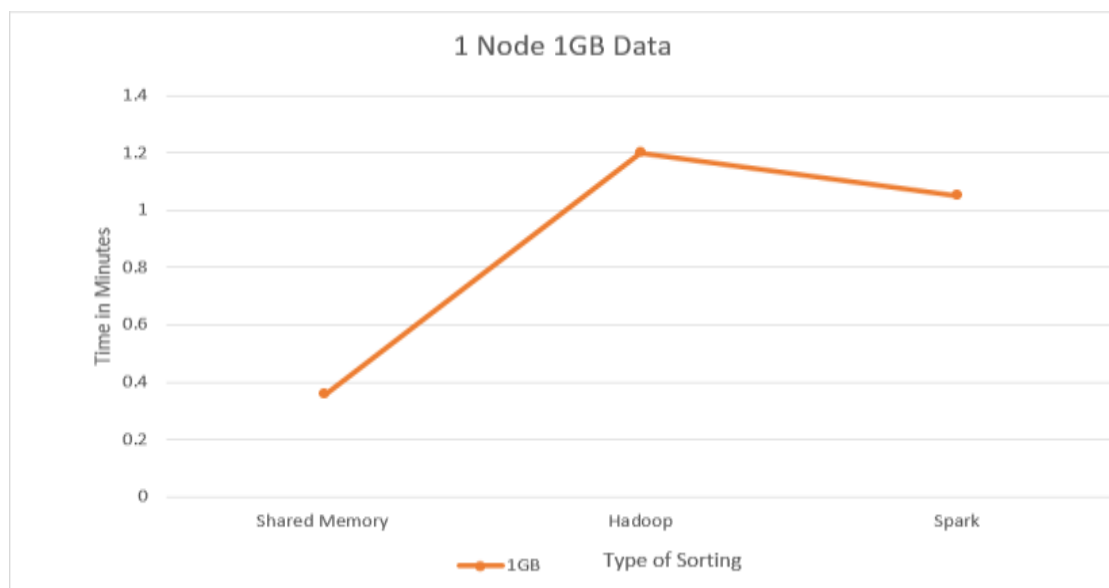
- As we can see from the above graph best performance was achieved with 8 threads , while running experiment for 1TB with 8 threads the intermediate chunks created from the large input file was of size 128 MB.
- The experiment with 4 threads was performed with intermediate chunks size of 64 MB which also conceeds relatively less time than working with single thread.
- What important result I was able to extract was working with 128MB chunks gave the most optimal result.
- What I can conclude from the overall sharedmemory experiment is working with 4 threads (as there are 4 virtual cores available in d2.xlarge instance) and 128MB Chunks for splitting the large files gives the most optimal results.**

Performance Evaluation

Execution Time Charts & Throughput Chart for 1 node Experiments

- 1) Comparision of sorting 1GB and 1TB data on a **single** node. The following table shows the time required by different sorting techniques in minutes to sort 1GB and 1TB of data.

