Chapter 3
A Deep Dive into the Hadoop World to Explore Its Various Performances

Dipayan Dev and Ripon Patgiri

Abstract Size of the data used in todays 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. To handle and for analysis of large scale datasets, an open-source implementation of Apache framework, Hadoop is used now-a-days. For managing and storing of all the resources across its cluster, Hadoop possesses a distributed file system called Hadoop Distributed File System (HDFS). HDFS is written completely in Java and is depicted in such a way that in can store Big data more reliably, and can stream those at high processing time to the user applications. Hadoop has been widely used in recent days by popular organizations like Yahoo, Facebook and various online shopping market venders. On the other hand, experiments on Data-Intensive computations are going on to parallelize the processing of data. None of them could actually achieve a desirable performance. Hadoop, with its Map-Reduce parallel data processing capability can achieve these goals efficiently. This chapter initially provides an overview of the HDFS in details. The next portion of the paper evaluates Hadoops performance with various factors in different environments. The chapter shows how files less than the block size affect Hadoops R/W performance and how the time of execution of a job depends on block size and number of reducers. Chapter concludes with providing the different real challenges of Hadoop in recent days and scope for future work.

Keywords Hadoop · Big data · HDFS · Small files · Map-Reduce

D. Dev (✉) · R. Patgiri
Department of Computer Science, NIT Silchar, Silchar, India
e-mail: dev.dipayan16@gmail.com

R. Patgiri
e-mail: ripon@cse.nits.ac.in

B.S.P. Mishra et al. (eds.), *Techniques and Environments for Big Data Analysis*,
Studies in Big Data 17, DOI 10.1007/978-3-319-27520-8_3

3.1 Introduction

The last few years of internet technology as well as computer world has seen a lot of growth and popularity in the field of cloud computing [9, 11]. As a consequence, the cloud applications have given birth to Big data. Hadoop, an open source distributed system made by Apache Software Foundation, has contributed hugely to handle and manage such Big data [2]. Hadoop [1] has a master-slave architecture, provides scalability and reliability to a great extent. In the last few years, this framework is extensively used and accepted by different cloud vendors as well as Big data handler. With the help of Hadoop, a user can deploy programs and execute processes on the configured cluster.

The main parts of Hadoop include HDFS (Hadoop Distributed File System) [1, 13] and Map-Reduce paradigm [6]. A crucial property of Hadoop Framework is the capacity to partition the computation and data among various nodes (known as Data Node) and running the computations in parallel. Hadoop increases the storage capacity, I/O BW and other computation capacity by adding commodity servers.

HDFS keeps application data and file systems meta-data [7, 23] in different servers. Like popular DFS, viz. PVFS [4, 20], Lustre [15] and GFS [10, 16], HDFS too saves its meta-data in a server, known as Name Node. Rest of the application data are kept on the slaves server, known as Data Nodes.

The main purpose of the paper is to focus and reveal the various criteria that influence the efficiency of Hadoop cluster. None of the previous papers demonstrated this kind of work that exposes the dependencies of Hadoop efficiency.

In this chapter, 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 chapter is organized as follows. We discuss Hadoop Architecture in Sect. 3.2. In Sect. 3.3, we discuss about file I/O operation and interaction of Hadoop with the clients. Section 3.4 discusses three experimental works to evaluate the crucial factors on which Hadoop clusters performance depends. The next part, Sect. 3.5 deals with the major challenges of Hadoop field. The conclusions and future work in the concerned issues are given in Sect. 3.6.

3.2 Related Work

Evaluation of performance in the field of Hadoop and different large-scale file systems have been carried out many times [17, 18, 21]. HDFS performance is analyzed in [18] whose results shows that HDFS performs poor because of the various delays in task scheduling, fragmentation, huge disk seeks caused by disk contention under excessive workloads. The performance of Hadoop relies heavily on the operating system as well as on the algorithms that were employed by the disk scheduler and various allocators.

In [21], the authors tried to integrate PVFS with Hadoop and they compared the performance with Hadoop Distributed File System using some sets of benchmarks. They paper indicates various optimizations of Hadoop that can be matched with PVFS and how durability, consistency and persistent tradeoffs made by these large-scale file system effect the cluster performance. Their results showed that, PVFS performance is as good as HDFS in Hadoop Framework.

The authors of the paper [17], analyzed the performance of BlobSeerDFS with HDFS. Their result demonstrated that, BlobSeerDFS achieved higher throughput when compared to HDFS.

In our work, we have evaluated the performance of HDFS with various parameters considering different sizes of file sizes. Compared with the conference version in [8], this chapter describes the Hadoop architecture in little more depth and extra simulation is carried out to portrait HDFS read and write behavior with different sizes of files.

3.3 Architecture of Hadoop

Hadoop framework uses pure Master/Slaves architecture (Fig. 3.1). The master nodes are given the responsibility of Name Node and Job Tracker. The main duty of Job-Trackers is to initiate tasks, track and dispatch their implementation. The charge of Data Node and Task Tracker is given to Slave nodes. The main role of TaskTracker is to process of local data and collection of all these result data as per the request received from applications and then report the performance in periodic interval to JobTracker [12]. HDFS, which seems to be heart of Hadoop, all of its charges, are given to NameNode and DataNode for fulfilling, while JobTracker and TaskTracker mainly deal with Map-Reduce application.

