HDFS+: Erasure Coding Based Hadoop Distributed File System

Fredrick RomanusIshengoma

Abstract: A simple replication-based mechanism has been used to achieve high data reliability of Hadoop Distributed File System (HDFS). However, replication based mechanisms have high degree of disk storage requirement since it makes copies of full block without consideration of storage size. Studies have shown that erasure-coding mechanism can provide more storage space when used as an alternative to replication. Also, it can increase write throughput compared to replication mechanism. To improve both space efficiency and I/O performance of the HDFS while preserving the same data reliability level, we propose HDFS+, an erasure coding based Hadoop Distributed File System. The proposed scheme writes a full block on the primary DataNode and then performs erasure coding with Vandermonde-based Reed-Solomon algorithm that divides data into m data fragments and encode them into *n*data fragments *(n>m),* which are saved in N distinct DataNodes such that the original object can be reconstructed from any m fragments. The experimental results show that our scheme can save up to 33% of storage space while outperforming the original scheme in write performance by 1.4 times. Our scheme provides the same read performance as the original scheme as long as data can be read from the primary DataNode even under single-node or double-node failure. Otherwise, the read performance of the HDFS+ decreases to some extent. However, as the number of fragments increases, we show that the performance degradation becomes negligible.

Index Terms: Erasure coding, Hadoop, HDFS, I/O performance, node failure, replication, space efficiency.

1 Introduction

The tremendous advances in networking, storage capacity and processing speed of computing devices in the last decade have given rise to new applications which involves accessing and storing thousands of gigabytes of data [1, 2]. Hadoop [3, 4] is a popular open-source implementation of MapReduce [5] framework designed to analyze large data sets. It has two parts; Hadoop Distributed File System (HDFS) [6, 7] and MapReduce. HDFS is the file system used by Hadoop to store its data. It has become popular due to its reliability, scalability, and low-cost storage capability. HDFS is designed to operate on commodity hardware components, which are prone to failure. Files are triplicated (triple replication) to guarantee high data reliability. The higher value of replication factor helps HDFS to be highly fault tolerance and to increase read bandwidth. HDFS's triplication policy enables the tolerance of two node failures at maximum on its default configuration. However, it increases the storage overhead three times. An alternative efficient way to provide the same data reliability while reducing the storage overhead is to use erasure codes [9, 11, 12, 13, 14, 15]. Erasure codes store data objects as equations thus reducing much of the storage cost. Moreover, with erasure codes, I/O performance can be improved as it reduces bandwidth overhead of redundancy. In this paper, we design and implement HDFS+ by replacing triple replication of HDFS with erasure codes, and evaluate its performance in terms of space efficiency and I/O compared to the HDFS's scheme. The rest of the paper is organized as follows: We start by providing some related work in section II. Section III is dedicated to the background of the study. The proposed scheme is presented in section IV. In section V, we provide the implementation and performance evaluation is presented in section VI. We finalize with conclusion and future work in section VII.

2 Related Work

DiskReduce [8] framework proposed by Fan et al, integrates HDFS with RAID for reducing storage overhead. In DiskReduce, the write operation follows the HDFS's scheme of triple data replication and later RAID encoding process is performed in the background. The system collects k different blocks into a RAID set and m encoding blocks are calculated, and all (k+m) blocks are saved in different DataNodes. After encoding, the number of data copies is reduced from three copies to one copy. Microsoft [10] integrated HDFS with erasure codes and introduced new power proportionality and complexity tradeoffs. The system waits to receive m blocks, before calculating and writing (n-m) parity blocks. The scheme is designed to perform erasure coding online in the data center environment. Hendricks et al. [17] introduced a Byzantine fault-tolerant protocol and shows that erasure coding based mechanism can achieve higher write throughput compared to replication-based mechanisms for a distributed storage system. Maheswaran et al [18] proposed and implemented a new set of erasure codes on Hadoop HDFS with the aim of overcoming the limitation of high repair cost of Reed-Solomon codes. The study shows a reduction of approximately 2x on the repair disk I/O and repair network traffic. However, this coding requires 14% more storage compared to Reed-Solomon codes.

3 Background

In this section, we shall present basic concepts that are essential to our work.

3.1 Hadoop Distributed File System

The HDFS architecture consists of the following components: NameNode, DataNode and Client as shown in figure 3. NameNode is a central server that controls the Hadoop cluster. It keeps data structures that map a block to a filename (Namespace) and a block to a DataNode (Inode). It also executes namespace operations of the file system like opening, closing and renaming files and directories. DataNode is a node that stores HDFS blocks and acts as a slave to the NameNode. In HDFS, a file is broken down into one or more blocks and these blocks are stored in the collection of DataNodes. DataNode acts according to commands received from the NameNode such as replicating a block or deleting over replicated block also read and write request from the Client. DataNode is required to send heartbeat and block report to the NameNode in a regular interval. NameNode uses the received report to verify the file system metadata and block map. The Client is responsible for sending write and read request to the NameNode using the Client Protocol.