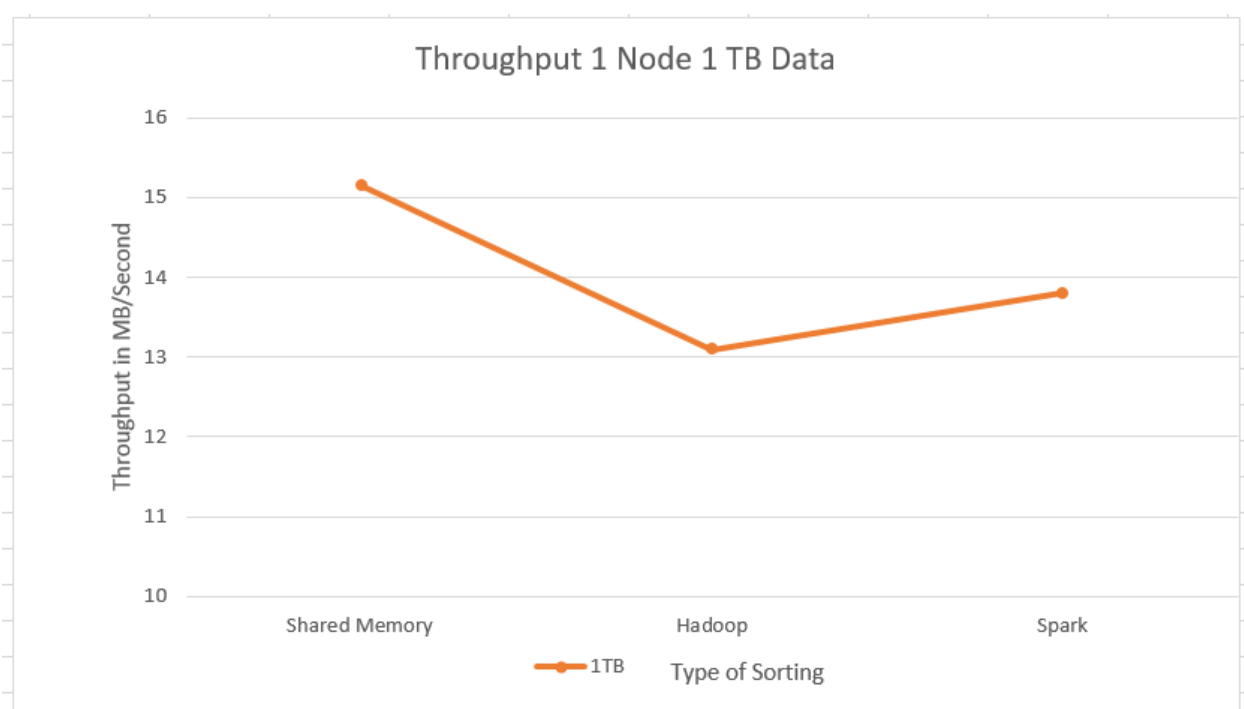
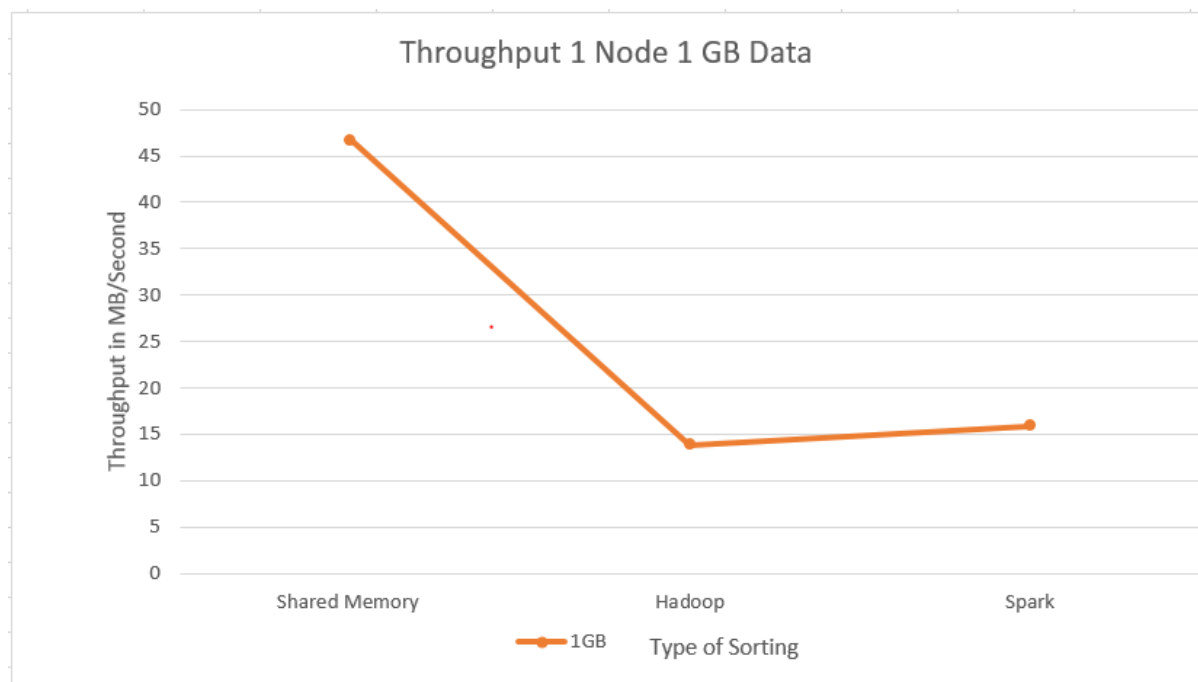
	Shared Memory Execution Time (Minutes)	Hadoop Execution Time (Minutes)	Spark Execution Time (Minutes)
1GB	0.357	1.200	1.05
1TB	1100.489	1272.900	1207.54



Execution Time Charts for 1 node

- 2) Comparison of throughput obtained by sorting 1GB and 1TB dataset on a **single** node. The following table shows the **throughput** obtained in MB/second for sorting 1GB and 1TB of data on a single node.

