aHDFS: An Erasure-Coded Data Archival System for Hadoop Clusters

Yuanqi Chen, Yi Zhou, *Student Member, IEEE*, Shubbhi Taneja, *Student Member, IEEE*, Xiao Qin, *Senior Member, IEEE*, and Jianzhong Huang

Abstract—In this paper, we propose an erasure-coded data archival system called *aHDFS* for Hadoop clusters, where $RS(k+r,k)$ codes are employed to archive data replicas in the Hadoop distributed file system or HDFS. We develop two archival strategies (i.e., *aHDFS-Grouping* and *aHDFS-Pipeline*) in *aHDFS* to speed up the data archival process. *aHDFS-Grouping* - a MapReduce-based data archiving scheme - keeps each mapper's intermediate output *Key-Value* pairs in a local key-value store. With the local store in place, *aHDFS-Grouping* merges all the intermediate *key-value* pairs with the same key into one single *key-value* pair, followed by shuffling the single *Key-Value* pair to reducers to generate final parity blocks. *aHDFS-Pipeline* forms a data archival pipeline using multiple data node in a Hadoop cluster. *aHDFS-Pipeline* delivers the merged single *key-value* pair to a subsequent node's local key-value store. Last node in the pipeline is responsible for outputting parity blocks. We implement *aHDFS* in a real-world Hadoop cluster. The experimental results show that *aHDFS-Grouping* and *aHDFS-Pipeline* speed up *Baseline*'s shuffle and reduce phases by a factor of 10 and 5, respectively. When block size is larger than 32 MB, *aHDFS* improves the performance of *HDFS-RAID* and *HDFS-EC* by approximately 31.8 and 15.7 percent, respectively.

Index Terms—Hadoop distributed file system, replica-based storage clusters, archival performance, erasure codes, parallel encoding

1 Introduction

Existing disk-based archival storage systems are inadequate for Hadoop clusters due to the ignorance of data replicas and the map-reduce programming model. To tackle this problem, we develop an erasure-coded data archival system called *aHDFS*, which archives rare accessed data in large-scale data centers to minimize storage cost. *aHDFS* exploits parallel and pipelined encoding processes to speed up data archival performance in Hadoop distributed file system (HDFS) on Hadoop clusters. In particular, *aHDFS* leverages the three-way data replicas and the map-reduce programming models in Hadoop cluster to boost the archival performance in HDFS. We show how to accelerate the encoding process in *aHDFS* by the virtue of data locality of three replicas of blocks in HDFS.

The following three factors motivate us to develop the erasure-code-based archival system for Hadoop clusters:

- a pressing need to lower storage cost,
- high cost-effectiveness of erasure-code storage and,
- the popularity of Hadoop computing platforms.

- *Y. Chen, Y. Zhou, S. Taneja, and X. Qin are with the Department of Computer Science and Software Engineering, Shelby Center for Engineering Technology, Samuel Ginn College of Engineering, Auburn University, AL 36849-5347. E-mail: {yzc0029, xqin}@auburn.edu, yzhou056@gmail.com, szt0024@tigermail.auburn.edu.*
- *J. Huang is with Wuhan National Lab. for Optoelectronics, Huazhong University of Science and Technology (HUST), Wuhan, Hubei 430074, China. E-mail: hjzh@hust.edu.cn.*

Manuscript received 13 May 2016; revised 13 Apr. 2017; accepted 30 Apr. 2017. Date of publication 19 May 2017; date of current version 11 Oct. 2017. (Corresponding author: Yuanqi Chen.)
Recommended for acceptance by G. Tan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2706686

Reducing Storage Cost. Petabytes of data are nowdays stored in large distributed storage systems [1] like the Google File System (GFS) [2], the Hadoop Distributed File System, [3], the Windows Azure Storage [4]. In 2012, two millions of search queries received per minute; by 2014, that number has more than double. Every minute, 2.5 millions pieces of content shared in Facebook, 50 thousand applications downloaded by Apple users, 200 million messages sent via Emails [5]. Such a massive amount of data demands large-scale storage systems maintained in data centers. With an increasing number of storage nodes installed, the data storage cost goes up dramatically.

A large number of nodes leads to a high possibility of failures caused by unreliable components, software glitches, machine reboots, maintenance operations and the like. To guarantee high reliability and availability in the presence of various types of failures, data redundancy are commonly used in cluster storage systems. Two widely adopted fault-tolerant solutions are replicating additional data blocks (i.e., 3X-replica redundancy) and storing additional information as parity blocks (i.e., erasure-coded storage). For example, the 3X-replica redundancy is employed in Google's GFS [2], HDFS [3], Amazon S3 [6] to achieve fault tolerance. Also, erasure-coded storage is widely used in cloud storage platforms (e.g., Windows Azure and Facebook HDFS [4], [7], [8]) and data centers [9], [10], [11].

One way to reduce storage cost is to convert a *3X-replica*-based storage system into an erasure-coded storage. It makes sense to maintain *3X replicas* for frequently accessed data. Importantly, managing non-popular data using erasure-coded schemes facilitates savings in storage capacity without adversely imposing performance penalty. A significant portion of data in data centers are considered as non-popular

1045-9219 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

data, because data have an inevitable trend of decreasing access frequencies. Evidence shows that most of data are accessed within a short duration of the data's lifetime. For example, over 90 percent of accesses in a Yahoo! M45 Hadoop cluster occur within the first day after data creation [12].

Applying Erasure-Coded Storage. Although 3X-replica redundancy (a.k.a., three-way replication) achieves high performance than erasure-coded schemes, 3X-replica redundancy inevitably leads to low storage utilization. Cost-effective erasure-coded storage systems are deployed in large data centers to achieve high reliability at low storage cost [8], [9]. Reed-Solomon code [13] is a popular family of erasure codes used in Google's ColossusFS [14], Facebook's HDFS [9], [15], and several other storage systems [16], [17].

The Reed-Solomon code (RS) has a general 1.5x storage overhead in ColossusFS [14]; the Facebook's HDFS reduces the storage overhead down to 1.4x by adopting RS(k+r,k) (e.g., (10+4,10)), the overhead of which is half less than that of 3X-replica redundancy schemes. We are motivated to devleop an economically friendly *aHDFS* to archive data replicas in Hadoop clusters using erasure codes.

Archiving Data in Hadoop. Hadoop is a MapReduce implementation for clusters, where HDFS stores data to offer high aggregate I/O bandwidth. Hadoop is a simple yet efficient parallel and distributed computing framework providing high scalability and fault tolerance [18], [19]. The popularity of Hadoop inspires us to develop data achival schemes for HDFS to maintain a low data storage cost for Hadoop clusters deployed in large-scale data centers.

Facebook's *HDFS-RAID* implements RS encoding and decoding in the format of a distributed RAID file system running above HDFS [15]. *HDFS-EC*-developed by Cloudera engineers-employs erasure coding to reduce storage overhead compared to replication while maintaining high fault tolerance [20]. Our *aHDFS* is similar to HDFS-RAID and HDFS-EC in the way that erasure coding is a promising and practical approach to saving storage cost. *aHDFS* differs from HDFS-RAID and HDFS-EC in that *aHDFS* is a data archival system that archives rarely accessed replicas in HDFS; Archiving replicated data is out the scope of the existing HDFS-RAID and HDFS-EC systems.

Determining rarely accessed replicas can be achieved by a few straightforward ways like Opass [21]. For example, the average number of accesses can be used to distinguish rarely accessed files from popular ones (see, for example, [22]). Only the files that have been accessed fewer than a specified threshold will be treated as non-popular data to be archived. In this study, we rely on a dedicated popularity-tracking module to identify rarely accessed replicas. Because such a module is orthogonal to aHDFS, we pay little attention to the impacts of the popularity tracking module on the performance of aHDFS.

