International Journal of Cloud Computing and Services Science (IJ-CLOSER)
Vol.3, No.5, October 2014, pp. 283~294
ISSN: 2089-3337

Hadoop Distributed File System-Experiment and Analysis for Optimum Performance

Rajeev Kumar Gupta*, R.K. Pateriya**
Department of Computer Science & Engg , MANIT, Bhopal

Article Info

Article history:

Received Aug 30th, 2014
Revised Sept 25th, 2014
Accepted Oct 10th, 2014

Keyword:

HDFS;
NameNode;
DataNode;
JobTracker;
Metadata

ABSTRACT

The size of the data used in today's enterprises has been growing at exponential rates from last few years. Simultaneously, the need to process and analyze the large volumes of data has also increased. Hadoop is a popular open-source implementation of MapReduce for the analysis of large datasets. To manage and storage resources across the cluster, Hadoop uses a distributed user-level filesystem. This filesystem, HDFS is written in Java and designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications .This paper initially deals with the review of HDFS in details. Later on, the paper reports the experimental work of Hadoop with the big data and suggests the various factors on which Hadoop cluster shows an optimal performance. Paper concludes with providing the different real field challenges of Hadoop in recent days.

Corresponding Author:

Rajeev Kumar Gupta
Department of Computer Science & Engg ,
MANIT, Bhopal
Email: dgoyal@gyanvihar.org

1. INTRODUCTION

In recent years, with the popularization of the concept and application of cloud computing, Hadoop, an open-source cloud computing model, is focused on more and more by academia and industrial circles. Hadoop [1][7] is a basic distributed system architecture and an open-source distributed computing software with reliability and scalability researched and developed by Apache Software Foundation. In the last few years, Hadoop has been widely used by Internet corporations. With Hadoop, users can develop application programs and run applications on the composition of the cluster.

The main part of Hadoop include HDFS (Hadoop Distributed File System). A crucial property of Hadoop Framework is the capacity to partition the data and computation across many hosts (known as DataNode) and executing computations of various applications in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers.

HDFS stores its file system metadata and application data separately. As other distributed file systems, like PVFS [2][10], Lustre [11] and GFS[9][13], HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called Data Nodes. All servers are fully connected and communicate with each other using TCP-based protocols.

In this paper, we have discussed about the mechanism of HDFS and we tried to find out the different factors on which HDFS provides maximum efficiency. The remainder of this paper is organized as follows. We discuss Hadoop Architecture in Section 2. In Section 3, we discuss about file I/O operation and issue related to interaction with the clients. Section 4 discusses out two experimental works, based on which we show the

performance of HDFS on different scenario. The next part, Section 5 deals with the major challenges of Hadoop field. The conclusions are given in Section 6.

2. ARCHITECTURE

Hadoop cluster uses a Master/Slaves structure (Figure. 1).The master is responsible for NameNode and JobTracker. JobTracker's main duty is to initiate the tasks, track and dispatch their implementation. Slave is in charge of DataNode and TaskTracker [3]. TaskTracker manages the processing of local data and the collecting of result data according to the requests from applications and reports the states and performance to JobTracker [4]. NameNode and DataNode are charged with fulfilling HDFS tasks, while JobTracker and TaskTracker mainly deal with MapReduce tasks.
In this paper, we are only dealing with HDFS architecture. So, here is a brief about NameNode and DataNode. [5]

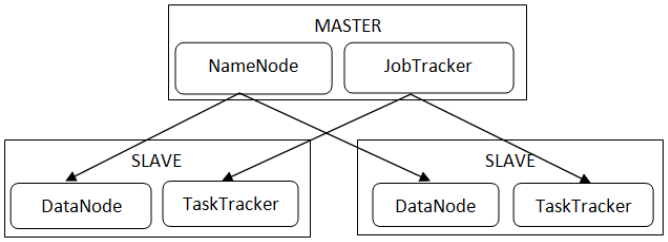

Figure 1. Master/Slaves structure of Hadoop cluster

2.1. NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (user defined, default 128MB) and each block of the file is independently replicated at multiple Data Nodes (user defined, default 3). The NameNode maintains the namespace tree and the mapping of file blocks to Data Nodes.
HDFS keeps the entire namespace in RAM. The inode data and the list of blocks belonging to each file comprise the metadata of the name system called the image. The persistent record of the image stored in the local host's native files system is called a checkpoint.