In this chapter, we are only dealing with HDFS architecture. So, here is a brief description about Name Node and Data Node. A short description of the client interaction with the Name Node and Data Node is also portrayed.

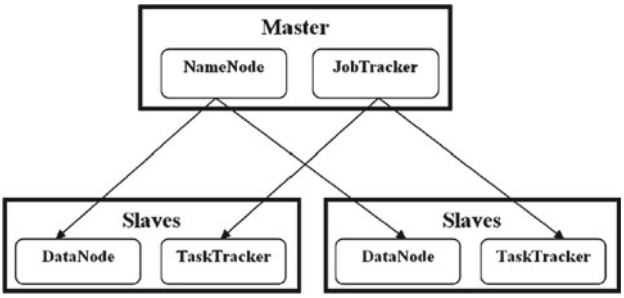

Fig. 3.1 Master-slave architecture of Hadoop cluster

3.3.1 NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by i-nodes, 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 128 MB) 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. For, faster execution of the cluster operations, HDFS has to keep the whole namespace in its Main Memory. I-node data and other list of block, which belong to each file, constitute the meta-data of the name system, termed as FSImage. The un-interrupted record of the FSImage stored in the Name Nodes local filesystem is termed as checkpoint.

3.3.2 DataNodes

The block replica stored on Data Nodes is regarded as two files in the local hosts own file system. The first one constitutes the main data and second file acts as storage for meta-data of blocks. During startup each Data Node get connect to the Name Node. The phenomenon is just like a normal handshake. The purpose of this type of handshake is to verify the namespaceID and to check whether there is a mismatch between the software versions of the Data Nodes. If either does not match the same with the Name Node, that particular Data Node gets automatically shut down. A namespaceID is assigned to the file system when it is formatted each time. The namespaceID is persistently stored on all nodes of the cluster. A node with a different namespaceID will not be able to join the cluster, thus maintaining the integrity of the file system. A Data Node that is newly initialized and without any namespaceID is permitted to join the cluster and receive the clusters namespaceID. After the handshake is done, the Data Node registers with the Name Node. A storageID is allotted to the Data Node for the first time, during the registration with Name Node. Data Node periodically sends block report to the Name Node to identify the block replicas in its control. The report consists of block id, the length of each block etc. During normal operations, all the Data Nodes periodically send a response to the Name Node to confirm that, it is alive and active in operation and all the different block replicas it stores are available. If the Name Node does not receive a heartbeat from a Data Node in 10 min the Name Node considers the Data Node to be dead and the block replicas stored at that Data Node becomes inaccessible. The Name Node then schedules the things again and allocates all of those blocks to other Data Nodes, which is selected randomly.

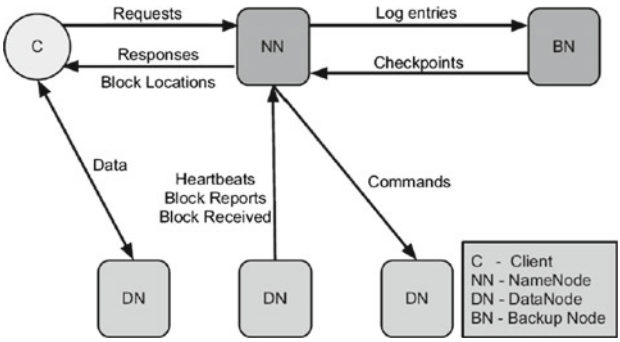

Fig. 3.2 Hadoop high-level architecture, interaction of clients with HDFS

3.3.3 HDFS Client Interaction with Hadoop

This part explains the interaction among client and Name Node and Data Node. HDFS client is the media via which a user application accesses the file system. For faster access, the namespace is saved in the Name Nodes RAM. A user references all the files and directories by paths in the namespace. It remains unknown to an application that, that file system meta-data and application data are put on separate servers as well as about the replication of blocks. During the time, an application reads a file for the first time; HDFS client gets response from the Name Node of all lists of Data Nodes that stores replicas of the blocks of that particular file. The client application checks the lists out, then a Data Node is contacted directly and requests to transfer the desired block for reading. When a client wants to write in the HDFS, it first sends a query to the Name Node to choose Data Nodes for hosting replicas of the blocks of the file. When the client application, get response from the Name Node, it start searches for that given Data Node. When if finds, the first block is filled or doesnt have enough disk space, the client requests new Data Nodes to choose for hosting replicas for the next block. Like this way, the process continues. The detail interactions among the client, the Name Node and the Data Nodes are illustrated in Fig. 3.2.

3.4 File I/O Operations and Management of Replication

3.4.1 File Read and Write

This part of paper, describes the operation of HDFS for different I/O file operations. When a client wants to write some data or add something into HDFS, he has to do the taskthrough an application. HDFS follows a single-writer, multiple-reader model [19]. When the application closes the file, the content already written cannot

be manipulated or altered. However, new data can be appended in to by reopening the file All HDFS client, when tries to open a file for writing into it, is granted a lease for it; no other client can write to the file. The writing application periodically can renew the given lease by sending heartbeats to the Name Node. When the file is closed down, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. The writer is granted an exclusive access for the file, as long as the soft limit persists. After the soft limits expiry comes and even then the client does not able to close the file or make renew of the lease, another client can pre-empt the lease.