IJSTR©2013
www.ijstr.org

When a red or write request is sent to the NameNode, NameNode returns DataNode address information to the Client. For the write request, Client writes the blocks orderly to the DataNodes and for the read request, the Client retrieve the data from the list of DataNodes given by the NameNode. Data transfer between the Client and the DataNode is done using Data Transfer Protocol as shown in figure 3.

3.1.1 Write Operation

The write operation for the HDFS is shown in figure 1 for the Hadoop cluster having three DataNodes with default settings. The steps are as follows: (1) A block to be written into HDSF is initially cached in a temporary local file until at least one HDFS block size is reached, and then a write request is sent to the NameNode. (2) NameNode generates blockid, find and returns a list of DataNodes for saving the data to the Client.

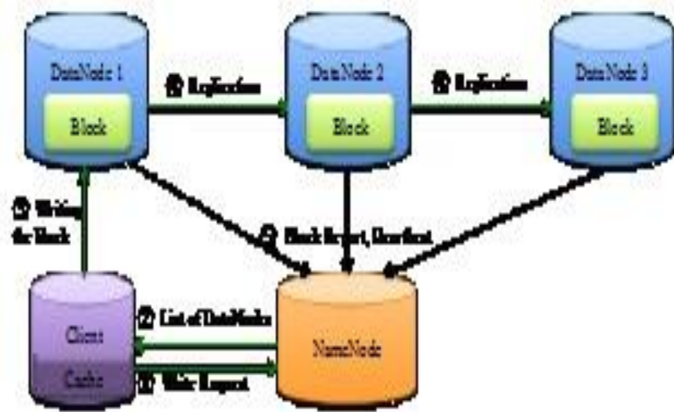

Fig. 1 HDFS write with default 3-way replication

(3) The Client writes the block to the first DataNode as shown in figure 1. (4) The first DataNode sends the block to the second DataNode in the pipeline, and the second DataNode passes it to the third DataNode. (5) In a regular time interval NameNode keep receiving a Blockreport and a Heartbeat from each of the DataNodes in the cluster.

3.1.2 Read Operation

The read operation for the HDFS is shown in figure 2 for the Hadoop cluster having three DataNodes with default settings. It has the following steps: (1) When the Client wants to read a file; it sends a read request to the NameNode. (2) NameNode locates the file, find all its blocks (using file to block mapping), find the DataNodes having these blocks, order the DataNodes according to the distance from the Client and provide a list of DataNodes to the Client as shown in figure 2. (3) The Client communicates with the nearest DataNodes to retrieve data. (4) If it cannot retrieve the data from the first DataNode then it will try from the next DataNode in the list as shown in figure 2.

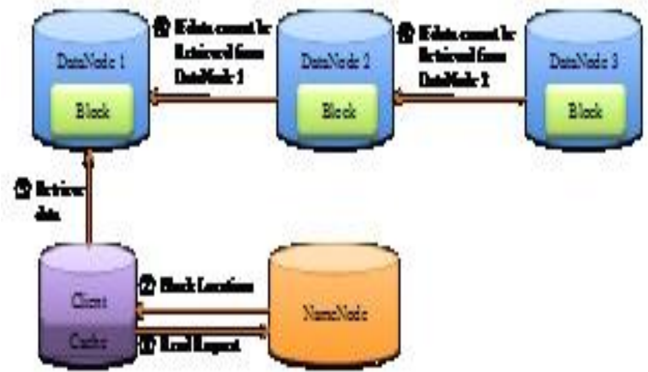

Fig. 2 HDFS read with default 3-way replication

3.2 Erasure Coding

A Maximum Distance Separable (MDS) erasure code (n, k) [11] or simply k-of-n code, takes n storage nodes, k of these holds data and m coding information, such that n can be reconstructed up to when $(n - k)$ nodes fails without data loss. The node failures are named as *erasures.* Reed-Solomon (RS) codes [16] are standard MDS codes that are obtained by evaluating polynomials over a finite field. Vandermonde-based RS code uses linear algebra in its encoding process where k data words are multiplied by the *Generator Matrix* [9] to form a *codeword*. When a node fails, the decoding process is done first by deleting rows of Generator Matrix, G_k, and then performing matrix inversion, G_k^{-1} and multiplying it to the existing words: where $\{x_0, x_1 \ldots\ x_{k-1}\}$ are the original words and $\{c'_0, c'_1 \ldots\ c'_{k-1}\}$ are the existing words after a node failure.

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{k-1} \end{bmatrix} = G_k^{-1} \begin{bmatrix} c'_0 \\ c'_1 \\ \vdots \\ c'_{k-1} \end{bmatrix}$$

We used Vandermonde-based RS codes [9] because of its powerfulness and flexibility. It has the ability to ensure data reliability for any value of n and m compared to other simple erasure code functions like XOR. For example, a (5, 2) Reed Solomon code is capable of tolerating all 3 node failures compared to (5, 2) XOR-based code which may only be able to allow at most 3 node failures.