Salient Features of aHDFS. We make use of 3X-replica redundancy in HDFS to boost archival performance in clusters by the virtue of the MapReduce programming model in addition to reducing network traffic. We develop *aHDFS* - a parallel data archival system - for Hadoop clusters. *aHDFS* has the following five salient features.

- *aHDFS* manages multiple Map tasks across the data nodes that are archiving their local data in parallel.
- Two archival schemes (i.e., parallel archiving and pipeline archiving) are seamlessly integrated into *aHDFS*, which switches between the two techniques based on the size and locality of archived files.
- Intermediate parity blocks generated in the Map phrase substantially lower the I/O and computing load during the data reconstruction process.
- *aHDFS* reduces network traffic among nodes during the course of data archiving, because aHDF takes the advantage of the good data locality of the 3X-replica technique.
- We develop *aHDFS* on the top of Hadoop system, allowing a system administrator to easily and quickly deploy our *aHDFS* without having to modify and rebuild HDFS in Hadoop clusters.

Contributions. The contributions of this study are:

- We propose two efficient archival techniques - parallel archiving and pipeline archiving. These two schemes make use of the 3X-replica data layout to speed up archival performance.
- We apply the MapReduce programming model to develop the *aHDFS* system, where the two data archival techniques are implemented for Hadoop clusters.
- We conduct extensive experiment to investigate the impacts of the block size, file size, the value size on the performance of *aHDFS*.

Organization. The remainder of this paper is organized as follows. Section 2 introduces the background of this study. The basic idea of our data archival system is discussed in Section 3. Section 4 presents design and implementation of *aHDFS*. Then, in Section 5, we present the preliminary results from a baseline archival strategy. We evaluate *aHDFS* performance in Section 6. Section 7 summarizes the related work. We conclude our study with future research directions in Section 8.

2 Background

2.1 Erasure-Coded Data Storage

When data becomes corrupted in storage systems, erasure-coded data can be reconstructed by using data stored in a disk array or distributed storage systems. Erasure coding has two salient features, namely, improving fault-tolerance performance and minimizing the storage-capacity overhead in date centers. For instance, Huang et al. adopted an erasure-coded or RS codes to build an erasure-coded storage cluster [23] to improve both data storage reliability and efficiency.

$RS(k+r,k)$ encodes source data with *RS-generated-coding matrix* (see the $k+r$ by k matrix on the left-hand side of the equation in Fig. 1). It is worth mentioning that the RS-generated-coding matrix contains two matrices:

- *Identical Matrix.* This is a $k \times k$ matrix, where all the diagonal elements are set to 1 and the other elements are set to 0 (e.g., the k by k sub-matrix on the top portion of the left-hand side of the equation in Fig. 1).
- *Redundancy Matrix.* This is a $k \times r$ matrix generating parity blocks (e.g., the r by k matrix on the bottom part on the left-hand side of the equation in Fig. 1).

With a simple linear algebra calculation, the parity data blocks (see the bottom part on the right-hand side of the

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

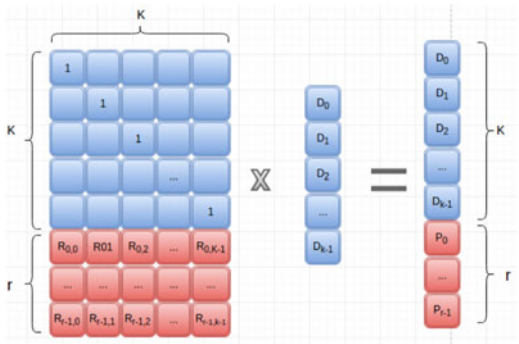


Fig. 1. With a simple linear algebra calculation, the parity data blocks (see the bottom part on the right-hand side of the equation are derived from the $k \times r$ *redundancy matrix* (see the bottom part on the left-hand side of the equation).

equation in Fig. 1) are derived from the $k \times r$ *redundancy matrix*. In a data archival scenario, we simply apply the redundancy matrix to obtain calculate r parity blocks, which can be employed to reconstruct blocks for data failures (see also Section 2.2 for detailed discussions). A $RS(k+r, r)$ code scheme can sustain up to r numbers of concurrent blocks failures.

2.2 Data Reconstruction with Reed-Solomon Coding

Now we elaborate the idea of data reconstruction using the Reed-Solomon coding. $RS(k+r, k)$ code can facilitate data recovery from a failed storage systems with up to r concurrent corrupted data blocks. Fig. 2 outlines a data reconstruction process, where RS coding is deployed to reconstructs source data blocks from surviving blocks and reconstruction matrix, which are defined as follows. We also introduce the definition of an inverse matrix used in the data reconstruction process.

- *Surviving Blocks.* This is archived data blocks which are not corrupted (see, for example, $k \times 1$ matrix on right-most-hand side of the equation in Fig. 2).
- *Reconstruction Matrix* T. This is a $k \times k$ matrix which are derived from *RS-generated-coding matrix* by getting rid of corresponding rows of corrupted blocks in *Surviving Blocks* (see, for example, the k by k matrix at the right middle of the left-hand side equation in Fig. 2).
- *Inverse Matrix* T^{-1}. This $k \times k$ matrix is an inverse matrix of *Reconstruction Matrix* T (see for example, the left-most matrix on left-hand side of equation and right-hand side of equation in Fig. 2).

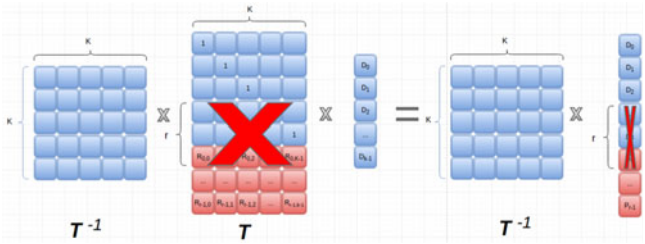


Fig. 2. Data reconstruction using the Reed-Solomon coding scheme $RS(k+r, k)$. The failed source data (see $D_0, \ldots, D_{k-1}$ on the left-hand side of the equation) can be reconstructed by multiplying inverse matrix T^{-1} with surviving blocks (see the two items on the right-hand side of the equation.

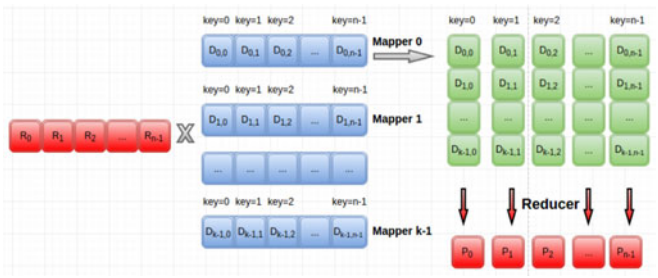


Fig. 3. The basic idea of *aHDFS*: The key-value design in the context of erasure-coded data archival systems.

Fig. 2 shows that the corrupted source data (see $D_0, \ldots, D_{k-1}$ on the left-hand side of the equation in Fig. 2) can be reconstructed by multiplying inverse matrix T^{-1} with surviving blocks (see the two items on the right-hand side of the equation in Fig. 2).

3 The Basic Idea of aHDFS

Before presenting the design of *aHDFS* (see Section 4), let us shed some light on the basic idea of our data archival system using the MapReduce programming model. Fig. 3 illustrates the key idea behind *aHDFS* leveraging the *MapReduce* programming model to generate parity data blocks with the *Key-Value* pair design. We refer to the vector $(R_0, \ldots, R_{n-1})$ on the left-hand side in Fig. 3 as a *redundancy vector*, which is one row from the *redundancy matrix* in Fig. 1. For simplicity, we only show one *redundancy vector* in Fig. 3; the redundancy vector can be extended to a *Redundancy Matrix* in our *aHDFS* design.