2.2. Data Nodes

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive. During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down. The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus preserving the integrity of the file system. The consistency of software versions is important because incompatible version may cause data corruption or loss, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID. After the handshake the DataNode registers with the NameNode. Data Nodes persistently store their unique storageIDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it

registers with the NameNode for the first time and never changes after that. A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to date view of where block replicas are located on the cluster. During normal operation Data Nodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes. Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions. The NameNode does not directly call Data Nodes. It uses replies to heartbeats to send instructions to the Data Nodes. The instructions include commands to:

• replicate blocks to other nodes;
• remove local block replicas;
• re-register or to shut down the node;
• send an immediate block report.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

2.3. HDFS Client

User applications access the file system using the HDFS client. HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references all the files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas. When an application reads a file, the HDFS client first asks the NameNode for the list of Data Nodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose Data Nodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new Data Nodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of Data Nodes is likely to be different. The interactions among the client, the NameNode and the Data Nodes are illustrated in Fig. 2. HDFS allows applications like, MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows a user to set the replication factor for a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increases their read bandwidth.

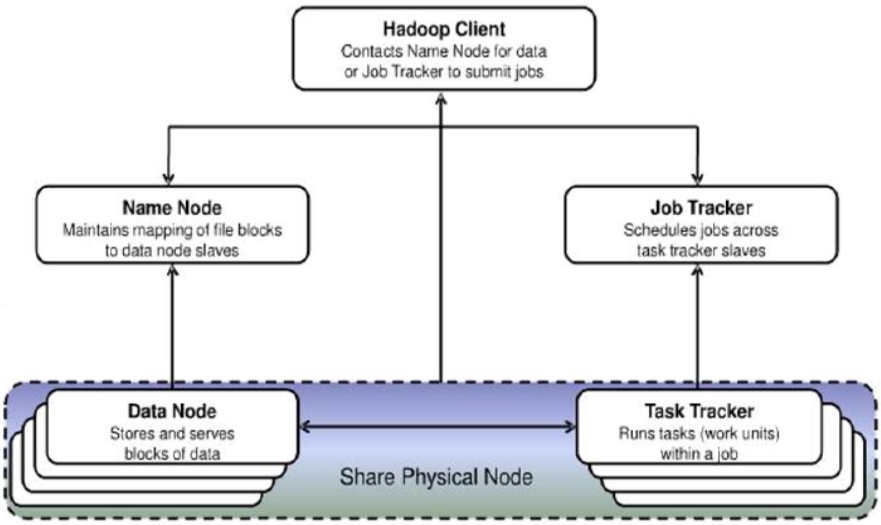

Figure 2. Hadoop high-level architecture, interaction between Client and HDFS

3. FILE I/O OPERATIONS AND MANGEMENT OF REPLICATION

3.1. File Read and Write

A client through an application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of Data Nodes to host replicas of the block. The Data Nodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgement for the previous packets.

After data are written to an HDFS file, it does not provide any guarantee that data are visible to the new reader until the file is closed. If a user application needs to see the updated file, it should explicitly call the hflush operation. The current packet is immediately pushed to the pipeline, and the hflush operation will wait until all Data Nodes in the pipeline acknowledge the successful transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.

In a cluster of thousands of nodes, failures of a node are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, Data Nodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksum in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content.