4 Proposed Scheme (HDFS+)

Our scheme disables the HDFS replication and adds the encoding and decoding functions executed by the DataNode. Additional fragments are written as soon as they are generated and they are saved in distinct DataNodes. DataTransferProtocol is used to send and receive data between the DataNodes. An example of HDFS+ deployment is shown in figure 5.

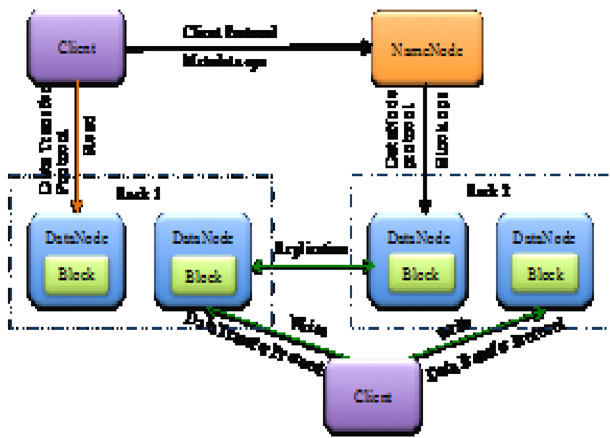

Fig. 3 HDFS Architecture

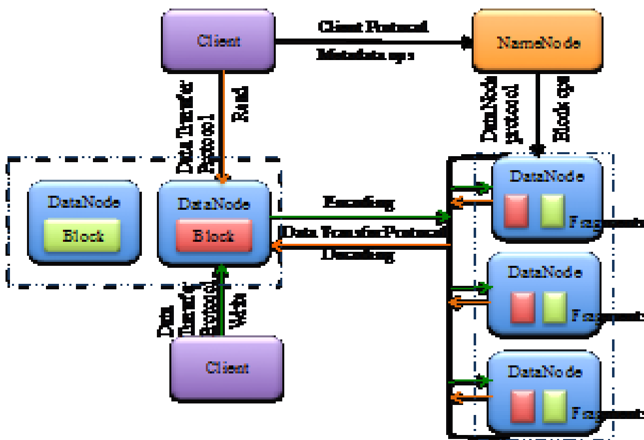

Fig. 4 HDFS+ Architecture

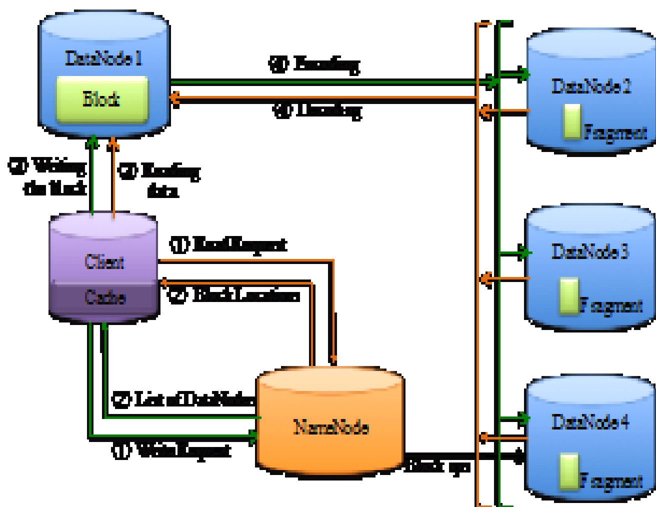

Fig. 5 Example of HDFS+deployment with 4 DataNodes

4.1 Write Operation

The write operation in HDFS+ is shown in figure 6 and it follows the following procedure: First, when the Client wants to write a file, it sends write request to the NameNode. Second, the NameNode provides to the Client the list of available DataNodes to save the data. Third, the Client chooses the nearest DataNode, and writes the blocks into it. This DataNode becomes the primary DataNode. Fourth, the primary DataNode performs erasure-encoding using configured (n, k) values that divide the block into m fragments and encode them into n fragments as shown in figure 6. The fragments created are saved on different DataNodes in the Hadoop cluster. Fifth, the acknowledgement of successful write is sent to the NameNode.