Data to be archived are considered as source data of mappers (see the k by n source data matrix on the right-hand side of Fig. 3). In the example presented in Fig. 3, there are a total of k source data blocks to be archived; each data block is represented as a row in the source data matrix. The top portion on the right-hand side of Fig. 3 is the intermediate output data from the mappers; the bottom part (i.e., $(P_0, \ldots P_{n-1})$ on the right-hand side of Fig. 3 is the parity data blocks produced by reducers.

In the map phase, the number of spawned mappers depends on the number of *data blocks*. For example, k mappers are originated to be in charge of archiving k data blocks. In our system, we equally divide each block to n sub-blocks represented as $D_{i,0}$ $D_{i,1}$, . . . , $D_{i,n-1}$. We assign one key to each sub-block; thus, the keys sequentially assigned to the n sub-blocks are $0, 1, \ldots,$ and $n-1$. The value of each key is the data of each sub-block. One may tune sub-block size in order to optimize mapper performance (see also Section 5.4). By multiplying the sub-blocks with the *redundancy matrix*, the intermediate sub-blocks outputted from the multiple mappers are grouped by keys and shuffled to the reducers for further processing.

In the reduce phase, sub-blocks with the same key are grouped and calculated to derive one parity sub-block (e.g., $P_i, 0 \leq i \leq n-1$). Thus, all the intermediate sub-blocks with keys 0, 1, . . . , and n are reduced to parity sub-blocks P_0, P_1, . . . , P_{n-1}, respectively.

It is worth noting that our *aHDFS* is distinctive from HDFS-RAID [15] and HDFS-EC [20] in that *aHDFS* is centered around how to archive rarely accessed replicas in

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

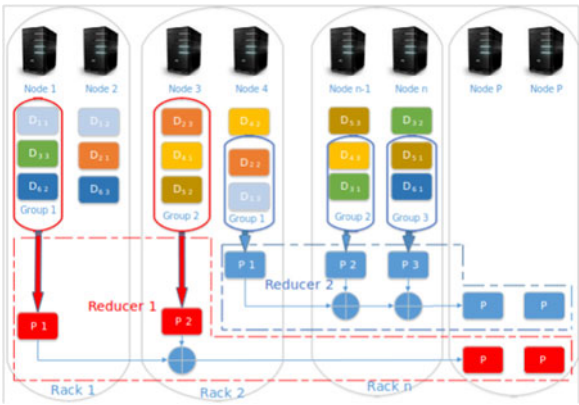


Fig. 4. Applying the MapReduce programming model to design our parallel data archiving scheme for the $RS(6+2,6)$-coded storage.

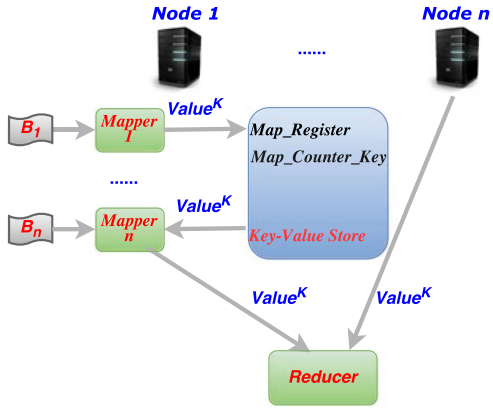


Fig. 5. The MapReduce-based parallel archiving scheme.

HDFS, whereas the other two existing systems are focused on RS encoding and decoding in HDFS.

4 The Design of aHDFS

In this section, we propose two Hadoop-based data archival strategies, namely, the *aHDFS-Grouping* and *aHDFS-Pipeline* schemes. We decide to use the HDFS default settings. Thus, in *aHDFS*, the number of the replicas of each block is three. Among the three replicas, two replicas are residing in two different nodes located in the same rack; the third replica is kept in another rack.

4.1 aHDFS-Grouping: Parallel Data Archiving

We start this section by illustrating how the *parallel data archiving* scheme works from the perspective of the *MapReduce* programming model. Then, we incorporate the key idea of the *Key-Value* pair design into parallel data archival.

4.1.1 Overview of Parallel Archiving

Let us make use of a concrete example (see Fig. 4) to demonstrate the idea of applying the MapReduce programming model to design our parallel data archival system. For simplicity, we consider a small file containing six (6) blocks denoted as $D_{i,j}$, where $i \in \{1,6\}$ and $j \in \{1,3\}$. Here $D_{i,j}$ represents the jth replica of the ith block in the file; for example, $D_{4,2}$ signifies the second replica of the 4th block. Recall that two out of the three replicas of each block are residing in the same rack, whereas the third one is stored in another rack. For instance, block 1's two replicas (i.e., $D_{1,1}$ and $D_{1,2}$) are residing in rack 1; block 1's third replica $D_{1,3}$ is hosted in a second rack (i.e., rack 2). Fig. 4 reveals two representative run-time cases of the parallel data archiving in *aHDFS*. Let us elaborate the two cases below.

- *Case 1.* There are three map tasks spawned on nodes 1 and 3, respectively. More specifically, the three mappers are concurrently processing $D_{1,1}$, $D_{2,3}$, and $D_{6,2}$ on node 1; whereas the other three mappers are handling $D_{2,3}$, $D_{4,1}$, and $D_{5,2}$), respectively. The intermediate results from the six mapper tasks running on nodes 1 and 3 are grouped in form of a single block referred to as an intermediate parity block (see, for example, $P1$ and $P2$ in node 1 and node 3, respectively).
- *Case 2.* There are two mappers spawned on each of nodes 4, $n-1$, and n. Thus, node 4 runs two mappers to process blocks $D_{2,2}$ and $D_{1,3}$; node $n-1$ manages two mappers to deal with blocks $D_{4,3}$ and $D_{3,1}$; node n governs two mappers to treat blocks $D_{5,1}$ and $D_{6,1}$. The intermediate output from each group of the two map tasks on the same node is merged as one block also referred to as an intermediate parity block (see, for example, $P1$, $P2$, and $P3$ on node 4, $n-1$, and n, respectively). The intermediate parity blocks are delivered to reducers to generate final parity blocks P and P' in node P and P', respectively.

In a real-world scenario, mapper tasks may be distributed among the nodes in an unbalanced fashion. Nevertheless, the key idea illustrated in the aforementioned two cases remains unchanged. In summary, the idea behind our the parallel archiving scheme is to group intermediate results yielded from map tasks to form one intermediate parity block to be delivered to a reducer for further computing.

4.1.2 MapReduce-Based Parallel Archiving

Now we design a parallel archiving scheme using the MapReduce programming model. Fig. 5 illustrates our MapReduce-based parallel archiving scheme, detailing the activities of several mappers and one reducer in node 1. The behaviors of the other nodes are identical as that of node 1.

We delineate the MapReduce-based parallel archiving algorithm in Section 4.1.5. To optimize the performance of our parallel archiving algorithm, we propose a new grouping strategy (see Section 4.1.3) to reduce the number of intermediate key-value pairs. We also design a local key-value store (see Section 4.1.4) to minimize the disk I/O load imposed in the map and reduce phases. The analysis of network traffic and disk I/Os can be found in Section 4.1.6.

4.1.3 Grouping Strategy

Fig. 5 illustrates our MapReduce-based strategy of grouping intermediate output from the multiple mappers (i.e., mappers $1-n$). A conventional wisdom is to deliver an intermediate result created by each mapper to a reducer through the shuffling phase. To optimize the performance of our parallel archiving scheme, we group multiple intermediate results sharing the same key into one *Key-Value* pair to be transferred to the reducer. During the course of grouping,

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

of course, the XOR operations are performed to generate the value in the *Key-Value* pair. The advantage of our new group strategy makes it possible to shift the grouping and computing load traditionally handled by reducers to mappers. In doing so, we alleviate the potential performance bottleneck problem incurred in the reducer.