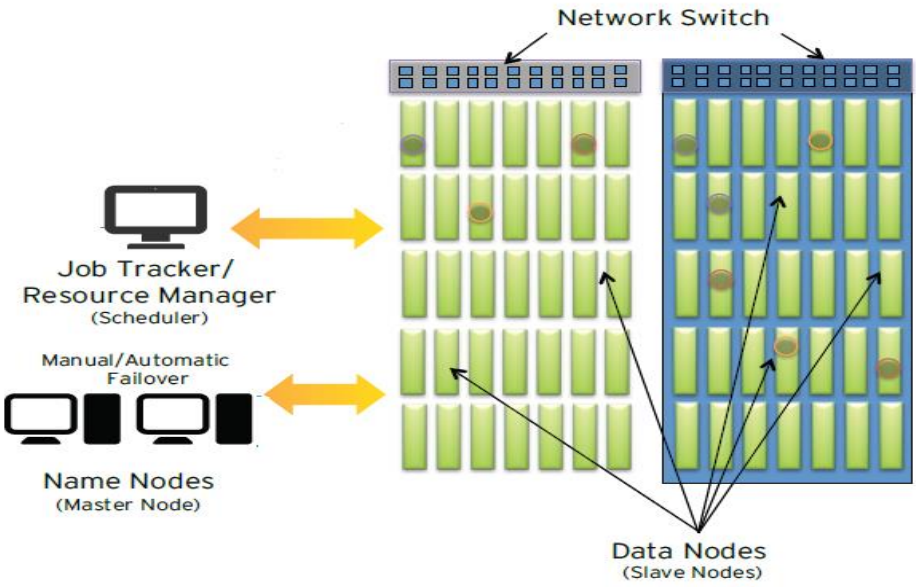

Figure 3. Racks of commodity hardware and interaction with NameNode and JobTracker

3.2. Block Placement

For a large cluster, it is never practical to connect all nodes to a particular switch. A better solution is that, the nodes of a rack should share a switch, and rack switches are connected by one or more core switches. In most of the cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks. Figure 3 shows a cluster topology, used in Hadoop architecture.
HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing up their distances to their closest common ancestor. A shorter distance between two nodes means that the greater bandwidth they can utilize to transfer data.
The default HDFS block placement policy provides a tradeoff between minimizing the write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located, the second and the third replicas on two different nodes in a different rack, and the rest are placed on random nodes with restrictions that no more than one replica is placed at one node and no more than two replicas are placed in the same rack when the number of replicas is less than twice the number of racks. The choice to place the second and third replicas on a different rack better distributes the block replicas for a single file across the cluster. If the first two replicas were placed on the same rack, for any file, two-thirds of its block replicas would be on the same rack.
In General, the HDFS follows the following replica placement policy:
1. No DataNode contains more than one replica of any block.
2. No rack contains more than two replicas of the same block, provided there are sufficient racks on the cluster.

3.3. Replication management

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability. When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of the new block placement. If the number of

existing replicas is one, HDFS places the next replica on a different rack. In case that the block has two existing replicas, if the two existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas. The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as under-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decides to remove an old replica because the over replication policy prefers not to reduce the number of racks.

4. PERFORMANCE EVALUATION

4.1. Experimental Setup

For performance characterization, a 8-node Hadoop cluster was configured. The first 5 nodes provided both computation (as MapReduce clients) and storage resources (as DataNode servers), and the 6th and 7th node served as JobTracker (Resource-Manager) and NameNode storage manager. Each node is running at 3.10 GHz clock speed and with 4GB of RAM and a gigabit Ethernet NIC. All nodes used Hadoop framework 2.4.0, and Java 1.6.0. The all nodes in configured with 500GB Hard Disk. Total configured capacity shown on NameNode web interface is 2.66TB.