	Shared Memory Throughput (MB/second)	Hadoop Throughput (MB/second)	Spark Throughput (MB/second)
1GB	46.68	13.88	15.87
1TB	15.144	13.09	13.802



Throughput Charts for Single Node

Evaluation for 1 node Experiments :

As we can see from the above graphs while working on single node using 1GB and 1TB datasets , Shared Memory Sort works faster than Hadoop and Spark. The reason for this is while working on single node there are loads of framework overhead while working with hadoop and spark. The framework needs to set hundreds of parameters for its internal calculation for generating the desired result .Working with Shared Memory Tera Sort will be fairly simple considering only a single node as there will no additional overheads and the program can function smoothly.

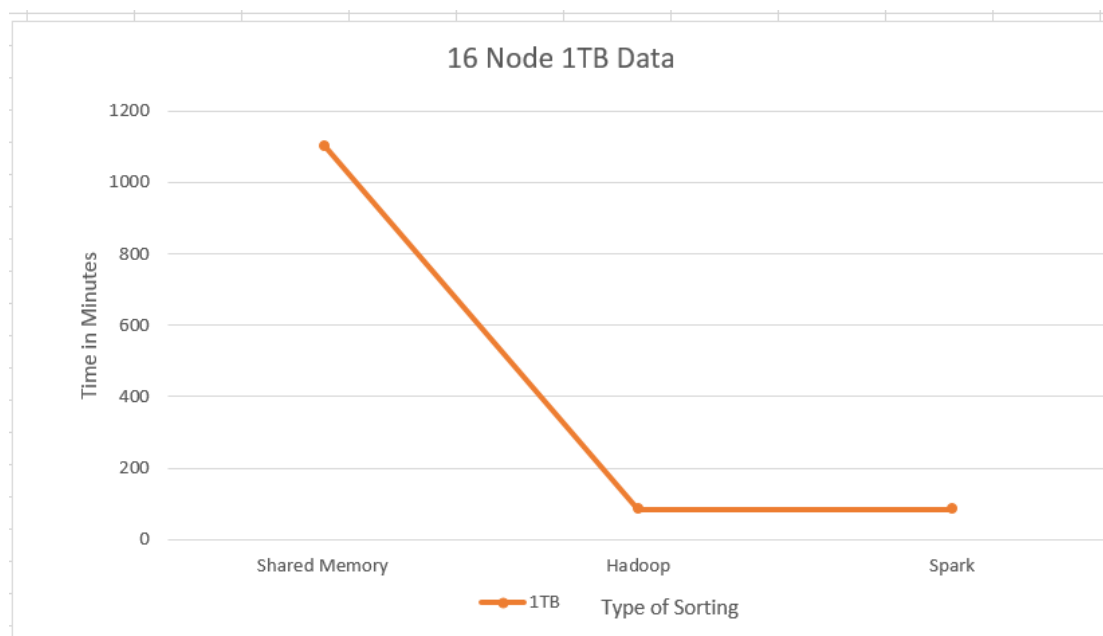
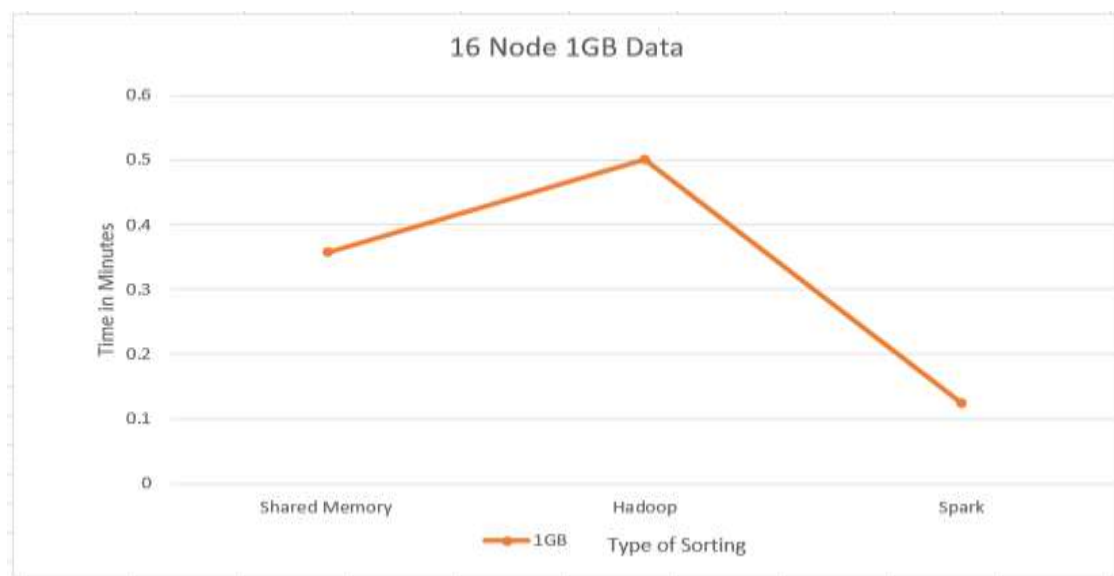
Spark works better than hadoop as spark processes data in memory. Spark jobs work best when data fits in memory. Spark also ensures lower latency computations by caching the partial results in memory across different workers.