4.1.4 Local Key-Value Store

At the heart of our MapReduce-based parallel archiving scheme is a *local Key-Value store* managed in each node (see the large block sitting in the center of Fig. 5). Fig. 5 shows that a local key-value store manipulated by node 1 enables the above grouping strategy to merge multiple intermediate results sharing the same key into one *Key-Value* pair.

Let us first introduce the notation used in Fig. 5, followed by the functionality of the local key-value stores. The arrows in Fig. 5 represent data flows among mappers, reducers, and the key-value data store. It is worth noting that the transferred data are in the form of $Value^k$s, where k is a key. The local key-value store residing in node 1 is responsible for temporally storing intermediate *Key-Value* pairs created by the mappers. n number of data servers, distributed to nodes $1-n$ nodes, are booted up ahead of spawned mappers. Each local key-value store, which processes local data, independently functions without collaborating with the other key-value stores on remote nodes.

The metadata of a local key-value store includes two counters, namely, the *Map_Register* and *Map_Counter_Key* counters (see also the local key-value store in Fig. 5). The first counter (i.e., *Map_Register*) is used to keep track of the total number of spawned mapper tasks on its local node. The second counter (i.e., *Map_Counter_Key*) monitors the number of mappers that have already stored their intermediate *Key-Value* pairs with respect to specific keys to the local key-value store.

The local key-value store is superior to Hadoop combiners. Although a Hadoop combiner groups map output records with the same key to reduce to number of key-value pairs delivered over the network to reducers, each combiner hooks up with only a single mapper, thereby suppressing the contribution of reducing network traffic. When multiple mappers are residing on one node, *aHDFS* employs the local Key-Value store to form a superior combiner summarizing key-value records of multiple mappers within one local node.

4.1.5 The Parallel Archiving Algorithm

Now we present the algorithm of the MapReduce-based parallel archiving scheme. Algorithm 1 illustrates the detailed process of parallel archiving, where the Map-Register and Map-Counter-Key values keep track of the total number of spawned mappers and the number of mappers that have stored intermediate results, respectively. Recall that Map-Register and Map-Counter-Key can be found in the local key-value store.

When a mapper delivers key *K*'s value to the local key-value store, the algorithm checks whether the local store buffers a copy of key *K*'s value (see Line 1 in Algorithm 1). If the value of key *K* is not residing in the local store, the algorithm places key *K* and its value into buffer *Val_Buffer*, followed by increasing the counter of key *Key* by 1 (see Lines 1-3 in Algorithm 1 and mapper 1 in Fig. 5).

Algorithm 1. The Parallel Archiving Algorithm

Input: *Map-Register*, *Map-Counter-Key*, *Val-Buffer*
Map-Register: spawned Mapper tasks on each node
Map-Counter-Key: the number of mapper tasks, which have already stored the value of a key into data server
Val-Buffer: store the value of *Key-Value* pairs;

1: **if** *Val-Buffer*.get(KEY_k) == $null$ **then**
2: *Val-Buffer*.put(KEY_k, VALUE)
3: *Map-Counter-KEY*$_k$++
4: **else**
5: Return *Val-Buffer*.get(KEY_k) to Mapper
6: Mapper executes exclusive or calculation to returned value and value of KEY_k emitted
7: *Map-Counter-KEY*$_k$++
8: **if** *Map-Counter-KEY*$_k$==*Map-Register* **then**
9: write calculated result to reducer
10: *Val-Buffer* remove Key_k
11: **else**
12: store calculated result to data server
13: *Map-Counter-KEY*$_k$++
14: **end**
15: **end**

If key *K*'s is found in the local key-value store, then the stored value will be forwarded back to the mapper, which will in turn be XORed with the same key's value emitted by the mapper (see Lines 5-6 in Algorithm 1 and Mapper 2 in Fig. 5).

After key *K*'s counter is increased by 1, the algorithm checks if the mapper is the last one storing key *K*'s value to the local key-value store. The algorithm achieves this goal by comparing *Map_Counter_Key* with *Map_ Register*. If the mapper is the last one, this mapper writes the key-value pair to a reducer rather than storing this key-value pair in the local store (see Lines 9-10 in Algorithm 1 and Mapper n in Fig. 5).

Unlike the last mapper, non-last mappers buffer the key-value pairs to the local store while increasing the counters of their key by 1 (see Lines 12-13 in Algorithm 1 and Mapper 2 in Fig. 5).

4.1.6 Reducing Network Traffic and I/O Load

Intuitively, we show that only one copy of each key *K*'s value is shuffled to a reducer. As a result, regardless of the number of spawned mappers on each node, only one intermediate key-value pair is transferred on the network during the shuffling phase. Suppose there are m mappers on a node, our algorithm is capable of reducing the shuffling network load by $\frac{m-1}{m} \times 100\%$. For example, given a node running eight mappers, our algorithm reduces the network traffic by 87.5 percent.

Similarly, our algorithm significantly lowers the I/O load, because the total amount of data written to a local disk is reduced by $\frac{m-1}{m} \times 100\%$, where m is the number of running mappers on a node. Our algorithm is conducive to suppressing the I/O load of the map and reduce phases, because the local key-value store (see Fig. 5) buffers intermediate *Key-Value* pairs in the main memory rather than the hard drive.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

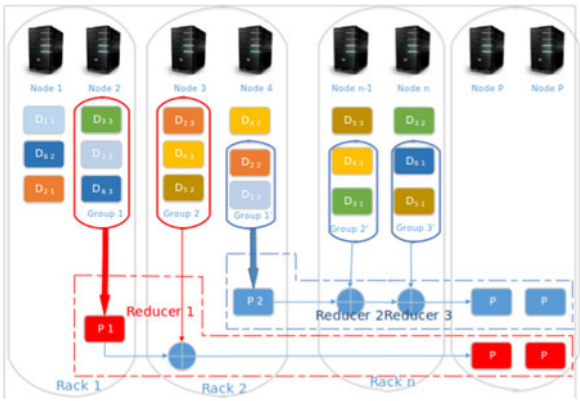


Fig. 6. Incorporating a pipeline into our parallel data archiving scheme for the $RS(6+2,6)$-coded storage. This example is independent of that in Fig. 4.

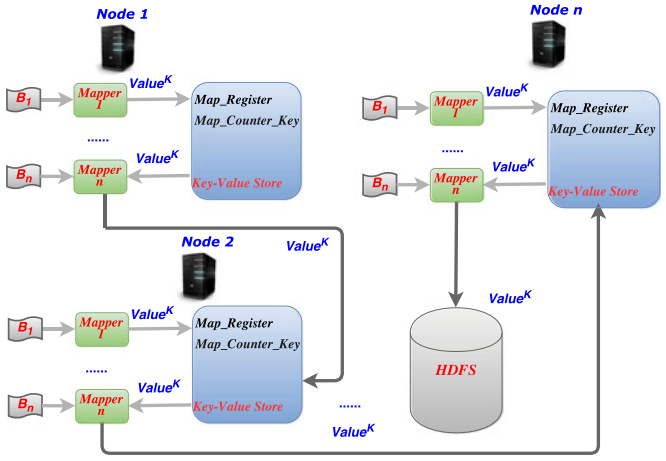


Fig. 7. The MapReduce-based parallel archiving scheme coupled with a pipeline.

4.1.7 Theoretical Analysis

The time complexity of aHDFS largely depends on the time spent in RS encoding. Let us consider HDFS where there are r' replicas. To tolerate r' concurrent failures, we archive replicated data in the format of RS(k+r, k) erasure code, where r is set to r'. Each parity block (i.e., a total of r') is the XORing result of original blocks and the redundancy matrix. Thus, the time complexity is derived from the number k of original blocks and the number r' of parity blocks as $O(k \times r')$. When aHDFS assigns k blocks to multiple mappers implement parallel data achieving, the time complexity becomes $O(k \times r/p)$, where p is the number of nodes running parallel mappers.