The client application is also grant a hard limit of one hour. When this hard limits expiry time arrives, and here also if the client fails to renew the hard limit, HDFS taken it for granted that the client has left the network. HDFS then automatically closes that file on behalf of the writer and recovers the lease granted for it. The lease provided to a writer never does prevent any other client to read that file. HDFS follows a read by simultaneous reader at a time scheme.

An HDFS file can be defined as a collection of chunks of data or blocks. When a client application wants for a new block, Name Node does an allocation for the block, specified with a unique blockID. It then searches for a list of Data Nodes that can host the replicas of the block. Data Nodes, that act as a pipeline, generally possess has a tendency to minimize the total network distance of the last Data Node from the client.

HDFS treats all the Bytes as a sequence of packets. Bytes written by a client program is stored as a buffer. After the packet buffer is filled up (Usually 64KB), all of these are pushed Data Nodes pipeline.

When packets of bytes (data) are written into the HDFS, it never assures the reader, that he can read the file, until and unless the file is closed. There is a hush operation provided by the Hadoop, which a user application explicitly uses to see the updated file. Immediately the current packet gets pushed to the Data Nodes, and the hush operation waits until all the Data Nodes, standing in the pipeline acknowledge the successful transmission of the packet. Hadoop clusters consist of thousands of nodes. So, it is quite natural to come across failure frequently. The replicas that are stored in the Data Nodes might become corrupted as a result of memory faults, disks or several network issues. Checksums are verified by the HDFS clients to check the integrity of the data block of a HDFS file. During reading, it helps for the detection of any corrupt packet in the network. All the checksums are stored in a meta-data file in the Data-Nodes system, which is different from the blocks data file During the reading of files by HDFS, all block data and checksums are transferred to the client. It then calculates the checksums for the data and confirms that the newly calculated checksum matched with the one it received. If it doesnt match, the client informs the Name Node about the damaged replica and brings another replica from different Data Node [19].

When a file is opened by the client for reading, the client first fetches the whole list of blocks the different location of the replicas from the Name Node. The location of each block is sorted by the distance from the client. In the process of reading the content of the block, the client posses a property to read the closest replica first. If this attempt fails, it tries for the next one in sequence. If the Data Node is not available

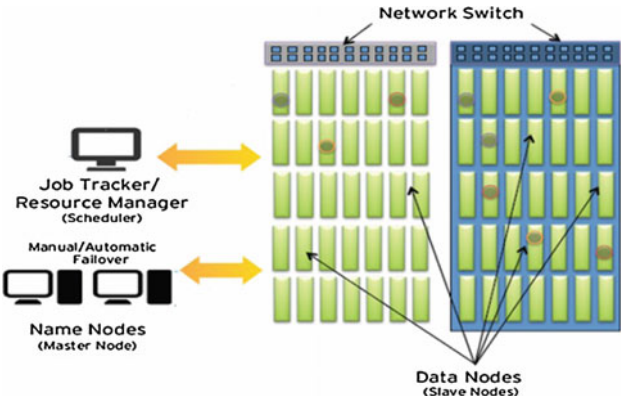

Fig. 3.3 Racks of commodity hardware and interaction with NameNode and JobTracker

or if the block replica is found to be corrupted (checksum test), the read operation then fails completely for that Data Node.

HDFS can permit a client to read the content of a file, even if it is opened for writing. But, while reading, the length of the last block, which is being written at that point of time, remains unknown to the Name Node. In this kind of case, the client asks one of the replicas for the longest length before starting to read the content.

3.4.2 Block Placement

When we want to setup a large cluster, it is never a good idea to connect all nodes to a particular switch. A better solution is that nodes of a rack should share a common switch, and switches of the rack are connected by one or more switches. The configuration of commodity hardware is done in such a way that, the network bandwidth between two nodes in the same rack is greater than network bandwidth between two nodes in different racks. Figure 3.3 shows a cluster topology, used in Hadoop architecture.

HDFS calculates the network bandwidth between two nodes by measuring the distance between them. Generally, the distance from a node to its parent node is always measured as one. A distance between two nodes can be measured by just adding up their distances to their closest node. For greater bandwidth, we should have shorter distance between two nodes. This eventually increases the capacity to transfer data.

The main concern of HDFS block replacement policy is minimization of write cost, and maximization of data reliability, scalability and increase the overall bandwidth of the cluster.

After the generation of new block, HDFS searches for a location where the writer is placed and places the first replica on that node, the 2nd and 3rd replicas are stored similarly on two different nodes in a different rack, and the rest are placed on random nodes. HDFS provides a restriction that, more than one replica cannot be stored at one node and more than two replicas cannot be stored in the same rack. The mechanism of storing the 2nd and 3rd replicas, each on different rack provides better replication management of the block replicas for a single file across the cluster.

Summing up all the above policy, the HDFS follows the following replica placement policy:

1. A single Data Node does not contain more than one replica of any block.
2. A single rack never contains more than two replicas of the same block, given then there is significant number of racks in the cluster.