4.2. Test using TestDFSIO to find the various criteria on which Hadoop shows most optimal efficiency

The test process aims at finding optimal efficiency of the performance characteristics of two different sizes of files and bottlenecks posed by the network interface. The comparison is done to check the performance between small and big files. A test o write and read between 1 GB file and 10 GB file is carried out. A total of 500 GB data is created through it. HDFS block size of 512 MB is used.
For this test we have used industry standard benchmarks: TestDFSIO
TestDFSIO is used to measure performance of HDFS as well as of the network and IO subsystems. The command reads and writes files in HDFS which is useful in measuring system-wide performance and exposing network bottlenecks on the NameNode and DataNodes. A majority of MapReduce workloads are IO bound more than compute and hence TestDFSIO can provide an accurate initial picture of such scenarios.
We executed two test for both write and read: one for 50 files each of size 10GB and other with 500 files each of size 1GB.
As an example, the command for a read test may look like:
$ hadoop jar Hadoop-*test*.jar TestDFSIO –read –nrFiles 100 –fileSize 10000
This command will read 100 files, each of size 10GB from the HDFS and thereafter provides the necessary performance measures.
The command for a write test may look like:
$ hadoop jar Hadoop-*test*.jar TestDFSIO –write –nrFiles 100 –fileSize 10000
This command will write 100 files, each of size 10GB from the HDFS and thereafter provides the necessary performance measures.
TestDFSIO generates 1 map task per file and splits are defined such that each map gets only one file name. After every run, the command generates a log file indicating performance in terms of 4 metrics: Throughput in MBytes/s, Average IO rate in MBytes/s, IO rate standard deviation and execution time.

Output of different test is given below:

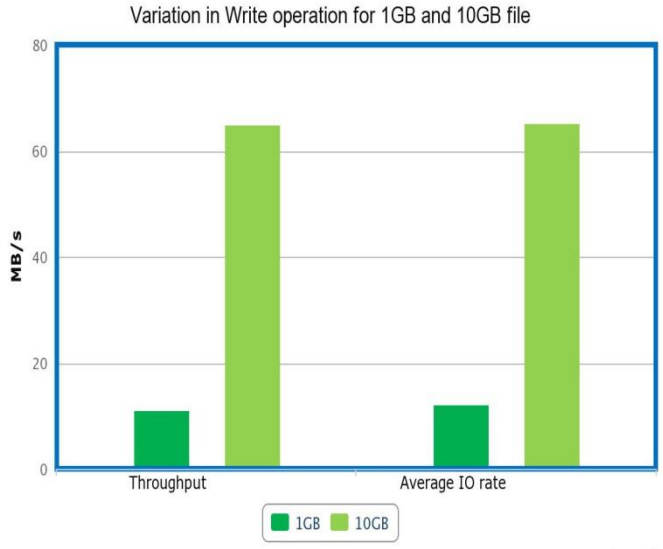

Figure 4. The write bandwidth and throughput of 10GB file size is 6 to 7 times greater than 1GB

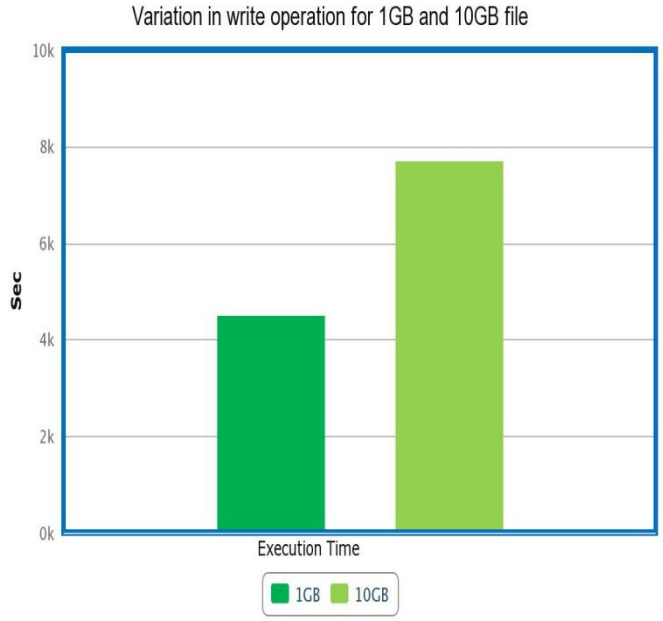

Figure 5. The same amount of write takes almost 7.5 to 8 times less time in case of 10GB