HDFS+ Write Algorithm:

Input: Data block d_b to be written
Output: Writing data block d_b to HDFS+
1: //C: Client
2: //D_N: DataNode
3://LD_N: List of DataNodes {D_{N1}, D_{N2}, ...,D_{NN}}
4://PD_N: Primary DataNode
5://N_N: NameNode
6: C sends write request to N_N
7: N_N provides LD_N to C for saving d_b
8: Cchooses one D_N from LD_N and makes it PD_N
9: C writes d_b to PD_N
10: PD_N asks N_N for LD_N for saving fragments
11:N_N provides LD_N to PD_N
12: PD_N erasure encode d_b
13: PD_N save fragments to distinct LD_N

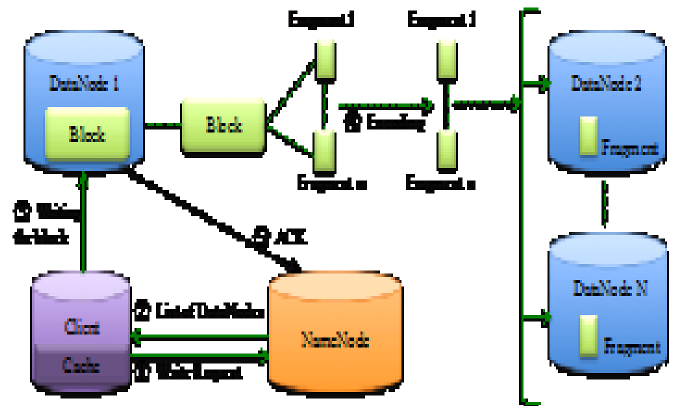

Fig. 6 HDFS+ Write Operation

By comparing figure 1 and figure 6, we could see that the proposed scheme is better than the original one in terms of space and I/O performance.

4.2 Read Operation

The Read operation of HDFS+ is shown in figure 7 and it follows the following procedure. First, when the Client wants to read a file, it sends a read request to the NameNode. Second, the NameNode provides to the Client the list of DataNodes that it can retrieve data from. Third, the Client chooses the best possible DataNode to read from, and check if it is a primary DataNode or not. Fourth, if the node selected is the primary DataNode, then the Client retrieve data. If the node selected is not the primary DataNode then it set that DataNode to be the primary DataNode. Fifth, the primary DataNode requests fragments from the other DataNodes with fragments for the requested data. When sufficient fragments have been obtained, the primary DataNode decodes the data and supply it to the read application request.

HDFS+ Read Algorithm:

Input: Data block d_b to be read
Output: Reading data block d_b to HDFS+
1: //C: Client
2: //D_N: DataNode
3://LD_N: List of DataNodes {$D_{N1}, D_{N2}, \dots, D_{NN}$}
4://BD_N: Best possible node to read from
5://PD_N: Primary DataNode
6://N_N: NameNode
7: C sends a request to read d_b to the N_N
8: N_N provides LD_N that has d_b or its fragments to C
9: Cchooses the BD_N from the given LD_N
10: if BD_N is PD_N
11: C retrieve d_b
12: else
13: set BD_N to be PD_N
14: PD_N request fragments from other nodes in LD_N
15: PD_N decodes the d_b
16: PD_N supply the d_b to C
17: end if

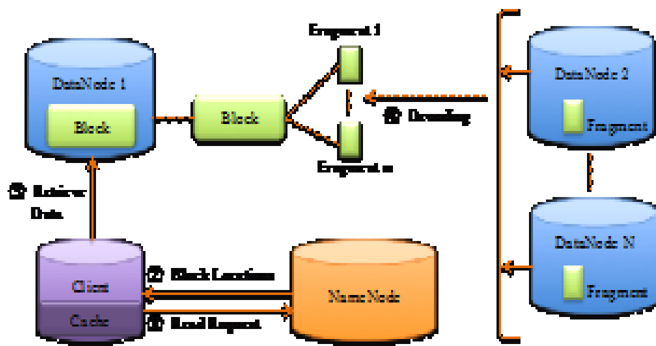

Fig. 7 HDFS+ Read Operation

By comparing figure 2 and figure 7, we could see that the proposed scheme is better than the original one in terms of space and I/O performance.

5 Implementation

Experiment setup: Our Hadoop cluster consists of 4 nodes (1 master, 4 slaves). The master node acted as a slave node in the cluster. Machines were running Linux CentOS 5.6, Hadoop stable version 0.20.203 and Java Sun's JDK 1.6.0_26. Table 1 shows the DataNodes in our Hadoop cluster with their hardware specifications.

TABLE 1
HARDWARE ENVIRONMENT FOR THE EXPERIMENT

DataNode	RAM (GB)	CPU	Hard Disk (GB)
1	15	Intel core i7-2600 3.4HGz	4000
2	1.7	Intel Core 2 Quad 240GHz	250
3	3.5	Intel Core 2 213GHz	410
4	3.9	Intel Core 2 213GHz	240

We used TestDFSIO, Hadoop I/O benchmark that is included in the source distribution for performance test. TestDFSIO takes a configured file size and the number of files, push them into HDFS where a map job is run for each file and writes the file into HDFS. We used Java Native Interface (JNI) to enable the interaction between jerasure library, which is written in C, and HDFS that is java-based. Encoding and decoding native methods are loaded and executed by the DataNode. Since HDFS recognizes the stored data in form of blocks, HDFS block was used as the unit of encoding where m data fragments are encoded to produce (n-m) additional fragments. We used Vandermonde-based RS erasure code of the jerasure library [9]. A one-to-one mapping of data and additional fragments is maintained. Additional queue to handle the additional fragments from erasure coding is added alongside with the HDFS data Queue.