3.4.3 Replication Management

The Name Node attempts to make it sure that each block of the files always has some significant number of replicas stored in the Data Nodes. Data Nodes periodically send its block report to Name Nodes. Verifying the report, the Name Node detects whether a block is under or over replicated. When Name Nodes finds it to be over replicated, it chooses a replica from any random Data Node to remove. Generally the Name Node does not prefer to reduce the number of racks which has available number of host replicas, and mainly prefers to remove from that Data Node, which has the least disk space available. The main purpose is balancing the storage utilization across all the Data Nodes, without hampering the block availability. There is a replication priority queue, which stores the blocks that is under replicated. Highest priority factor is given to the blocks, which has only one replica. On the other hand, blocks having more than two third of the specified replication factor are given the lowest priority. A thread running in background, search the replication priority queue to determine the best position for the new replicas. Block replication of HDFS also follows the same kind of policy like that of new block placement. If HDFS finds the number of existing replica of a block is one, it searches for that block and places the next replica on a rack which is different from the existing one. In case, if two replica of a block are found to be in a particular rack, the third one is kept on a different rack. Basically, the main motive here is minimizing the cost of creation of new replicas of blocks. The Name Node also does proper distribution of the block to make sure that all replicas of a particular block are not put on one single rack. If there is a situation comes, that the Name Node detects that a blocks replicas end up storing itself at one common rack, the Name Node treats it as under-replicated and eventually allocate that particular block to a different rack. When the Name Node receives the notification that a replica is created on different node, the block becomes again becomes over- replicated. So, following the previous policy written above, the Name Node, at that situation, decides to remove an old replica chosen randomly.

3.5 Performance Evaluation

In this section, the performance of Hadoop Distributed File System is evaluated in order to conclude the efficiency dependence of a Hadoop cluster.

3.5.1 Experimental Setup

For performance characterization, a 46-node Hadoop cluster was configured. The first 44 nodes provided both computation (as Map-Reduce clients) and storage resources (as Data Node servers), and the rest two nodes served as Job Tracker (Resource-Manager) and NameNode storage manager. Each node is running at 3.10 GHz clock speed and with 4 GB of RAM and a gigabit Ethernet NIC. All nodes used Hadoop framework 2.6.0, and Java 1.7.0. Ubuntu 14.04 [22] is used as out Operating System.

3.5.2 Test Using TestDFSIO to Evaluate Average I/O and Throughput of the Cluster

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 of 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 Map-Reduce workloads are IO bound more than compute and hence TestDFSIO can provide an accurate initial picture of such scenarios.

We executed two tests for both write and read: one for 50 files each of size 10 GB and other with 500 files each of size 1 GB.

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 10 GB 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 10 GB from the HDFS and thereafter provides the necessary performance measures.

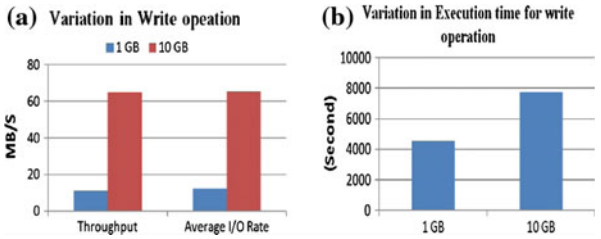

Fig. 3.4 Evaluation of write operation (**a**) the write bandwidth and throughput of (**b**) the same amount of write takes almost 10 GB file size is almost 6 times greater than 1.71 more time in case of 10 and 1 GB

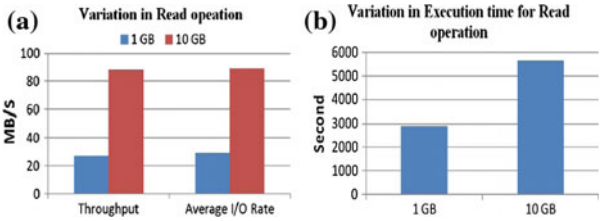

Fig. 3.5 Evaluation of read operation (**a**) reading same amount of data in 10 GB file- (**b**) the same amount of read takes almost 1.95 size offers close to 4 times more throughput and more time in case of 10 GB average bandwidth

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 tests are given in Figs. 3.4 and 3.5.

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

Experiment shows, test execution time is almost half during the 1 GB file test. This the total time it takes for the Hadoop jar command to execute.

From Figs. 3.4a and 3.5a, we can visualize that, the throughput and IO Rate too shows a significant declined in terms of both write and read for the 1 GB 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 1 GB 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 10 GB file test. The 10 GB file test yields throughput

results of 64.94 MB/s on each node manager. Therefore, the 10 GB file test yields (8.33 × 64.94 MB/s) = 540.95 MB/s per node manager. Whereas, the 1 GB file test yields throughput results of 11.13 MB/s on each nodemanager. Therefore, the 1 GB file test yields (83.33 × 11.13 MB/s) = 927.46 MB/s per node manager.

Clearly the 1 GB 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.

3.5.3 Dependence of Execution Time of Write Operation 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. 512, 256, 128 and 64 MB.

Based on the Test-Report we obtained, the charts in the Figs. 3.6, 3.7, 3.8 and 3.9 have been made and proper conclusion is followed.

