Assignment 2 Report

SORT ON HADOOP/SPARK

INTRODUCTION:

The report covers the performance evaluation of the Terasort application on Hadoop, Spark and Shared memory in the Amazon Web Services(EC2 instances). Terasort is an application for sorting terabytes of data. Terabytes of data is generated in a input file, each line containing 100 bytes of data. The first 10 bytes represents the key, followed by 90 bytes which represents the value. The sorting is done on the key and the sorted files is stored in the output file. Experiments for calculating the Throughput of the system is performed for 10GB on single node and 100GB on 16 node clusters.

PRE-REQUISITE:

- Amazon Web Services(EC2 instances - C3.large instance) - Linux Operating System

C3.large INSTANCE SPECIFICATION:

- Virtual CPU - 2, Memory - 3.75 GB, SSD Storage - 2*16 GB

CONCEPTS USED IN THE PROGRAM:

- Map-Reduce
- Multi-threading

HADOOP:

Hadoop is a open source framework which is written in java by apache software foundation. The framework is used to write software application which requires to process vast amounts of data (terabytes/petabytes of data). It works in parallel on large clusters and process data in a reliable and fault-tolerant manner. It uses a concept called Map-Reduce. The backbone of Hadoop is the Hadoop Distributed File System (HDFS), where the data to be processed in Hadoop are stored. For any Hadoop job to be executed, the data should be stored in the HDFS. The data in Hadoop is processed in batches, Therefore it supports native batch processing.

- **Map-Reduce** is an algorithm / concept to process Huge amount of data in a faster way. The algorithm splits the input data-set into independent chunks.
- **Map** task will process these chunks in a completely parallel manner. Based on the functionality, key-value pairs will be generated and the output of the map phase will be given to the reduce phase. (The sorting functionality of the independent chunks will be done at this phase and the output key-value pair will be sorted)
- The primary phases of **reduce** phase are shuffle, sort and reduce.
  - **Shuffle** - Input to the reducer will be the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via the HTTP.
  - **Sort** - The framework groups Reducer inputs by the keys in this stage. The shuffle and sort phases occur simultaneously; while map-outputs are being fetched and they are arranged.
  - **Reduce** -The output of the reduce task is written to the HDFS. Applications can use the context to report progress, set application-levels status messages and update counters, or indicate if they are alive.

COMPONENTS OF HADOOP (Version 1.2.1)

- **HDFS:** It is referred as Hadoop Distributed File System, where Big-Data (Terabytes/Petabytes of data) is stored using commodity hardware. It is designed to work with large data-sets, where the default block size is 64 MB (which can be changed as per requirements).

- **Namenode:** Namenode is the **master** node of the system. It is used to store meta-data information about the Datanodes, (i.e) which data nodes have data, slave node details, data nodes location and timestamps. **(Answer to Question 1)**
- **Secondary Namenode:** It is responsible for performing periodic checkpoints. In the event of Namenode failure, it can restart the Namenode using the checkpoint.
- **Datanode:** Datanodes are the **slaves** nodes of the system which are deployed on each machine and provide the actual storage. They are responsible for serving read/write requests for the clients. All Datanodes periodically sends heartbeat messages to the Namenode, to say that they are alive. **(Answer to Question 1)**
- **JobTracker:** It is a service within Hadoop the schedules Map-Reduce tasks to the Datanodes in the cluster. It talks to the Namenode to determine the location of the data and also locates TaskTracker nodes with available mappers/reducer slots near the data. It submits the work for the chosen TaskTracker nodes. If the TaskTracker fails, it may resubmit the job to some-other Datanodes, which has the required data to perform the necessary operation.
- **TaskTracker:** TaskTracker resides in every Datanodes of the cluster and accepts tasks - Map, Reduce and shuffle operations from a JobTracker. The TaskTracker periodically updates the JobTracker about its progress and status.

I have used **Hadoop 1.2.1** version, for running my Terasort application. Automated scripts are provided, which automates the whole cluster. The readme.txt explains the scripts needed in automating the cluster. I have configured Hadoop in such a way that, there will be always Namenode (Master Node) / Secondary Namenode and slave/slaves in another node, such that the data to be sorted is not lost when the slave instances gets failed. It is always good to have Namenode separately as it keeps track of the meta-data information of all the slaves.