As the time required by Shared Memory Sort is relatively low it will obtain high amount of throughput as compared to hadoop and spark while working with 1GB of dataset. However while working with 1TB of dataset there will only be little improvement working with Shared Memory Sort.

Execution Time Charts & Throughput Chart for 16 node Experiments

- 1) Comparison of sorting 1GB and 1TB dataset over a **cluster of 16 nodes**. The following table shows the time required by different sorting techniques in minutes to sort 1GB and 1TB of data over a virtual cluster of 16 nodes.

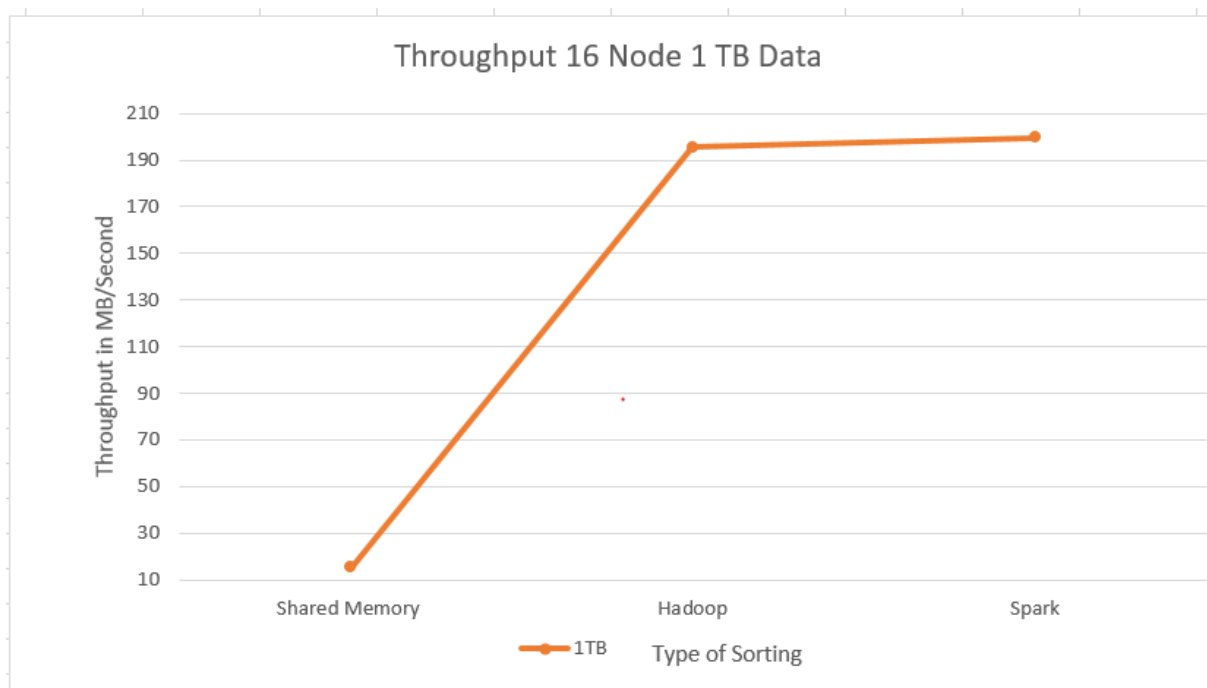
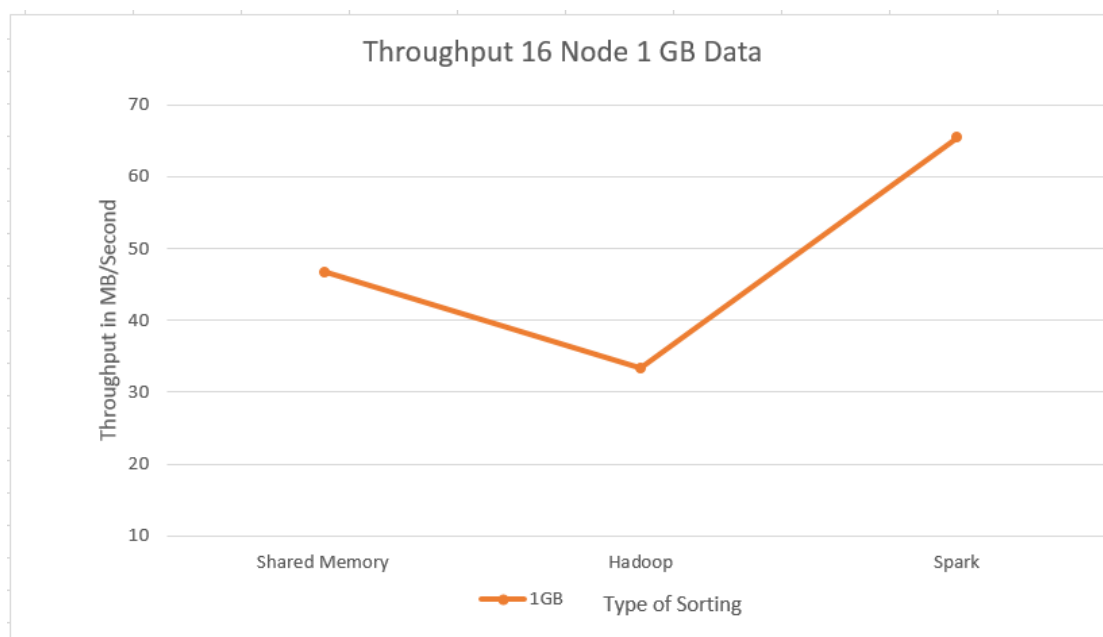
	Shared Memory Execution Time (Minutes)	Hadoop Execution Time (Minutes)	Spark Execution Time (Minutes)
1GB	0.357	0.5	0.124
1TB	1100.489	85.21	83.50



Execution Time Charts for 16 nodes

2) Comparison of throughput obtained by sorting 1GB and 1TB dataset over a cluster of 16 nodes. The following table shows the **throughput** obtained in MB/second for sorting 1GB and 1TB of data on a single node.

	Shared Memory Throughput (MB/second)	Hadoop Throughput (MB/second)	Spark Throughput (MB/second)
1GB	46.68	33.33	65.40
1TB	15.144	195.59	199.6



Throughput Charts for 16 Nodes

Evaluation for 16 nodes Experiments :

As we see from the above graphs while working with 16 nodes spark works faster as compared to Shared Memory Sort and Hadoop. As shared memory is running on single node it expected it to run quite slower when we are running our 1TB experiments.

Following are the reasons why spark runs much faster as compared to hadoop

- Spark reduces the number of read/write to disc as compared to hadoop, While working with hadoop MapReduce pushes the data back to disk after processing it.
- Spark Stores intermediate data in memory .
- Spark uses the concept of resilient distributed dataset which allows it to transparently store in memory and reducing disk read/write.

From the graphs of throughput we can observe that spark obtains throughput of nearly 200 MB/s where as hadoop obtains throughput of around 195 MB/s and shared memory obtains throughput of around 15MB/s. While running Spark and hadoop with large datasets we can neglect the time consumed due to framework overheads, where as while working with smaller datasets hadoop and Spark do not achieve optimal results due to framework overheads.