4.2 aHDFS-Pipeline: Data Archiving with a Pipeline

4.2.1 Overview of Parallel Archiving with a Pipeline

To boost the performance of the parallel data archiving scheme proposed in Section 4.1, we design a pipeline and incorporate the pipelining technique into our *aHDFS* system. Fig. 6 shows a way of incorporating a pipeline into our parallel data archiving scheme for the $RS(6+2,6)$-coded storage. Please note that the example illustrated in Fig. 6 is independent of that in Fig. 4.

In the parallel archiving scheme described in Section 4.1.1 (see Fig. 5), intermediate parity blocks created by mappers running on a node are delivers to reducers. Unlike this process, the pipelined data archiving scheme delivers intermediate parity blocks (see blocks $P1$ and $P2$ in Fig. 6) to subsequent nodes (for example, node 3 and node $n-1$ in Fig. 6). Next, the subsequent nodes repeatedly deliver their intermediate parity blocks to their subsequent nodes (see, for example, node n is the subsequent one of node $n-1$). Finally, the last node that has no subsequent node writes its parity blocks to the file system (i.e., HDFS).

4.2.2 Constructing Data Archiving Pipelines

According to data placement governed by the layout of HDFS, all nodes can be divided into multiple groups, each of which handles data archiving operations in a pipelined manner. For example, Fig. 6 shows that node group 1 consists of nodes 2 and 3, whereas node group 2 contains nodes 4, $n-1$, and n. In this example, *aHDFS* builds an archiving pipeline between nodes 2 and 3 in node group 1; *aHDFS* constructs another archiving pipeline among nodes 4, $n-1$, and n in node group 2. These two archiving pipelines perform data archival operations in parallel. Please refer to the two run-time cases discussed in Section 4.1.1 for details on how to form multiple node groups to conduct parallel data archiving.

Multiple data archiving pipelines are capable of simultaneously carrying out encoding processes for separate data sets, thereby delivering high data-archival performance through improved archival parallelism.

4.2.3 MapReduce-Based Pipelined Data Archiving

Our pipelined data archiving strategy is an extension of the parallel data archiving scheme proposed in Section 4.1. Fig. 7 plots that in the pipelined archiving strategy, the last mapper writes *key-value* pairs to its subsequent node's local *key-value* store (see Section 4.1.4) rather than a reducer. The last mapper in each node propagates *key-value* pairs to the node's subsequent node's key-value store. The last node in an archiving pipeline has no subsequent node, it is the last node's responsibility to write a parity block to HDFS.

Compared with the parallel archiving scheme (see also Section 4.1.5), this pipelined data archiving approach evenly distributes network I/O load among multiple nodes in each pipeline. A salient feature of our pipelined archiving scheme lies in its improvement of the parallel archiving scheme in terms of the archiving performance in the shuffling and reduce phases.

Algorithm 2 outlines the procedure of MapReduce-based pipelined data archiving. Algorithm 2) has to figure out whether a node is the first or the last node in an archiving pipeline (see Lines 8 and 10 in Algorithm 2), because the first node has no predecessor node whereas the last node has no subsequent node. The intermediate data are deliver from one node to its subsequent node (see Lines 9 and 11 in Algorithm 2). The last node in an archiving pipeline is in charge of writing a parity block to reducer (see Lines 12 and 13 in Algorithm 2). To check if all the mappers has stored intermediate *Key-Value* pairs to the local *key-value* store, all the nodes except the first node in a pipeline has to add 1 to *Register-Map*, indicating that these non-first nodes receive one more value from their predecessors (see Lines 10-13 in Algorithm 2).

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Hardware and Software Configurations of the Testbed

Hardware	
CPU	Intel Genuine 62@ 2.2 GHz, 20 CPU
Memory	128 GB
Network	1 GigaBit Ethernet network card
Disk	1 TB
Software	
Operating System	CentOS 6.5 Linux 2.6.32-431. el6.x86_64 x86_64
Hadoop	2.6.0
Replication Factor	3
Default Block Size	128 MB
Default File Size	10 GB

Algorithm 2. *aHDFS* Pipeline Archiving

Input: *Map-Register*, *Map-Counter-Key*, *Val-Buffer*
Map-Register: spawned Mapper tasks on each node
Map-Counter-Key: amount of mapper tasks which have already stored the key's value into data server
Val-Buffer: store the value of *Key-Value* pairs

1: **if** *Val-Buffer*.get(KEY_k)==null **then**
2: *Val-Buffer*.put(KEY_k, VALUE)
3: *Map-Counter-*KEY_k++
4: **else**
5: Return *Val-Buffer*.get(KEY_k) to Mapper
6: Mapper executes exclusive or calculation to returned value and value of KEY_k emitted
7: *Map-Counter-*KEY_k++
8: **if** Nodes is first node &&*Map-Counter-*KEY_k==*Map-Register* **then**
9: write calculated result to subsequential node
10: **else if** *node is not last node* && *Map-Counter-*KEY_k==*Map-Register*+1 **then**
11: write calculated result to subsequential node
12: **else if** *node is not last node* && *Map-Counter-*KEY_k==*Map-Register*+1 **then**
13: write calculated result to reducer
14: *Val-Buffer* remove Key_k
15: **else**
16: store calculated result to data server
17: $Map - Counter - KEY_k$++
18: **end**
19: **end**

4.2.4 Improving I/O Performance

Block sizes of HDFS are large; the default block size of HDFS [3] is 128 MB. A storage request unit is an access unit from the I/O path's standpoint, where each block is accessed via a sequence of storage request units. I/O accesses tend to be non-sequential when storage request units are located on different blocks; as a result, such non-sequential access patterns downgrade the write bandwidth of our *aHDFS*.

The aforementioned non-sequential I/O access patterns may occur in *aHDFS* if the following two conditions are met: (1) Two parity blocks generated by two data archiving pipelines are written to an identical node and (2) the two parity blocks are written to different disk regions. We incorporate the *write aggregation* technique into *aHDFS*. It is noteworthy that the *write aggregation* technique is a popular I/O optimization scheme, because large storage request units help in achieving high write bandwidth [24], [25]. The write aggregation technique consolidates multiple request units into a single write request. A node storing parity blocks buffers storage request units delivered from multiple data archiving pipelines; then, the buffered parity-block requests are written to the node in form of a large sequential write request.

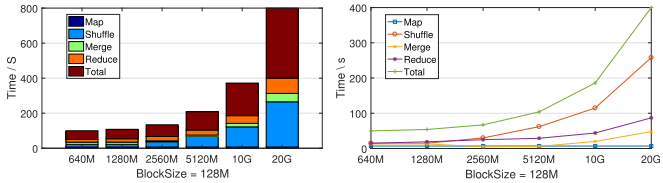


Fig. 8. The execution time of the four phases under the various file sizes.

5 Case Studies

We conduct three case studies to offer an insightful guidance on the development of the performance optimization strategies presented in Section 4.

5.1 Experimental Setup

We setup a Hadoop cluster, configuration of which is summarized in Table 1. The number of parity is set to three; and the number of nodes is configured as 10. We set the default value size, block size, and file size to 1 KB, 128 MB, and 10 GB, respectively. The replication factor of HDFS is configured to three. We measure data archival performance in terms of execution times in the map, shuffle, and reduce phases as well as the total archiving time.

We investigate the impacts of file size, block size, and *key-value* pair size on the archival performance of a baseline system, where we disable the proposed grouping strategy (see Section 4.1.3), local key-value store (see Section 4.1.4), and archiving pipeline (see Section 4.2.2).