First we need to set the Environmental variables for the user. The environmental variables can be set in the .bashrc file in your home directory. This files gives the path for your Hadoop home directory, Hadoop configuration & binary files directory and the Java home directory.

Configuration files:

It is the most important step in running a Hadoop job. There are various configuration files that needs to be setup correctly before running the Hadoop job. If the configuration is wrong, Hadoop may not be started or you will encounter errors during the course of running a Hadoop job. Configuring it properly with the correct properties and values, will increase the performance of Hadoop. I installed Hadoop 1.2.1 in my home directory. So the directory path for the configuration files is **/home/ubuntu/hadoop/conf**

The following configuration files needs to be edited:

1. **hadoop-env.sh** - This file contains some environment variable settings used by Hadoop. I have set the JAVA_HOME to my directory path where java is installed. In my case, it's in **/usr/lib/jvm/java-7-oracle**
2. **core-site.xml** - This file contains configuration settings for Hadoop core (For e.g I/O) that are common to HDFS and MapReduce Default file system configuration property - fs.default.name goes in this directory. In this configuration file, I have two properties
    - **fs.default.name** - It points to NameNode URL and port
    - **hadoop.tmp.dir** - It refers to the path where the distributed file system will be located. The Distributed file system will be configured from this path.
3. **hdfs-site.xml** - This file contains configuration settings for HDFS daemons, the Namenode, the Secondary Namenode and Datanodes. In this configuration I have two properties
    - **dfs.replication** - It gives the replication factor of the data stored in HDFS. Its default value is 3. I have minimized the replication factor to 1.

    - **dfs.permission** - It gives the permissions to access the HDFS. If it is set to false, any user can have access to production data in HDFS. If it is true, permission check will be enabled. I set the property to false.

4. **mapred-site.xml** - This file contains the configuration settings for MapReduce daemons, the JobTracker and the TaskTracker. Also the number of **mappers** and **reducers** needed to perform the map-reduce operation can be set in this file. I have the following properties in this file.
    - **mapred.job.tracker** - It is hostname and port in which job tracker listens for RPC communications. In our case it points to Namenode's hostname(public DNS) and port.
    - **mapred.map.tasks** - It refers to number of mappers needed to perform the Map tasks. In our case it is set to **4**. **(Answer to Question 3)**
    - **mapred.reduce.tasks** - It refers to number of reducers needed to perform the Reduce tasks. In our case it is set to **4**. **(Answer to Question 3)**
    - **mapred.map.child.java.opts** - It refers to setting the heap-size of the mappers, when each mapper spawns a new process in the JVM. Heap size can be defined using <-Xmx <size> >>
    - **mapred.reduce.child.java.opts** - It refers to setting the heap-size of the reducers, when each reducer spawns a new process in the JVM. Heap size can be defined using <-Xmx <size> >>
    - **mapred.job.map.memory.mb -** It refers to total virtual memory needed for your map tasks.
    - **mapred.job.reduce.memory.mb -** It refers to total virtual memory needed for your reduce tasks.

Once you set the configuration files, you are ready to start Hadoop by issuing a format command on the Namenode and starting the services using **start-all.sh** script. Once the script is started, all the services of Hadoop will be active and you can execute the Hadoop job. The explanation of my source code will be in readme.txt file. Also I have attached the screenshots of my Terasort application running on Hadoop.

Once the service is started, you can view the status of the job through the Web interface

1. Namenode Web Interface - http://Namenode's Amazon Public DNS:Port number/dfshealth.jsp

The port number is 50070 by default. You can also change it in the configuration files

2. JobTracker Web Interface - http://Namenode's Amazon Public DNS:Port number/jobtracker.jsp

The port number is 50030 by default. You can also change it in the configuration files.