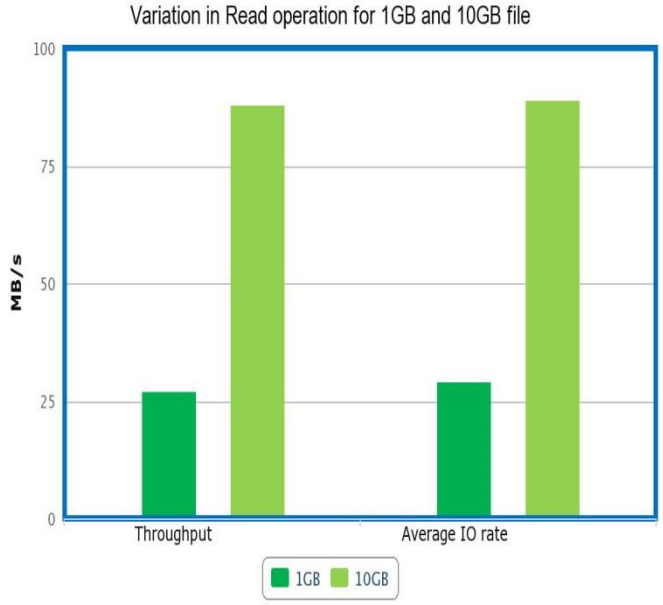

Figure 6. Reading same amount of data in 10GB filesize offers 3.4x more throughput and average bandwidth

To obtain a good picture of performance, each benchmark tool was run 3 times on each 1GB file size and results were averaged to reduce error margin. The same process was carried on 10GB file size to get data for comparison.

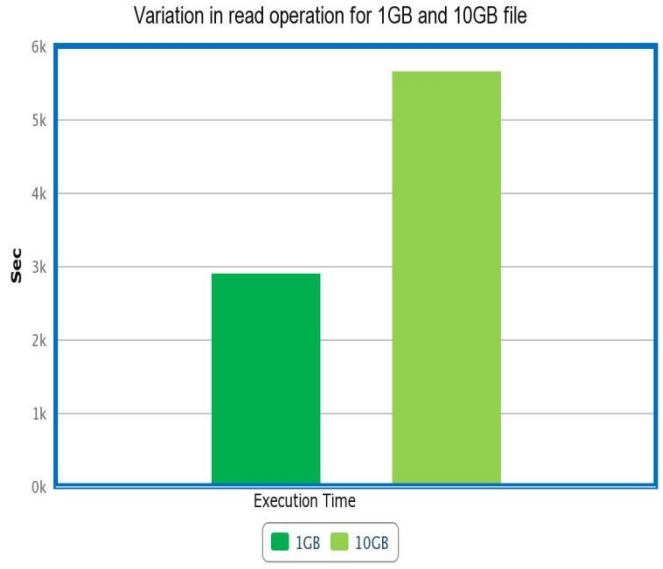

Figure 7. The same amount of read takes almost 4 times less time in case of 10GB

Experiment shows, test execution time is almost half during the 1GB file test. This the total time it takes for the Hadoop jar command to execute. From figure 4 and 6, we can visualize that, the throughput and IO Rate too shows a significant declined in terms of both write and read for the 1GB file test.
This is somewhat unexpected in nature. However, one major conclusion that we encountered is as follows:
In these tests there is always one reducer that runs after the all map tasks have complete. The reducer is responsible for generating the result set file. It basically sums up all of these values "rate,sqrate,size,etc." from each of the map tasks. So the Throughput, IO rate, STD deviation, results are based on individual map tasks and not the overall throughput of the cluster. The nrFiles is equal to number of map tasks. In the 1GB file test there will be (500/6) = 83.33 (approx) map tasks running simultaneously on each node manager node