6 Performance Evaluation

We tested performance of writing and reading 10 files of different file sizes on our scheme (HDFS+) and HDFS in its default settings with varying cases of zero-node failure, single-node failure and double- node failures. The Hadoop I/O benchmark tool TestDFSIO was run under loading of one map task and three reduce tasks per each node and a network performance of approximately 1 Gb/sec. HDFS was in its default configuration and HDFS+ was configured with (n, k)=(3,1) erasure code, which gives the maximum tolerance of two failures resembling HDFS reliability level in its default settings.

6.1 Space Efficiency

In figure 8 we compared the space efficiency between the original scheme (HDFS) and the proposed scheme (HDFS+).

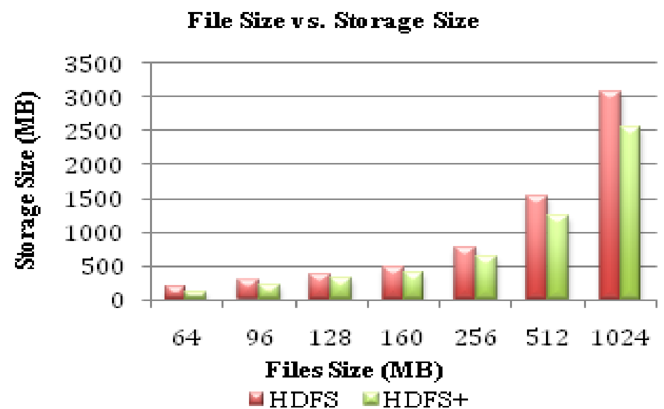

Fig. 8 Comparison between the original HDFS scheme and the proposed HDFS+ scheme on the space efficiency given the same reliability

We first observed the used and available free storage size in HDFS and HDFS+ before writing data. We then write files of different sizes using shell command "put" and observed the storage size acquired by that writing in both schemes. Storage size defines the total size of the saved file replicates for HDFS and the total size of the saved file fragments for the HDFS+. We could see that the proposed scheme (HDFS+) is better than the original scheme (HDFS) in terms of space efficiency. Our scheme increases the space efficiency by 16.67% when using *1-out-of-3* erasure coding. HDFS triplicates data, hence creating a storage overhead of *3x, x* being the size of the data file. Meanwhile, HDFS+ has storage overhead of $(x + x\left(\frac{1}{r}\right))$,

where $r = \frac{m}{n}$and $n > m$.Where *x* is the size of data file, *r* is the encoding rate, *m* is the number of fragments data is divided into; nis the number of fragments data recoded into and $\frac{1}{r}$ is the erasure coding storage overhead. Theoretically, our $(n, k) = (3,1)$ erasure code has a rate of encoding, $r = \frac{m}{n} = \frac{2}{3} = 0.66$ which gives the storage overhead of $\frac{1}{r} = 1.5$. Example, for a 256 MB file size, HDF triplicates it creating a total storage size of 768 MB. HDFS+ using $(n, k) = (3,1)$ erasure coding makes a total storage of 640 MB From the HDFS+ storage overhead equation, $(x + x\left(\frac{1}{r}\right))$, we can see that the space efficiency increases as the rate of encoding approaches to 1. The maximum space efficiency that HDFS+ can provide is 33%. Figure 9 shows the relationship between the rate of encoding and the space efficiency in HDF+.

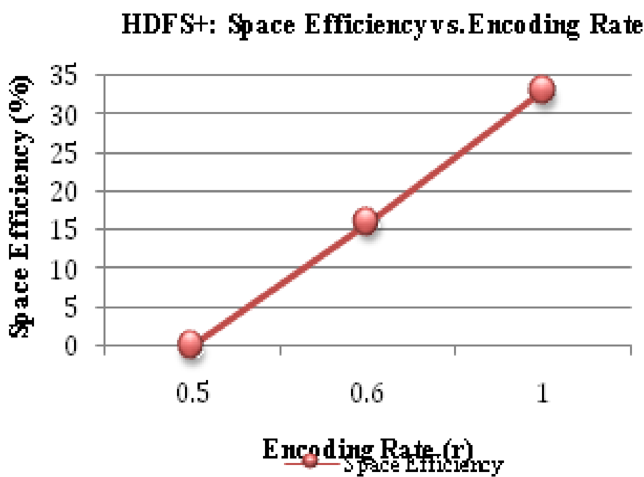

Fig. 9 HDFS and HDFS+ reading performance of 128MB file from 4 DataNodes during zero-node failure, single-node failure and double-node failures

6.2 I/O Performance

We then examined the write and read performances between the original scheme (HDFS) and the proposed scheme (HDFS+) using 10 files of 256 MB. We compared the write performances on HDFS and HDFS+ schemes with the following cases: HDFS with single replication, HDFS with double replication, HDFS with triple replication (the default HDFS setting) and HDFS+ with configured 1-out-of-3 erasure coding. In Figure 10 (a) we could see HDFS+ write performance outperformed the HDFS scheme by 1.4. The HDFS scheme pipeline DataNodes during writing, that is data is considered written after all the data blocks are replicated three times and saved to the DataNodes. HDFS+ needs to send one full block and n/m chunks to other DataNodes, compared to original scheme that sends three full blocks to the DataNodes.