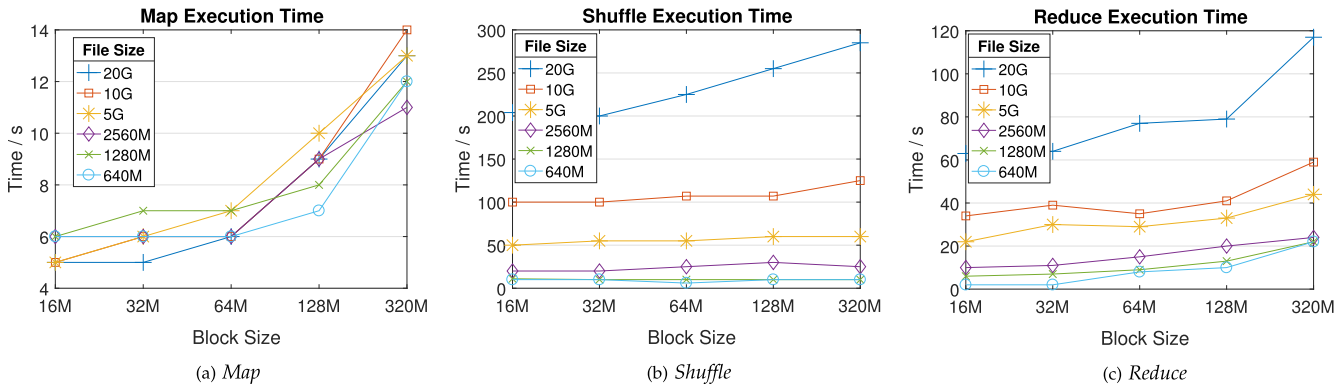
5.2 Case Study 1: File Size

We conduct six groups of experiments by varying file size, which is an affecting factor to archival performance. The file sizes chosen in the experiments are 640, 1,280, 2,560, 5,120 MB, 10, and 20 GB. We set the block size and the value size to their default values.

We aim to pinpoint data-archival-performance bottleneck among the four MapReduce phases, namely, map, shuffle, merge, and reduce. Fig. 8 illustrates the execution time of the four MapReduce phases under the various file size. Fig. 5.2 shows that with the file size increased from 640 MB to 20 GB, the total execution time significantly increases from 50 to 400 Sec. Such a trend is more pronounced when the file size is doubled from 10 GB up to 20 GB.

The results reveal that the map phase contributes a tiny portion to the total execution time (see Fig. 5.2). Additionally, the map execution time does not change noticeably when the file size dramatically increase from 640 MB to 20 GB. In contrast to the map tasks, the shuffle tasks play a major role in determining the total execution time. More specifically, the shuffling time accounts for more than half

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.



(a) *Map* (b) *Shuffle* (c) *Reduce*

Fig. 9. The execution times of the map, shuffle, and reduce phases under various block size.

of the total execution time (see Fig. 5.2). Fig. 5.2 shows that the archival-performance trend, measured in total execution time, largely depends on the shuffling execution time. Following the shuffle tasks, the reduce tasks consume approximately 25 percent of the total execution time (see Fig. 5.2).

Optimization Guidance. Because the shuffle and reduce phases account for 60 and 25 percent of the entire data archiving time, this convincing evidence suggests that shortening the time spent in the shuffle and reduce phases can greatly optimize the archival performance of *aHDFS*. An excessive amount of intermediate results from the mappers makes the shuffle phase impose a significant impact on the archival performance. The execution time of the reduce phase is also determined by the amount of data shuffled to the reducers.

We take the following two approaches to optimizing the data archival performance. First, because the load of the mappers is fairly light thanks to the high parallelism, we optimize the *aHDFS*'s performance by lowering the workload of the reducers, the computing load of which is reallocated to the mappers. Second, we minimize the amount of data shuffled to the reduce tasks. This goal can be achieved by our grouping strategy (see Section 4.1.3) and data-archiving pipelines (see Section 4.2.2).

5.3 Case Study 2: Block Size

In this section, we conduct five groups of experiments to evaluate the impacts of block size on archival performance. Again, we set the file size to 640, 1,280, 2,560, 5,120 MB, 10, and 20 GB, respectively.

Fig. 11 reveals that the execution times of the map, shuffle, and reduce phases, which are obviously affected by block size. Let us first consider the map phase (see Fig. 9a). When the block size increases from 16 to 320 MB, the execution time of the mappers doubles regardless of the file size. For example, when we set block size to 16 and 320 MB, the time spent in the map phase is 6 and 12 Sec., respectively. Interestingly, keeping the blocks size a constant, we observe that the map execution time discrepancy among various file size is within two seconds.

The trend shown in Fig. 9a is attributed to fact that each spawned mapper is in charge of one block serving as an input split. A small block size implies that the amount of data to be loaded and calculated by each map task is small. Moreover, a small block size of a fixed file size means a large number of mappers, which pushes the parallelism level up. For example, Fig. 10 shows that when we increase the number of mappers from 16 to 90, the average map execution time is correspondingly cut down from 13 to 7.

In contrast, the mapper performance is marginally improved by one second when the number of mappers is changed from 90 to 180 and from 180 to 360, respectively. Such a limited performance improvement lies in the heavy load in terms of the number of mappers on each data node of the Hadoop cluster. In our experiments, the average number of mappers assigned to each node is 1.6, 4.5, 9, 18, and 36, respectively. The nodes are lightly loaded if the average number of mappers per node is below nine. Under light workload condition, increasing the number of mappers on each node enhances the parallelism level. On the other hand, assigning too many mappers to each node can undoubtedly overload the node, thereby slowing down the speed of the mappers.

Fig. 9b shows that when the block size is increased from 16 to 320 MB, the shuffling time is enlarged. This trend becomes more pronounced when the file size is large (e.g., 20 GB). Similarly, Fig. 9c reveals that a large block size leads to a long reduce phase. Comparing Figs. 9b and 9c, we observe that reduce execution time is more sensitive to block size than shuffle execution time. For instance, in the 20 GB case, the shuffle and reduce execution times jump from 200 to 280 seconds and from 65 upto 120 seconds, respectively. In the same case, the map execution time only increases from 8 to 13 seconds.

Optimization Guidance. There are two approaches to tuning the performance of *aHDFS*. First, block size has a noticeable impact on map, shuffle, and reduce phases. We can speed up data archival performance by applying a small block size, which gives rise to high level of parallelisms. Second, to significantly shorten the reduce execution time, we make the

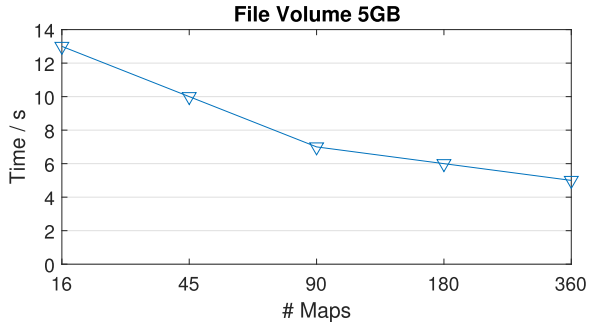
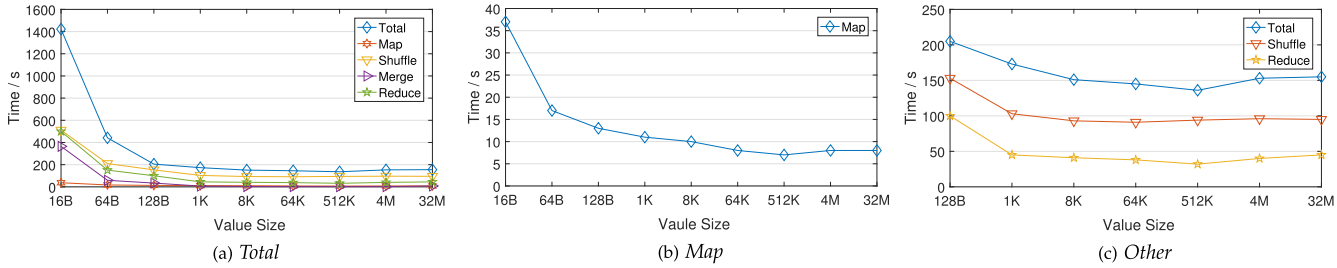


Fig. 10. The impact of the number of mappers on time spent in the map phase.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.