versus 8.33 map tasks on each node in the 10GB file test. The 10GB file test yields throughput results of 64.94 MB/s on each node manager. Therefore, the 10GB file test yields (8.33 * 64.94MB/s) 540.95MB/s per node manager. Whereas, the 1GB file test yields throughput results of 11.13 MB/s on each node manager. Therefore, the 1GB file test yields (83.33 * 11.13MB/s) 927.46 MB/s per node manager. Clearly the 1GB file test shows the maximum efficiency. However increasing the no of commodity hardware eventually decreases the execution time and increased the average IO rate. It also shows how MapReduce IO performance can vary depending on the data size, number of map/reduce tasks, and available cluster resources.

4.3. Dependence of Execution time of write operation based on No of Reducers and Block size.

A word count job is submitted and experiment is carried out on a 100 GB file, varying the no of reducers keeping with block size of HDFS fixed. The experiment carried out with 4 different types of block size, viz. 512MB, 256MB, 128MB and 64MB.

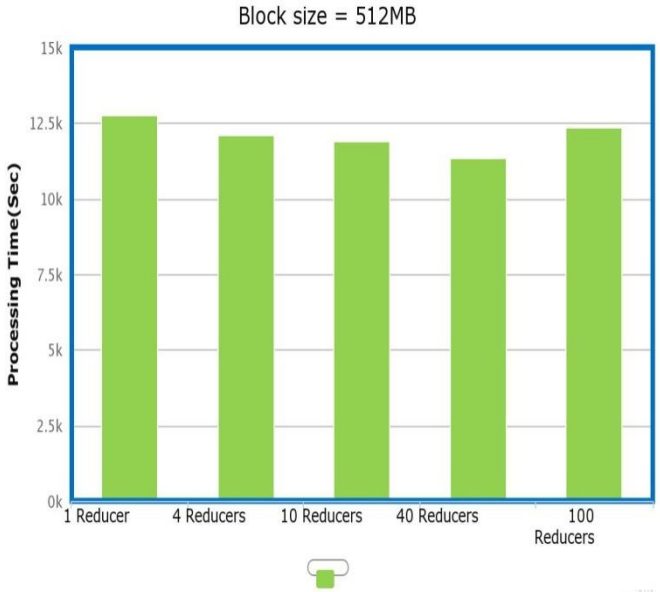

Figure 8. Variation of processing times with variation of reducers , keeping block size fixed at 512MB

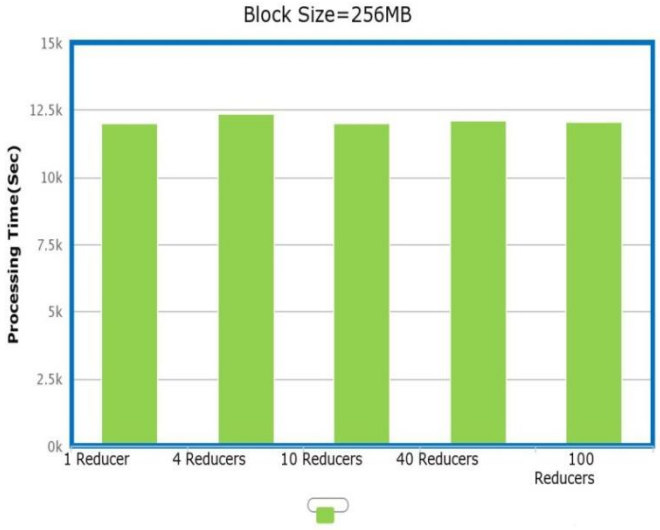

Figure 9. Variation of processing times with variation of reducers, keeping block size fixed at 256MB