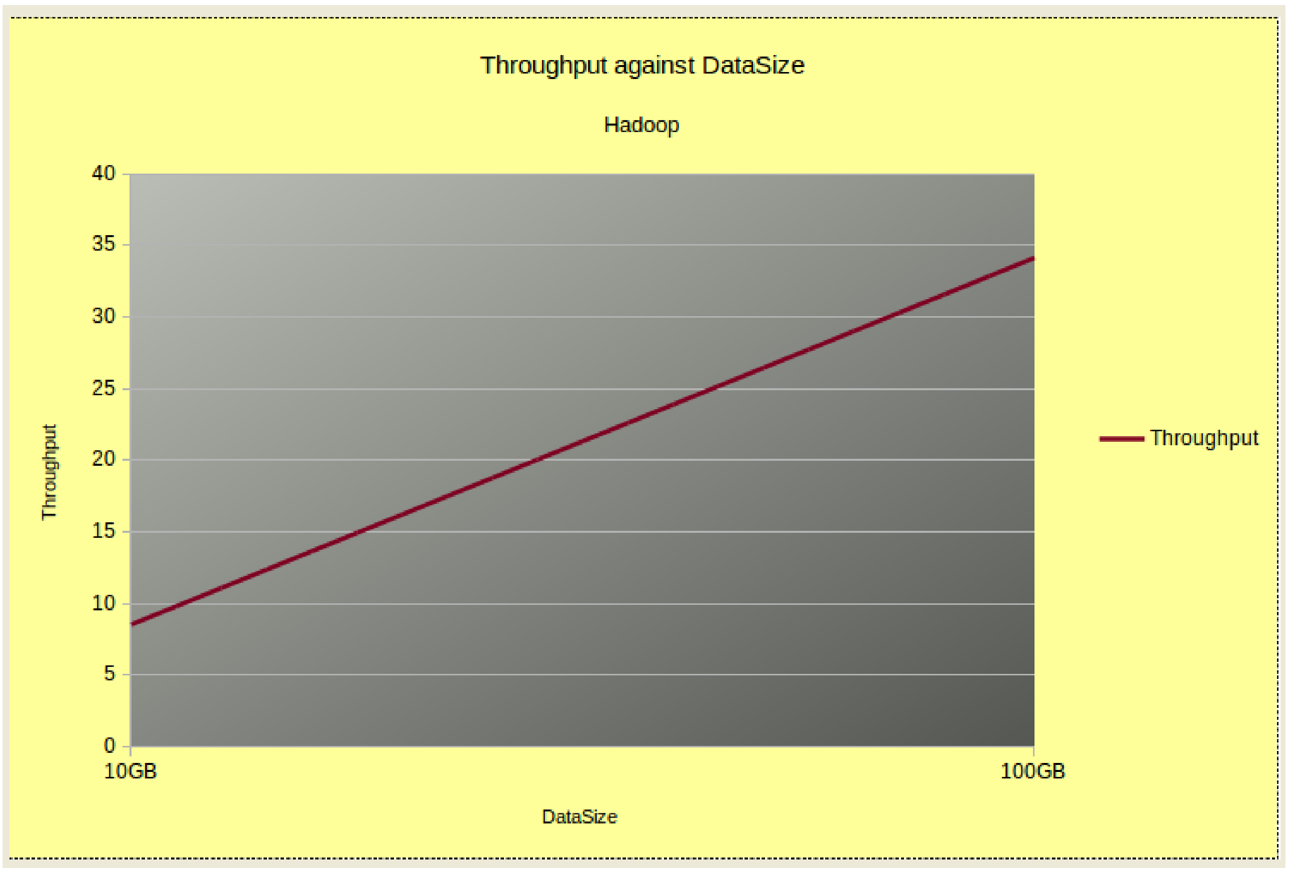
3. TaskTracker Web Interface - http://Datanode's Amazon Public DNS:Port number/tasktracker.jsp

The port number is 50060 by default. You can also change it in the configuration files.