(a) *Total* (b) *Map* (c) *Other*

Fig. 11. The execution times of the map, shuffle, and reduce phases under various value size.

data archiving process less reduce-intensive by shifting computing load from the reducers to the mappers in *aHDFS*.

5.4 Case Study 3: Key-Value Pair Size

Now we demonstrate the impacts of *key-value* pair size on mappers in data archiving. In this group of experiments, we test nine value sizes (i.e., 16B, 64B, . . . , 32M). Fig. 11a plots the correlation between execution times and the value size.

The results show that a large value size lowers the execution times. For example, when value size is 16B, the total execution time is 1,422 seconds (see Table 2), which is approximately three times larger than the case of value size being 64B. Similarly, the shuffling time dramatically changes from 511 to 210 seconds; the merging time declines from 365 to 59 seconds; and the reducing time significantly shrinks 509 to 151 seconds. Not surprisingly, the total execution time drops from 422 to 205 seconds when the value size is doubled from 64 to 128B. Fig. 11b shows a large value size helps in reducing the mapping time. For example, increasing the value size from 16 to 64B cuts the mapping time from 37 to 17 seconds.

Figs. 11a, 11b, and 11c shows that when the value size is smaller than 128 KB, the archival performance is extremely sensitive to the value size. When the value size goes beyond 128 KB, further increasing the value size can hardly lower the archiving time. We discover that 512 KB is an optimal value size, which leads to the smallest map and reduce phases (i.e., 7 and 32 seconds in Table 2).

The aforementioned performance trends are attributed to the data overhead introduced to *key-value* pairs. Each *key-value* pair has a eight-byte key and a four-byte partition identification. When the value size is 16B, the data overhead represents 75 percent of the value size; such overhead percentage drops down to 18.8 percent, if the value size is as large as 64B. Moreover, increasing the number of *key-value* pairs inevitably imposes data overhead used to maintain the partition identifications.

TABLE 2
The Execution Times of the Map, Shuffle, Merge, and Reduce Phases as Well Ass Total Execution Time Under Various Value Size

Value Size	Total	Map	Shuffle	Merge	Reduce
16 Byte	1,422	37	511	365	500
64 Byte	442	17	210	59	151
128 Byte	205	13	153	34	100
1 KB	173	11	103	7	45
8 KB	151	10	93	3	41
64 KB	145	8	91	2	38
512 KB	136	7	94	2	32
4 MB	153	8	96	2	40
32 MB	155	8	95	8	45

Fig. 12 illustrates that a large value size leads to a small amount of data emitted from the mappers. Let us consider an example, where the input data size is 10 GB (see Table 3). When the value size is 16B, the amounts of map emitted data, output and input data are 17.5, 67.15, and 48.75 GB, respectively (see Fig. 13 and Table 3). Table 3 shows that network I/O traffic during the shuffling stage is 18.75 GB when the value size is set to 16B. Fortunately, an increasing value size helps to cut the amount of map emitted data as well as to alleviate the heavy network traffic (i.e., approximately to 10 GB, see also Fig. 12). The I/O load in terms of writes and reads is gradually lowered down to the levels of 30 and 20 GB, respectively.

Optimization Guidance. This group of experiments suggests that the benefit of a large value size is four-fold: (1) minimizing data overhead; (2) reducing the number of I/O operations; (3) lowering network traffic; and (4) conserving time spent in sorting key-value pairs. Nevertheless, an extremely large value size may lead to the heap overflow problem, thereby downgrading archival performance (see Figs. 11b and 11c).

6 Performance Evaluation

We conduct extensive experiments to compare *aHDFS* with the baseline system, the performance of which is analyzed in Section 5. We denote *aHDFS-Grouping* as *aHDFS*, where the grouping and local key-value store strategies are incorporated. We denote *aHDFS-Pipeline* as *aHDFS*, where the archiving pipeline is enabled. We also compare *aHDFS* with the existing schemes - HDFS-RAID [15]. and HDFS-EC [20].

6.1 Key-Value Pair Size

We examine the impacts of key-value pair size on the archival performance of *aHDFS-Grouping*, *aHDFS-Pipeline*, and

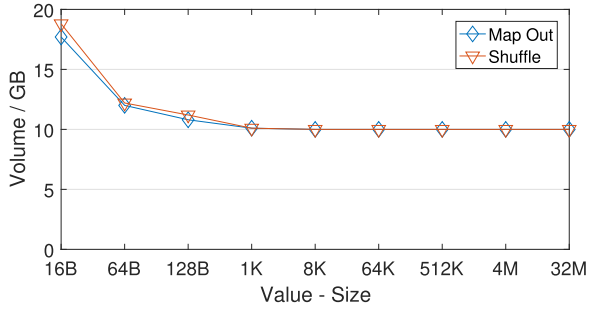


Fig. 12. The amount of data in map phase and shuffle phase, file size = 10 GB.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
I/O Write, Read, HDFS I/O Read, Mapper Output and Network Traffic I/O Under Various Vaue Size, File Size = 10 GB

Value Size	IO/W	IO/R	HDFS/R	Map/Out	Shuffling
16 Byte	48.75	67.15	10.00	17.50	18.75
64 Byte	26.51	38.70	10.00	11.88	12.19
128 Byte	24.30	35.48	10.00	10.94	11.17
1 KB	20.31	30.48	10.00	10.12	10.16
8 KB	20.04	30.07	10.00	10.01	10.02
64 KB	20.01	30.02	10.00	10.01	10.00
512 KB	20.00	30.01	10.00	10.00	10.00
4 MB	20.00	30.01	10.00	10.00	10.00
32 MB	20.00	30.01	10.00	10.00	10.00

Baseline. We test six optimal value sizes (see Case Study 3 in Section 5.4 for the optimal value sizes). The file and block sizes are set to 10 GB and 128 MB, respectively.

Fig. 14 shows execution times of the map, shuffle, reduce phases as well as the total time of the three systems under various value sizes. The shuffle and reduce times of aHDFS-Pipeline are in Figs. 14b and 14c, because observed shuffle and reduce times in aHDFS-Pipeline are extremely low as the pipeline hides the shuffle/reduce times during the data archival process. For the same reason, shuffle/reduce times of aHDFS-Pipepline are ignored in Figs. 15b and 15c. The results show that *Baseline* outperforms *aHDFS-Grouping* and *aHDFS-Pipeline* in mappers when the value size is 1 KB (see Fig. 14a). For example, the mapper execution times of *aHDFS-Grouping* and *aHDFS-Pipeline* are 65 and 85 seconds, which are approximately 6.5 and 8.5 times longer than that of *Baseline*. Interestingly, the discrepancy of the mapper execution times among the three schemes is gradually diminishing when the value size increases. For instance, if the value size is 8KB, *aHDFS-Grouping* and *aHDFS-Pipeline*'s mapper execution times are approximately 2.5 and 4 times longer than that of *Baseline*. The mapper time differences among the three systems is further narrowed down when the value size exceeds 512 KB.

The aforementioned performance trends are attributed to the fact that high computing load is reallocated from reducers to mappers in *aHDFS-Grouping* and *aHDFS-Pipeline*.

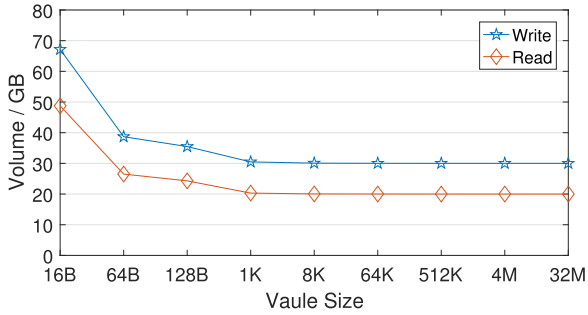


Fig. 13. The amount of input and output data during the MapReduce all phases, file size = 10 GB.