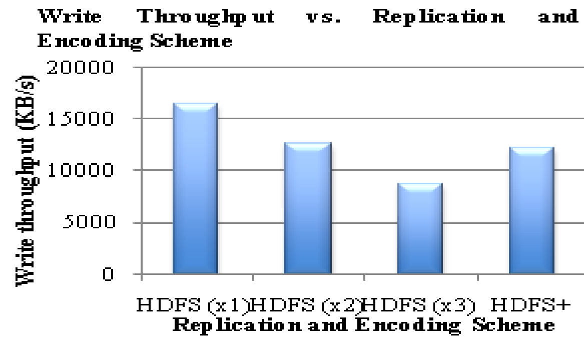

Fig. 10 (a) Comparison between the original scheme (HDFS) and the proposed scheme (HDFS+) using different replication and encoding schemes based on write performance.

In Figure 10 (a) we could see HDFS+ write performance outperformed the HDFS scheme by 1.4. The HDFS scheme pipeline DataNodes during writing, that is data is considered written after all the data blocks are replicated three times and saved to the DataNodes. HDFS+ needs to send one full block and n/m chunks to other DataNodes, compared to original scheme that sends three full blocks to the DataNodes.

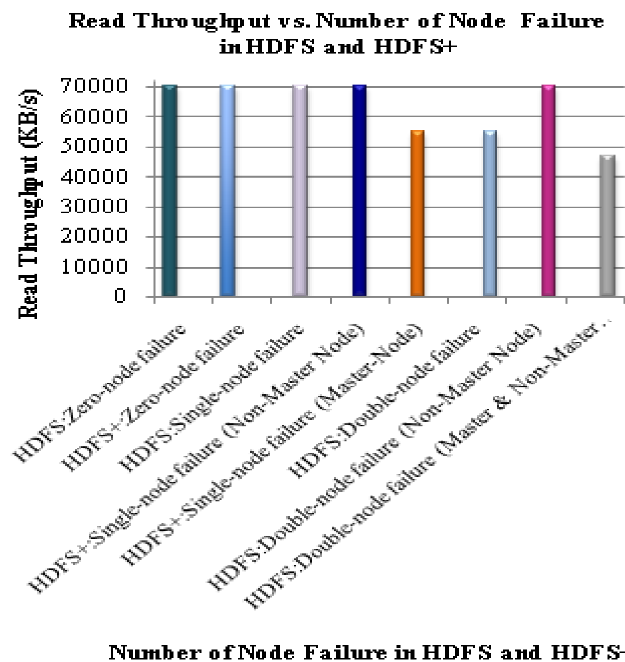

Fig. 10 (b) Comparison between original HDFS scheme and the proposed HDFS+ scheme based on the read performance under zero-node failure, single-node failure and double-node failures.

HDFS+ offer smaller sizes of encoded files, chunks that save enough network transmission time of offset the extra encoding time. We can see that there is an incremental rise in time used to write x1 to x2 and x3 in HDFS. Write throughputs are observed to be 16.44, 12.65 and 8.72 MB/s respectively for HDFS x1, x2 and x3 and 12.28 MB/s for HDFS+ with 1-out-of-3 erasure coding. HDFS+ increase write throughput since it writes less data to the network compared to the HDFS. Figure 10 (b) presents the read performance of HDFS+ compared to

the HDFS in zero-node failure, single-node failure and double-node failure. The HDFS+ read performance is observed to be approximately the same with HDFS in zero-node failure, single-node failure and double-node failure when data is read from Primary DataNode. This is because; in such scenario the read operation retrieve the block directly from the DataNode without performing the decoding process. HDFS+ decreases read performance in a single-node failure of master/primary DataNode by 22.5 %. This is due to decoding process (reconstructing data from the fragments) during read. In a double-node failure of a master node and a non-master node, HDFS+ decreases read performance by 34.2%. Otherwise, HDFS read throughput are observed to be approximately the same for zero-node failure, single-node failure and double-node failure. This is because HDFS retrieve blocks directly from the DataNode. We then computed the probability of all failure cases, having 4 DataNodes in HDFS+ and summarizes in Table 2. The probability of a single-node failure (master node) is 0.25, while the probability that other nodes will not fail is 0.75. The probability of double-node failure that involve a master-node and a non-master node is 0.5 and the probability of a double-node failure that involves two non-master nodes is 0.5.