Each of the services (Namenode, JobTracker and TaskTracker) should use unique ports for RPC communications. The Hadoop services will transfer important information like job status, heartbeat messages and the actual data between different nodes in the cluster through their designated ports. There would be port collision between different function modules if we use the same port number. **(Answer to Question 2)**

	Time (in seconds)
10 GB	**1200**
100 GB	**3000**

Throughput of 10 GB dataset in Hadoop = 8.53 MB/sec
Throughput of 100GB dataset in Hadoop = 34.13 MB/sec



X-axis: Data size
Y axis : Throughput (MB/sec)

SPARK

Apache Spark is an open source cluster computing framework. It provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It provides programmes with an application programming interface based on a data structure call **Resilient distributed dataset** (**RDD's**). It support both **batch processing** and **stream processing**. It was designed to overcome the limitation of Map-Reduce, which does not support iterative processing. Spark's RDD offers a restricted form of distributed shared memory, that has iterative processing algorithms to visit the dataset multiple-time in a loop, such that the latency can be reduced by a great magnitude compared to the Map-Reduce applications. It is greatly inspired by the machine learning systems, which formed the major thrust force for developing Spark. I have used Spark 1.6.0 version for running the terasort application. Spark is designed natively for scala language. Also python and java can be used for writing java applications.

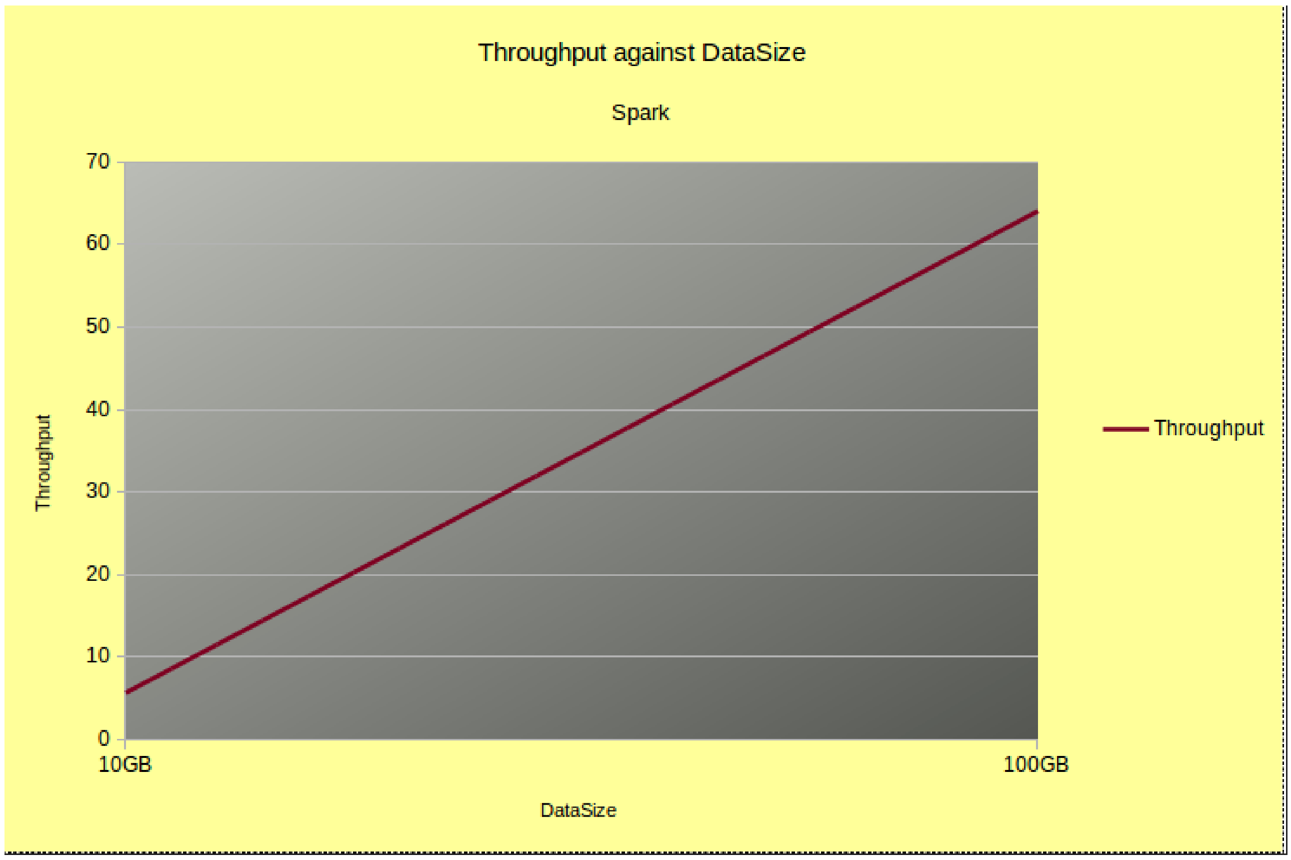
Once the service is started, can view the status of the job through the Web interface