What conclusions can you draw? 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? Compare your results with those from the Sort Benchmark

From the above graphs and observation we can conclude that for **1 node** scale Shared Memory Experiments works much better than hadoop and Spark .

For working with 16 nodes Spark works better than hadoop as most of the computations are performed in memory , even data transfer from one node to another nodes happens through main memory which makes spark to run faster.

For working with 100 nodes and 1000 nodes Spark works best , which is based on functional programming language scala which is better suited for distributed environment. The inmemory batch processing and reduction in disk read/write makes spark about 10 to 100 times faster than Map Reduce .

Winners 2014	Winner 2013
Apache Spark	Hadoop
100 TB in 1,406 seconds 207 Amazon EC2 i2.8xlarge nodes x (32 vCores - 2.5Ghz Intel Xeon E5-2670 v2, 244GB memory, 8x800 GB SSD)	102.5 TB in 4,328 seconds 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks)

Data Obtained from (sortbenchmark.org)

- As we see from the above table apache spark outperforms hadoop when performed experiment for sorting around 100 TB.
- The number of nodes and total memory used across all the nodes on spark is much less than that used for hadoop.
- The inmemory computations and less disk read write gives competitive edge to spark over hadoop.
- When compared to the experiments which I have performed it gives me competitive edge on spark over hadoop on performing experiment on 1TB with 16 nodes.
- As we can see the cluster configuration used for 100 TB sorting by the winners is way more than the cluster configuration used for my experiments , scaling my experiments to that configurations might obtain good results as compared to experiments on 16 nodes.

what can you learn from the CloudSort benchmark ?

- The benchmark suggests to perform the sorting for 100 TB of data using minimum cost on any cloud platform .

- The following are the reasons for using cloud platform for sorting
Accessibility : As we all are familiar public clouds like Amazon are accessible to every one with no up front cost. Setting up a private cloud is way more expensive.

Affordability : Running a 100 TB sort is very cheap and manageable on Amazon EC2 which may cost around 650\$.

Auditability : Auditability becomes extremely easy while working on public clouds , the auditors can re run the experiments and verify the results.

- The experiments need to be performed using on-demand instances so that there is consistency across experiments.

Conclusion : The sorting should be performed on public cloud for sorting fixed amount of data with relatively low cost , working on a public cloud will help new innovation in IO intensive tasks as most current public clouds offer poor IO intensive workloads. The most effective solution will be measured using parameter of total cost of ownership.

Challenges Faced while working on Experiments

- Understand the Hadoop and Spark architecture in details to obtain the optimal result was one of the challenges I faced while working on the assignment.
- Finding out the optimal number of reducers for sorting 1TB of data which later I found out to be 32 using the formula obtained from <https://hadoop.apache.org>.
(The right number of reducers seems to be 0.95 or 1.75 multiplied by ($\text{no. of nodes} \times \text{no. of maximum containers per node}$))
- Setting up 16 nodes for Hadoop and Spark with best optimal configurations was very challenging which I managed spending several days and trying out different properties that worked best for the experiments.
- Working with Shared memory experiments and obtaining what can be the optimal size of the small chunks created from the huge file was challenging which I later found out that creating 128MB chunks gave me the best results.
- Understand how to work with HDFS system was also challenging

References

- 1) <https://portal.futuresystems.org/manual/hadoop-wordcount>
- 2) <http://blog.ditullio.fr/2016/01/04/hadoop-basics-total-order-sorting-mapreduce/>
- 3) <http://codingjunkie.net/secondary-sort/>
- 4) http://www.tutorialspoint.com/apache_spark/apache_spark_quick_guide.htm
- 5) <http://doctuts.readthedocs.org/en/latest/hadoop.html>
- 6) <http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/>
- 7) <https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/mapred/lib/TotalOrderPartitioner.html>
- 8) http://www.tutorialspoint.com/map_reduce/map_reduce_partitioner.htm
- 9) op.apache.org/docs/r2.6.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Reducer
- 10) <http://www.ashishsharma.me/2011/08/external-merge-sort.html>
- 11) <http://spark.apache.org/docs/latest/>
- 12) http://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm
- 13) <http://www.codeodor.com/index.cfm/2007/5/14/Re-Sorting-really-BIG-files---the-Java-source-code/1208>
- 14) <http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/>