TABLE 2

PROBABILITY OF NODE FAILURE IN HDFS+ WITH 4 DATANODES

Failure Type	Node(s) Failure Case	Probability
Single Node	Master Node	0.25
Single Node	Non-Master Node	0.75
Double Node	Master-Node & Non-Master Node	0.5
Double Node	Non-Master Node & Non-Master Node	0.5

We used the cases and their probabilities in Table 2 to compute the expected read performance in the proposed HDFS+ scheme with 4 DataNodes. Figure 11 shows the performance of expected read throughput with the number of failure. The expected read performance for HDFS remain the same while in HDFS+ the performance decreases slightly with the number of node failure.

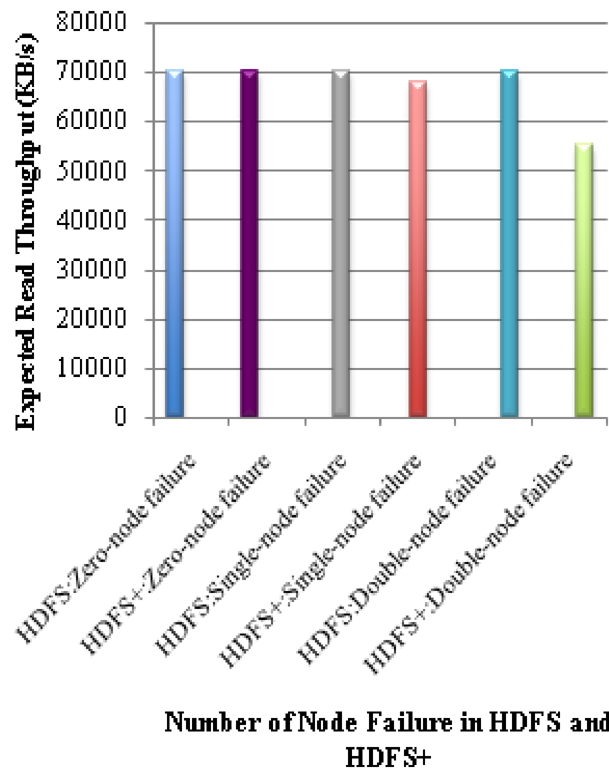

Fig. 11 Comparison between the original scheme (HDFS) and the proposed scheme (HDFS+) based on the expected read performance and number of node failures.

We then present the general case when n DataNodes are used. Table 3 shows the general case for each failure with n DataNodes. The cases cover the single-node failure of a master node, single-node failure of a non-master node, double-node failure of a master and a non-master nodes and double-node failure of two non-master nodes.

TABLE 3

GENERAL CASES FOR PROBABILITY OF NODE FAILURE IN HDFS+ WITH n DATANONES.

Event	Probability
Single node failure (Master Node)	$\frac{1}{n}$
Single node failure (Non-Master Node)	1- $\frac{1}{n}$
Double node failure (Master-Node & Non-Master Node)	$\frac{({}^{n-1}C_1)}{({}^{n}C_2)}$
Double node failure (Non-Master Node & Non-Master Node)	$\frac{({}^{n-1}C_2)}{({}^{n}C_2)}$

Based of Table 3 above, we plot the relationship that exists between the expected read throughput and the number of DataNodes in a single-node and double-node failures. The expected read throughput in a single-node failure for n DataNodes is given by the product of the average read

throughput and its probability based on the number of DataNodes.

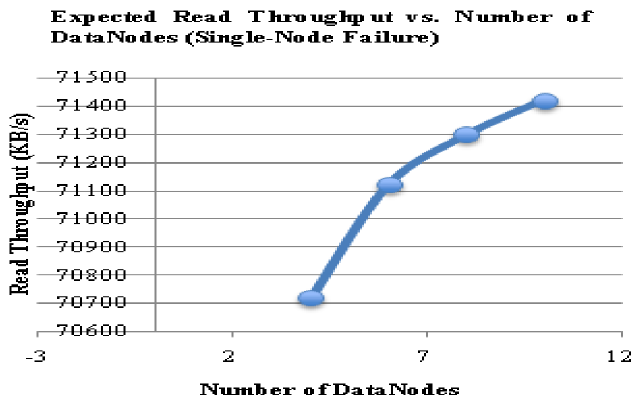

Fig. 12 Relationship between single-node failure and number of nodes in HDFS+.

Figure 13 plots the expected read throughput of a double-node failure in n DataNodes. We could see that the expected read throughput increases as the number of DataNodes increases for single-node and double-node failures. That means, as the number of DataNodes increases, the chances of primary DataNode to fail decreases. Also, there is a more chance to read from the primary DataNode or the best possible DataNode with high performance.