1. Spark Master's web interface: http://<Spark Master's Amazon public DNS>:8080
2. Spark Job's web interface: http://<Spark Master's Amazon public DNS>:4040/jobs

Automated scripts for setting up the Spark cluster is provided in the script file.. Also I attached the screenshots of running my spark Terasort in AWS.

	Time (in seconds)
10 GB	**1800**
100 GB	**1600**

Throughput of 10 GB dataset in Spark = 5.68 MB/sec
Throughput of 100GB dataset in Spark= 64 MB/sec



X-axis: Data size
Y axis : Throughput (MB/sec)

SHARED MEMORY

My shared memory terasort application has three phases.

1. Divide the data to be sorted into independent chunks that can be fit into memory for performing the sorting task.

2. Sort the data of the independent chunks

3. Combine the independent chunks and write the sorted data to a file.

Since the Input Data is 100GB, I have set the total chunks as 100, so that (100 GB / 100 = 1 GB of size per chunk) fits into Memory. So with 3.75 GB of RAM in C3.large instance, 1GB of memory will utilized at any point of time. Sometimes extra objects for monitoring the heap will be created from the files, which may occupy some extra space in the memory (say 0.5 GB). So on an average 1.5 GB of data will be memory.

More detailed explanation of the code is given as in-line comments in the shared-memory source code.

Multithreading.

Multithreading will increase the parallel processing of the sorting functionality. Say for example 2 threads is used at any point of time. Each thread will operate an some independent chunks parallely. Since each thread holds a maximum of 1.5 GB of data, the memory of the instance can be fully utilized (2 * 1.5 = 3GB), which speeds up the process by more than double the times compared to single threaded approach.

Shared Memory Throughput for varying number of threads with (1GB and 10GB of data)