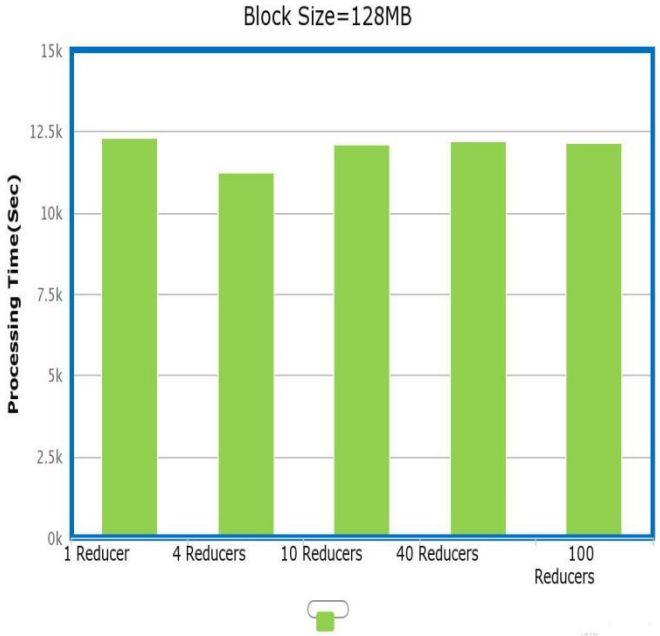

Figure 10. Variation of processing times with variation of reducers, keeping block size fixed at 128MB

Based on the Test-Report we obtained, the charts in the figure, 8, 9, 10, and 11 have been made and proper conclusion is followed.
All the above graphs appear to form a uniform straight line or show a slight negative slope which indicates that with increase in number of reducer for a give block size time for processing either remains same or reduces to some extent. Moreover, for significant large files, small change in block-size doesn't lead to change the drastic change in execution time.

5. MAJOR CHALLENGES IN HADOOP

Although the Hadoop Framework has been approved by everyone for his flexibility and faster parallel computing technology, there still are many problems which written in short in the following points [7]
(1) Single point of failure of NameNode/ JobTracker. Hadoop uses a master server to control sub servers', called slaves for the tasks to execute, leading to shortcomings like fatal single point of failure and lacking of space capacity, which seriously affect its scalability.
(2) HDFS faces huge problems, dealing with small files. HDFS data are stored in object form, and each object occupies about 200 Byte. Taking a replication factor of 3(default), it would take approximately 600 Byte. If there is a large number of these small files for storage, NameNode will request for a lot of space. That will severely restrict the scalability of the cluster. [8]

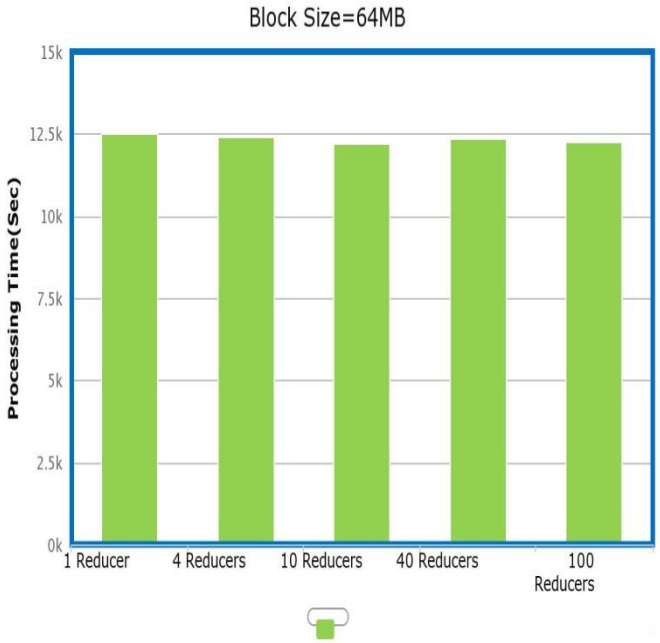

Figure 11. Variation of processing times with variation of reducers, keeping block size fixed at 64MB

3) JobTracker may at a time, become over loaded since it is responsible for monitoring and dispatching simultaneously. Research has begun to design the next generation Hadoop MapReduce to resolve this issue. The goal is to separate monitoring with dispatching and make a specialized component for monitoring, while JobTracker is only in charge of overall scheduling.
4) Data processing performance. Hadoop is similar to the database; it requires specialized optimization according to actual application needs. Many experiments show that there is still much room for the improvement of processing performance and thus improving time complexity of data for the execution of a particular job.