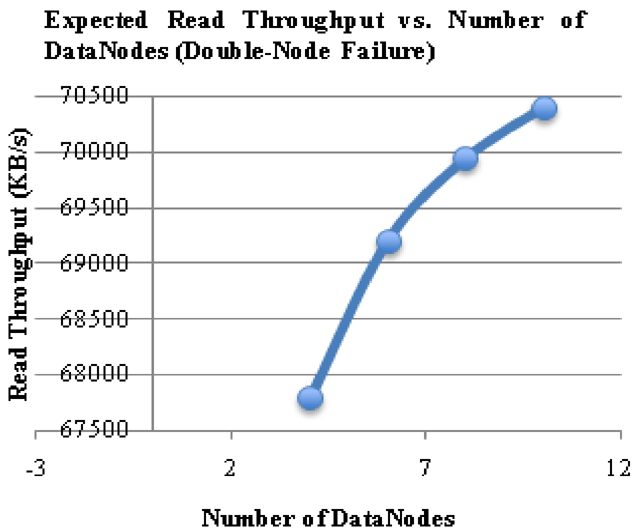

Fig. 13 Relationship between double-node failure and number of nodes in HDFS+.

7 Conclusion and Future Work

In this paper, we designed and implemented HDFS+, an erasure coding based Hadoop Distributed File System. We compared the performance of the proposed scheme with the HDFS. The experimental results show that the proposed scheme (HDFS+) can save up to 33 % of space while outperforming the original scheme in write throughput by 1.4 times. It maintains the read performance under zero-node failure, single-node failure and double-node failure provided that the data is read from the primary DataNode. The read performance of the proposed scheme (HDFS+) is decreased by 22.54 % in the presence of single node failure and 34.23% for double node failure that includes a primary DataNode. Our future work will focus on enhancing the read performance of the HDFS+ when data has to be reconstructed.

References

[1] G. D.H.-C.Du, “Recent Advancements and Future Challenges of Storage Systems”, Proceedings of the IEEE, 96(11): 1875-186, 2008.

[2] J.GantzandD.Reinsel, ”The Digital Universe Decade–Are You Ready?” http://idcdocserv.com/925, 2010.

[3] The Apache Software Foundation,“Welcome to Apache Hadoop,” http://hadoop.apache.org.

[4] T.White, “Hadoop: The definitive guide (Third Edition),” O'Reilly & Associates Incorporated, 2012.

[5] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Communications of the ACM 51(1), pp. 107–113, January 2008.

[6] The Apache Software Foundation, “Welcome to Hadoop Distributed File System,” http://hadoop.apache.org/hdfs.

[7] Konstantin Shvachko, HairongKuang, Sanjay Radia, Robert Chansler, “The Hadoop Distributed File System”, Proceedings of MSST2010, May 2010.

[8] Bin Fan. WittawatTantisiriroj, Lin Xiao, Garth Gibson, “DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing,” CMU Parallel Data Laboratory Technical Report CMU-PDL-11-112, October 2011.

[9] James S. Plank, JianqiangLuo, Catherine D. Schuman, LihaoXu, Zooko Wilcox-O'Hearn, “A Performance Evaluation and Examination of Open Source Erasure Coding Libraries for Storage,” FAST 2009.

[10] Zhe Zhang, AmeyDeshpande, Xiaosong Ma, EnoThereska, and Dushyanth Narayanan, “Does Erasure Coding Have a Role to Play in My Data Center?”,Microsoft Research Technical Report MSR-TR-2010-52. May 2010.

[11] A. G. Dimakis, K. Ramchandran, Y. Wu and C. Suh, “A Survey on Network Codes for Distributed Storage”, Proceedings of the IEEE, vol. 99, no. 3, pp. 476--489, March 2011.

[12] Yasushi Saito, SvendFrolund, Alistair Veitch, Arif Merchant and Susan Spence. Fab: Building Distributed Enterprise Disk Arrays from Commodity Components”, SIGOPS Oper. Syst. Rev,. 38(5):48-58, 2004.

[13] Rodrigo Rodrigues, Barbara Liskov, “High availability in Dhts: Erasure coding vs. replication”, In

Proceedings of 4th International Workshop on Peer-to-Peer Systems, 2005.

[14] Andreas Haeberlen, Alan Mislove, Peter Druschel, "Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures", In NSDI'05: Proceedings of the 2nd conference on Synopsum on Networked Systems Design & Implementation, pages 143-158, Berckley, CA, USA, 2005. USENIX Association.

[15] John Wilkes, Rrichard Golding, Carl Staelin, Tim Sulliva, "The HpAutoraidHierarchial Storage System", ACM Trans. Comput. Syst.14 (1): 108-136, 1996.

[16] Reed, I. S., and Solomon, G. "Polynomial codes over certain finite fields". Journal of the Society for Industrial and Applied Mathematics 8 (1960), 300–304.

[17] James Hendricks, Gregory R. Ganger, Michael K. Reiter, "Low-overhead Byzantine Fault-tolerant Storage", In *Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[18] Maheswaran Sathiamoorthy, MegasthenisAsteris, DimitricPapailiopoulos, Alexandros G. Dimakis, RamkumarVadali, Scott Chen, "XORing Elephants: Novel Erasure Codes for Big Data", Proceeding PVLDB'13 Proceedings of the 39th international conference on Very Large Data Bases Pages 325-336, 2013.