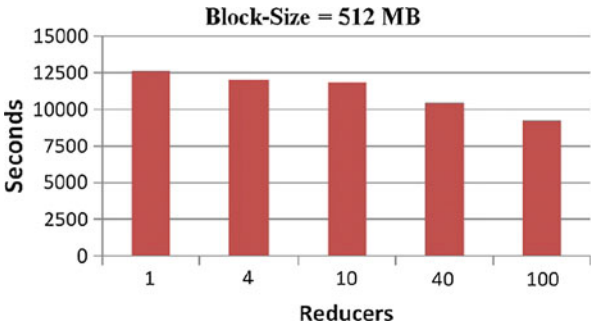

Fig. 3.6 Variation of processing times with variation of reducers, keeping block size fixed at 512 MB

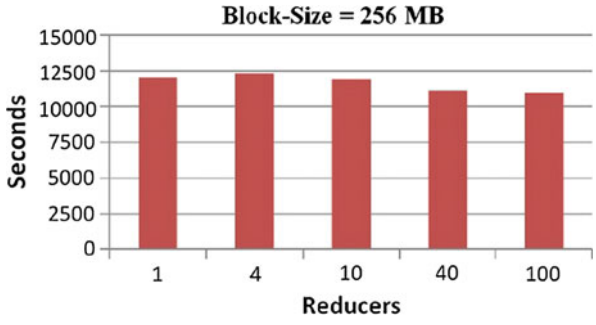

Fig. 3.7 Variation of processing times with variation of reducers, keeping block size fixed at 256 MB

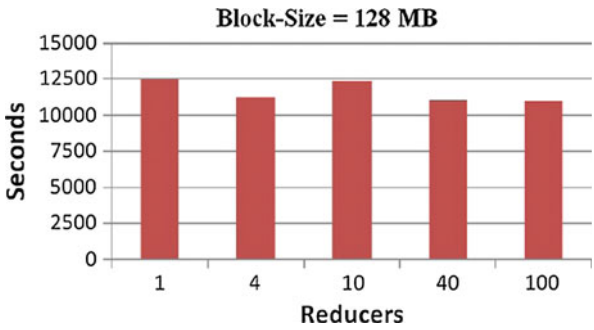

Fig. 3.8 Variation of processing times with variation of reducers, keeping block size fixed at 128 MB

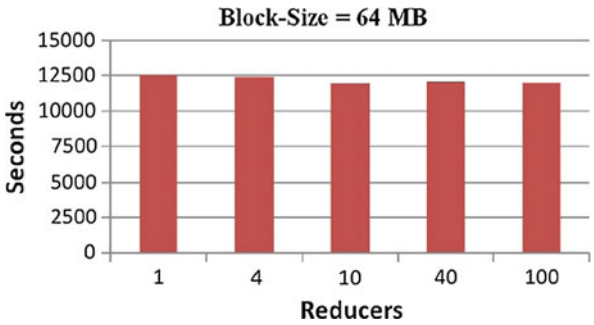

Fig. 3.9 Variation of processing times with variation of reducers, keeping block size fixed at 64 MB

All the above graphs appear to form a uniform straight line or in some it shows 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. But, a significant negative slope is visible in Fig. 3.6, where block size equals 512 MB. On the other hand, in Figs. 3.8 and 3.9, the execution times are unpredictable and show an unexpected behavior.

It can be concluded that, for large block size, the reducers play an important role in the execution time of a Map-Reduce job. For smaller block size, the change in number of reducers doesnt bring noticeable changes in the execution time.

Moreover, for significant large files, small change in block-size doesnt lead to change the drastic change in execution time.

3.5.4 Performance Evaluation of Read and Write Operations in HDFS Varying Number of Files and Sizes

The write operation of HDFS is carried out for the different data of sizes 1, 2, 4 and 8 TB as shown in Figs. 3.10, 3.11, 3.12 and 3.13 and Tables 3.1, 3.2, 3.3 and 3.4 respectively. The block size of HDFS is kept at 64 MB for all the experiments in this subsection. In all the four figures a similar trend is observed. Figures show that,