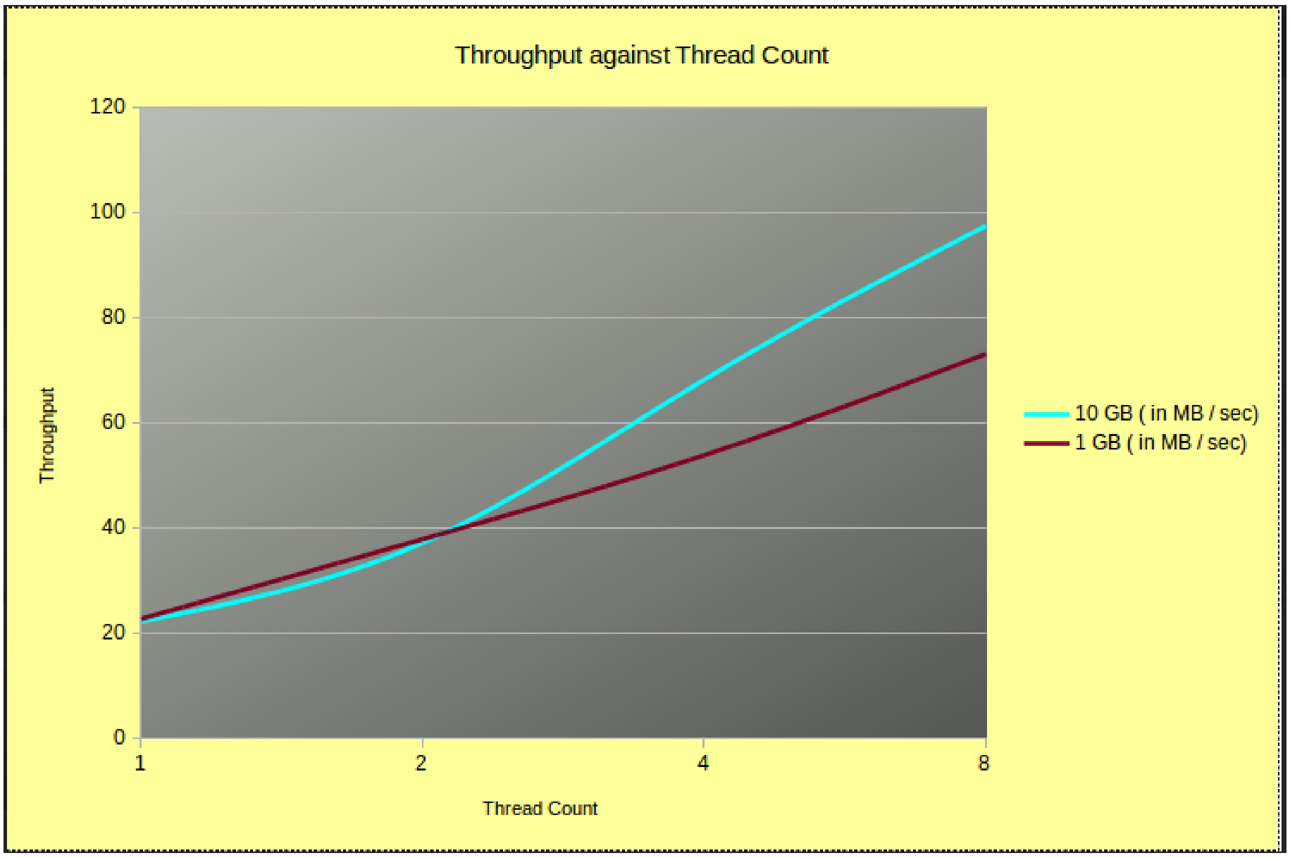
	Time(in seconds)	Time(in seconds)
Threads	**1 GB**	**10GB**
1	**45**	**460**
2	**27**	**275**
4	**19**	**150**
8	**14**	**105**

Throughput for 1 GB (in MB / sec)

1 thread - 22.75
2 thread - 37.92
4 thread - 53.89
8 thread - 73.14

Throughput for 10 GB (in MB / sec)

1 thread - 22.26
2 thread - 37.23
4 thread - 68.26
8 thread - 97.52



X AXIS - NUMBER OF THREADS
Y AXIS - THROUGHPUT (MB / seconds)

CONCLUSION:

From the graphs, it is clear that Sorting in the shared memory environment is faster than the other two frameworks. It may be useful in the shared memory system in a single node. In the multi-node system, clearly Spark is the winner. Since Spark uses iterative processing it is much faster than Hadoop in a distributed system environment for large datasets. The traditional map-reduce framework does not process the intermediate data iteratively. Also I used Hadoop 1.2.1 for processing the data-set. If I have used Hadoop 2.0 (YARN) function, the throughput performance can be improved, but it will not be as close to Spark.

In Hadoop, the performance mainly depends on the Mappers and Reducer. For high performance we can set more number of mappers and reducers. In this case, we have set to 4 in the mapred-site.xml configuration file. Also JVM heap size should be set to a appropriate value in the mapred-site.xml, so that there is no leak / out-of-space memory errors, when the data to be processed in each mapping/reducing slot is more than the capacity of the JVM.

In a multithreaded shared memory environment, as the number of threads increases, the performance increases. The performance will not increase, even if we increase the number of threads at a certain stage because, the instance has only two virtual cores and Threads will be in contention to acquire the cores, which will not increase the Throughput.

Therefore in a single node shared memory environment - Shared memory code works better.
For small data-set (10 - 50 GB) workloads in distributed systems - We can depend on Hadoop to yield good performance
For large data-set (100 GB or more) workloads in a distributed system - Clearly Spark is the winner.