6. CONCLUSION AND FUTURE WORK

This section presents some of the future work that we are considering; Hadoop being an open source project justifies the addition of new features and changes for the sake of better scalability and file management.
Hadoop is recently one of the best among managing the Big Data. Still, in our experiment, we have found that, it performs poor in terms of throughput when the numbers of files are relatively larger compared to smaller numbers of files. Our experiment shows, how the performance bottlenecks are not directly attributable to application code (or the MapReduce programming style), but rather are caused by the numbers of data nodes available, size of files in used in HDFS and also it depends on the number of reducers used.
However, the biggest issue on which on are focusing is the scalability of Hadoop Framework. The Hadoop cluster becomes unavailable when its NameNode is down.
Scalability of the NameNode [12] has been a key struggle. Because the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap has limited the number of files and also the number of blocks addressable. The main challenge with the NameNode has been that when its namespace table space becomes close the main memory of NameNode, it becomes unresponsive due to Java garbage collection. This scenario is bound to happen because the numbers of files used the users are increasing exponentially. Therefore, this is a burning issue in the recent days for Hadoop.
Currently, we are using two NameNode, so that the block ID gets divided between them, but this does not ensure a permanent solution to the cause. We need a dynamic NameNode configuration, that automatically distributes block ID among the NameNodes when the used space becomes close to the RAM.

7. ACKNOWLEDGMENT

We would like to thank National Institute of Technology, Silchar for granting us the laboratory. That contributes a lot to my research, for that I got the opportunity to configure the Hadoop.

REFERENCES

[1]. Apache Hadoop. http://hadoop.apache.org/
[2]. P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. "PVFS: A parallel file system for Linux clusters," in Proc. of 4th Annual Linux Showcase and Conference, 2000, pp. 317–327
[3]. Hong Sha, Yang Shenyuan. Key technology of cloud computing and research on cloud computing model based on Hadoop [J]. Software Guide, 2010, 9(9): 9~11.
[4]. Xie Guilan, Luo shengxian. Research on applications based on Hadoop MapReduce model [J]. Microcomputer & its applications, 2010, (8): 4~7.
[5]. Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The Hadoop Distributed Filesystem: Balancing Portability and Performance. Performance Analysis of Systems & Software (ISPASS),2010 122 – 133
[6]. J. Venner, Pro Hadoop. Apress, June 22, 2009.
[7]. Xin Daxin, Liu Fei. Research on optimization techniques for Hadoop cluster performance [J]. Computer Knowledge and Technology, 2011,8(7):5484~5486
[8]. Lu, Huang ; Hai-Shan, Chen ; Ting-Ting, Hu : Research on Hadoop Cloud Computing Model and its Applications. Networking and Distributed Computing (ICNDC), 2012, 59 – 63
[9]. S. Ghemawat, H. Gobioff, S. Leung. "The Google file system," In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29–43.
[10]. W. Tantisiriroj, S. Patil, G. Gibson. "Data-intensive file systems for Internet services: A rose by any other name ..." Technical Report CMUPDL- 08-114, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, October 2008
[11]. Lustre File System. http://www.lustre.org
[12]. K. V. Shvachko, "HDFS Scalability: The limits to growth," ;login:. April 2010, pp. 6–16
[13]. M. K. McKusick, S. Quinlan. "GFS: Evolution on Fast-forward," ACM Queue, vol. 7, no. 7, New York, NY. August 2009