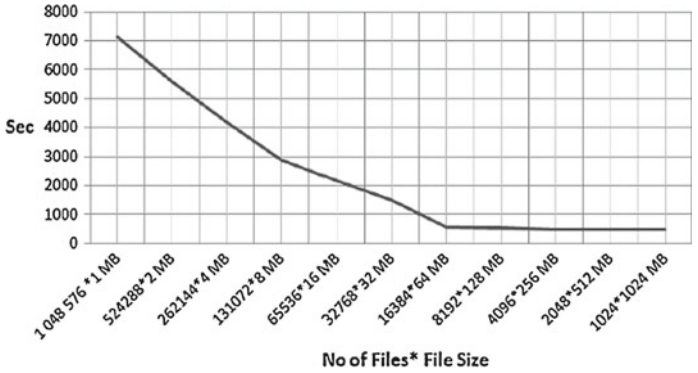

Fig. 3.10 Variation of write operation for 1 TB data size

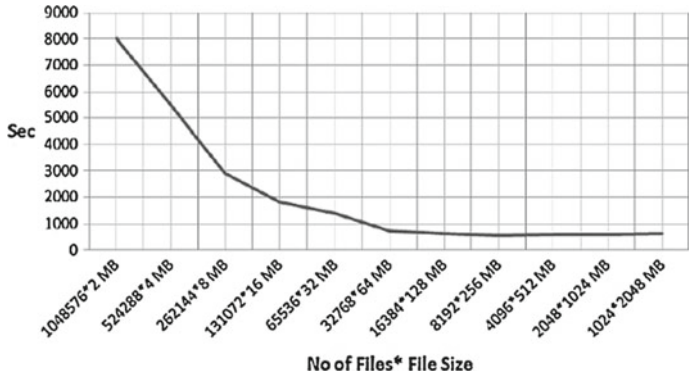

Fig. 3.11 Variation of write operation for 2 TB data size

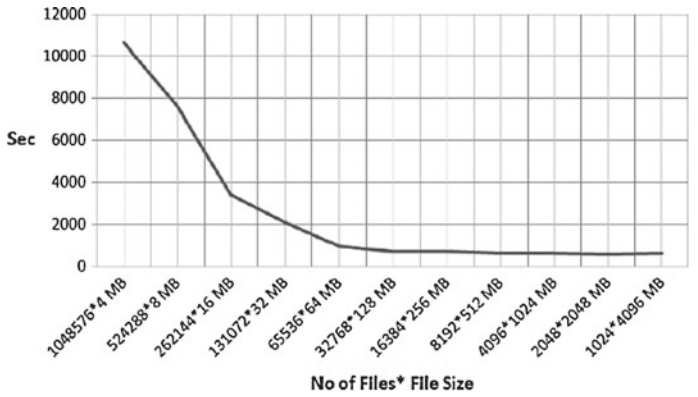

Fig. 3.12 Variation of write operation for 4 TB data size

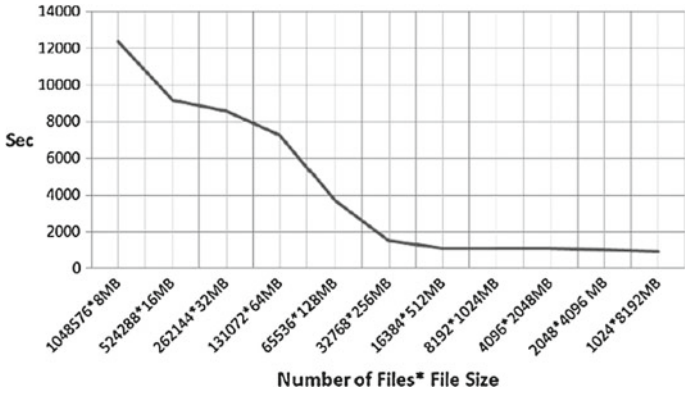

Fig. 3.13 Variation of write operation for 8 TB data size

Table 3.1 Execution time of write operation for 1 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	1	7140.68
2	524288	2	5590.058
3	262144	4	4182.967
4	131072	8	2883.34
5	65536	16	2171.219
6	32768	32	1490.87
7	16384	64	562.991
8	8192	128	544.897
9	4096	256	461.15
10	2048	512	459.18
11	1024	1024	462.07

HDFS performance is significantly poor when the file size is smaller than current block size (64 MB is our case). The execution times of the files for the write operation show a sharp decline when the size is greater than the block size.

In Figs. 3.14, 3.15, 3.16 and 3.17 and Tables 3.5, 3.6, 3.7 and 3.8, the performance of read operation for the different data of sizes 1, 2, 4 and 8 TB are shown respectively. Figure 3.14 indicates that, HDFS is taking much more time for reading 1 TB data when the file size is less than 64 MB. Whereas, when the size of the files is greater than the block size, HDFS requires much less time to read the data. Similar kind of scenario is observed in Figs. 3.15, 3.16 and 3.17.

Table 3.2 Execution time of write operation for 2 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	2	8040.32
2	524288	4	5482.437
3	262144	8	2911.926
4	131072	16	1829.159
5	65536	32	1382.462
6	32768	64	709.462
7	16384	128	623.556
8	8192	256	568.708
9	4096	512	610.539
10	2048	1024	601.506
11	1024	2048	618.796

Table 3.3 Execution time of write operation for 4 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	4	10642.859
2	524288	8	7601.318
3	262144	16	3407.575
4	131072	32	2083.141
5	65536	64	962.719
6	32768	128	723.435
7	16384	256	707.912
8	8192	512	630.882
9	4096	1024	611.868
10	2048	2048	602.868
11	1024	4096	618.948

3.6 Major Challenges in Hadoop Framework

Although the Hadoop Framework has been approved by everyone for its flexibility and faster parallel computing technology [5], there still are many problems which written in short in the following points:

1. Hadoop suffers from a irrecoverable failure called Single point of failure of Name Node. Hadoop possesses a single master server to control all the associated sub servers (slaves) for the tasks to execute, that leads to a server shortcomings like single point of failure and lacking of space capacity, which seriously affect its scalability. During the later versions of Apache Hadoop, they came out with a Secondary NameNode [14, 24] to deal with this problem.The secondary Name Node periodically check NameNodes namespace status and merges the fsimage

Table 3.4 Experimental results of write operation for 8 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	8	12360.926
2	524288	16	9161.401
3	262144	32	8592.253
4	131072	64	7265.476
5	65536	128	3684.956
6	32768	256	1514.76
7	16384	512	1092.849
8	8192	1024	1104.974
9	4096	2048	1097.671
10	2048	4096	981.7
11	1024	8192	889.511

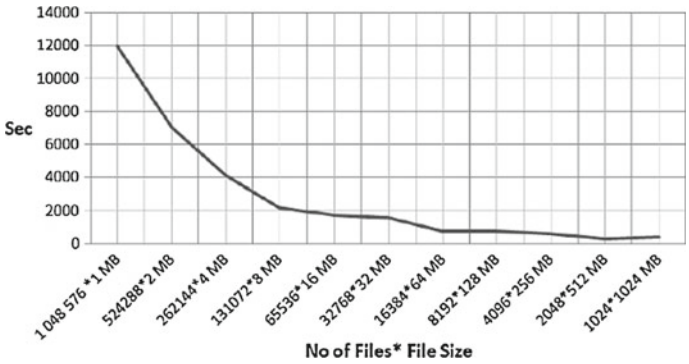

Fig. 3.14 Variation of read operation for 1 TB data size

with editlogs. It decreases the restart time of NameNode. But unfortunately is not a hot backup daemon of NameNode, not fully capable of hosting DataNodes in the absence of NameNode. So, could not resolve the SPOF of Hadoop.

2. As our experimental results show, HDFS faces huge problems, dealing with small files. HDFS data are stored in the Name Node as meta-data, and each meta-data corresponds to a block occupies about 200 Byte. Taking a replication factor of 3(default), it would take approximately 600 Byte. If there are such huge no of these kind of smaller files in the HDFS, Name Node will consume lot of space. Name Node keeps all the meta-data in its main memory, which leads to a challenging problem for the researchers [23].

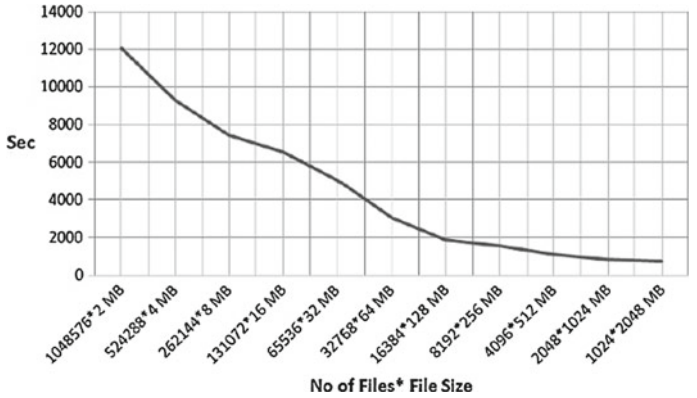

Fig. 3.15 Variation of read operation for 2 TB data size

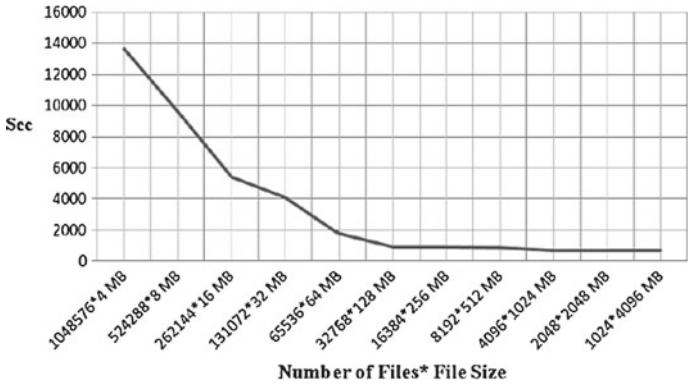

Fig. 3.16 Variation of read operation for 4 TB data size

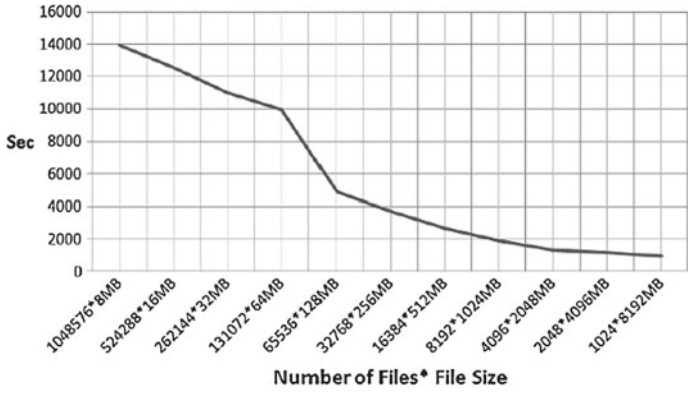

Fig. 3.17 Variation of read operation for 8 TB data size

Table 3.5 Experimental results of read operation for 1 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	1	11962.46
2	524288	2	7030.8
3	262144	4	4149.977
4	131072	8	2211.445
5	65536	16	1752.17
6	32768	32	1573.021
7	16384	64	759.939
8	8192	128	766.876
9	4096	256	611.896
10	2048	512	310.796
11	1024	1024	437.017

Table 3.6 Experimental results of read operation for 2 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	2	12062.4
2	524288	4	9288.534
3	262144	8	7452.232
4	131072	16	6535.733
5	65536	32	5041.818
6	32768	64	3048.021
7	16384	128	1864.993
8	8192	256	1542.613
9	4096	512	1099.71
10	2048	1024	812.796
11	1024	2048	747.017

3. Job Tracker at a certain time becomes extremely over loaded since it has the sole responsibility to monitor as well as dispatch simultaneously. Researchers are focusing to design a more developed version of Hadoop component for monitoring, while Job Tracker will be given the charge of overall scheduling.
4. Improving data processing performance is also a topic of major challenge for the upcoming days. A special optimization process should be assigned based on what is the actual need of application. Different experiments show that there are various scopes to increase the processing performance and thus improving time complexity of data for the execution of a particular job [3, 25].

Table 3.7 Experimental results of read operation for 4 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	4	13642.89
2	524288	8	9601.310
3	262144	16	5407.515
4	131072	32	4083.144
5	65536	64	1762.761
6	32768	128	923.675
7	16384	256	907.912
8	8192	512	830.882
9	4096	1024	711.868
10	2048	2048	702.868
11	1024	4096	718.948

Table 3.8 Experimental results of read operation for 8 TB data size

Sl no.	No. of files	File size in MB	Execution time (in secs)
1	1048576	8	13960.926
2	524288	16	12561.401
3	262144	32	10992.253
4	131072	64	9985.476
5	65536	128	4894.956
6	32768	256	3714.76
7	16384	512	2692.849
8	8192	1024	1894.974
9	4096	2048	1317.671
10	2048	4096	1181.7
11	1024	8192	989.511

3.7 Conclusion

This section of the chapter describes related 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 large-scale frameworks 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 read/write operations of files depend on its sizes and the block size of HDFS. The performance bottlenecks are not directly imputable to application code but actually depends on 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 we are focusing is the scalability of Hadoop Framework. The Hadoop cluster becomes unavailable when its NameNode is down.

Scalability issue of the Name Node has been a major struggle. The Name Node keeps all the namespace and block locations in its main memory. The main challenge with the Name Node has been that when its namespace table space becomes close the main memory of Name Node, 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.

Acknowledgments The research is supported by Data Science & Analytic Lab of NIT Silchar. The authors would also like to thank the anonymous reviewers for their valuable and constructive comments on improving the chapter.

References

1. Apache Hadoop. http://hadoop.apache.org/
2. Beaver, D., Kumar, S., Li, H. C., Sobel, J., & Vajgel, P. (2010). *Finding a needle in haystack: Facebooks photo storage*. In *OSDI, ACM* (pp. 1–8).
3. Bhandarkar, M. (2010). MapReduce programming with apache Hadoop. In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (Vol. 1, No. 1, pp. 19–23).
4. Carns, P. H., Ligon III, W. B., Ross, R. B., & Thakur, R. (2000). PVFS: A parallel file system for Linux clusters. In *Proceedings of 4th Annual Linux Showcase and Conference* (pp. 317–327).
5. Daxin, X., & Fei, L. (2011). Research on optimization techniques for Hadoop cluster performance. *Computer Knowledge and Technology*, *8*(7), 5484–5486.
6. Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings Sixth Symposium Operating System Design and Implementation (OSDI 04)* (pp. 137–150).
7. Dev D., & Patgiri, R. (in press). HAR+: Archive and metadata distribution! Why not both? In *ICCCI 2015*.
8. Dev D., & Patgiri, R. (in press). Performance evaluation of HDFS in big data management. In *ICHPCA-2014*.
9. Dev, D., & Baishnab, K. L. A. (2014). Review and research towards mobile cloud computing. In *2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (Mobile- Cloud)* (pp. 252–256).
10. Ghemawat, S., Gobio, H. & Leung, S.-T. (2003). The google file system. In *Proceedings 19th ACM Symposium Operating Systems Principles (SOSP03)* (pp. 29–43).
11. Grobauer, B., Walloschek, T., & Stocker, E. Understanding cloud computing vulnerabilities. In *IEEE International Conference on Security & Privacy* (vol. 9, pp. 50–57).
12. Guilan, X., & Shengxian, L. (2010). Research on applications based on Hadoop MapReduce model. *Microcomputer & Its Applications* (*8*), 4–7.
13. Hadoop Distributed File System Rebalancing Blocks. (2012). http://developer.yahoo.com/hadoop/tutorial/module2.html#rebalancing.
14. HDFS Federation. (2012). http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/Federation.html.
15. Lustre File System. http://www.lustre.org.
16. McKusick, K., & Quinlan, S. G. F. S. (2010). Evolution on Fast-Forward. *Communication of the ACM*, *53*(3), 42–49.

17. Nicolae, B., Moise, D., Antoniu, G., Boug, L., & Dorier, M. (2010). BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications. In *Proceedongs 24th IEEE Interational Parallel and Distributed Processing Symposium (IPDPS 2010)*.
18. Shafer, J. A. (2010). *Storage architecture for data-intensive computing*. Ph.D. thesis, Rice University. Advisor-Rixner, Scott.
19. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Vol. 1, No. 10, pp. 3–7).
20. Tantisiriroj, W., Patil, S., & Gibson, G. (2008, October). 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.
21. Tantisiriroj, W., Patil, S., Gibson, G., Son, S. W., Lang, S. J., & Ross, R. B. On the duality of data-intensive file system design: Reconciling HDFS and PVFS. In *SC11*.
22. Ubuntu. http://releases.ubuntu.com/14.04/.
23. Weil, S. A., Pollack, K. T., Brandt, S. A., & Miller, E. L. (2004). Dynamic metadata management for petabyte-scale file systems. In *ACM/IEEE SC* (pp. 4–12).
24. White, T. (2009). Hadoop, guide, The Definitive, & Inc, O' Reilly Media. (1005). Gravenstein Highway North, Sebastopol. CA, 95472.
25. Yan, J., Yang, X., Gu, R., Yuan, C., & Huang, Y. (2012). Performance optimization for short MapReduce job execution in Hadoop. In: *2012 Second International Conference on Cloud and Green Computing (CGC)* (Vol. 688, No. 694, pp. 1–3).