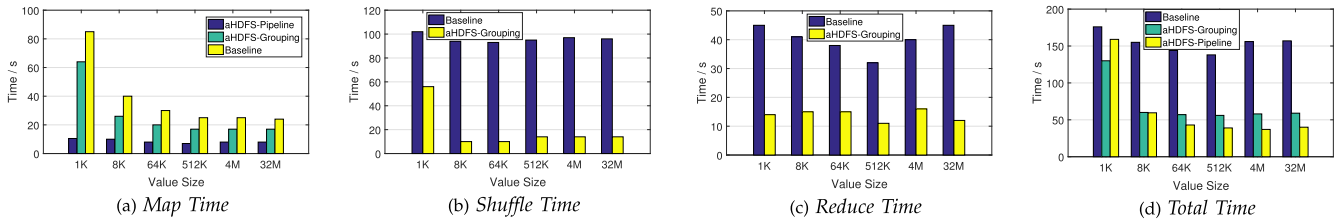
Each mapper in *aHDFS* should store temporal *key-value* pairs into the local *key-value* store; then, *aHDFS* performs the XOR operation on the emitted *key-value* pair with the previous stored *key-value* pairs in the local *key-value* store. Given fix-sized blocks, a small value size leads to a large number of *key-value* pairs, which in turn push up the number of I/O accesses to the local *key-value* store. Although the mapper execution time of *aHDFS* is longer compared to *Basedline*, *aHDFS-Grouping* is superior to *Baseline* in terms of shuffle and reduce execution times. For example, *aHDFS-Grouping* significantly shorten *Baseline*'s shuffle and reduce phase by approximate 90 and 60 percent when the value size is larger than 1 KB. (see Figs. 14b and 14c).

Fig. 14d shows that when the value size is large than 1 KB, *aHDFS-Grouping* and *aHDFS-Pipeline* can cut *Baseline*'s the overall archival time by up 60 and 75 percent, respectively.

6.2 File Size

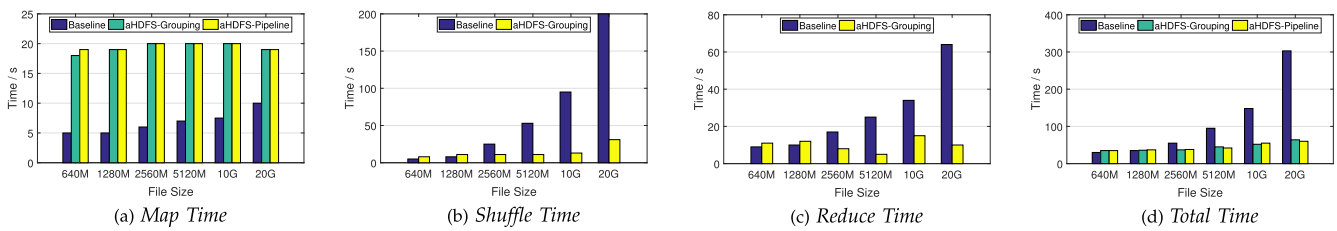
To evaluate the impacts of file size on the archival performance, we set file size to 640, 1,280, 2,560, 5,120 MB, 10, and 20 GB, respectively.

Fig. 15 illustrates the map, shuffle, and reduce times along with the total execution time. Fig. 15a shows that *Baseline*'s map time rises from 5 to 10 seconds when we increase the file size from 640 MB to 20 GB. In contrast, *aHDFS-Grouping* and *aHDFS-Pipeline*'s map times are almost kept at the level of 19 seconds. Such a downside



(a) *Map Time* (b) *Shuffle Time* (c) *Reduce Time* (d) *Total Time*

Fig. 14. The execution times of the map, shuffle, reduce phases as well as the total time of the three systems under various value sizes.



(a) *Map Time* (b) *Shuffle Time* (c) *Reduce Time* (d) *Total Time*

Fig. 15. The execution time of the map, shuffle, and reduce phases along with the total execution time of the three systems under various file size.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

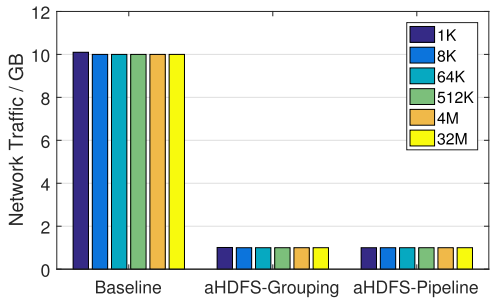


Fig. 16. Network traffic of aHDFS and the baseline scheme.

of *aHDFS-Grouping* and *aHDFS-Pipeline* is attributed to frequent accesses of the mappers to the local *key-value* store (see Sections 4.1.1 and 4.2.1).

Figs. 15b and 15c shows that *aHDFS-Pipeline* embraces neither shuffle nor reduce phases. Recall that *Baseline* and *aHDFS-Grouping* have a reduce phase being responsible for receiving output from mappers and computing final results to be written into HDFS. In *aHDFS-Pipeline*, the last finished mapper residing in the last node of a pipeline is responsible for writing parities to HDFS.

The shuffle and reduce tasks in *aHDFS-Grouping* take less time than those in *Baseline* in the large-file-size cases. For example, *aHDFS-Grouping* shortens *Baseline*'s shuffle time of by up to 85.7 percent when the file size is 20 GB (see Fig. 15b). A similar trend in the reducers can be observed from Fig. 15c. Such a performance improvement of *aHDFS-Grouping* over *Baseline* is more pronounced when we increase file size, because *key-value* pairs with same keys emitted from mappers on each node are grouped by *aHDFS-Grouping* as a single final *key-value* pair shuffled to reducers.

aHDFS-Grouping improves performance by (1) reducing network I/O traffic and (2) cutting down the data amount of $key - value$ pairs processed by the reducers (see Section 4.1.6). The evidence shown in Fig. 16 indicates that *aHDFS* is conducive to lowering the network I/O traffic of *Baseline* by approximately to 87 percent.

6.3 Block Size

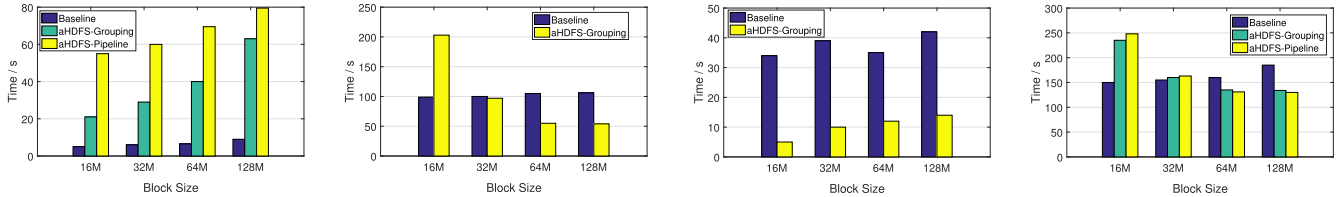
To evaluate the archival performance impacts of block size on *aHDFS*, we set the block size to 16, 32, 64 and 128 MB, respectively. Fig. 17d shows that *aHDFS-Grouping* and *aHDFS-Pipeline* can cut *Baseline*'s total archiving time by approximately 20 percent when the bock size is set to 64 and 128 MB. In what follows, we show the comparison of the three schemes with respect to execution times of the map, shuffle, and reduce phase.

Fig. 17a shows that block size has a big impact on mapper performance. For example, when block size increases from 16 to 128 MB, the map execution times of *Baseline*, *aHDFS-Grouping*, and *aHDFS-Pipeline* are enlarged by 60, 300, and 60 percent, respectively. Given files with fixed size, a small block size triggers a large number of mappers spawned, which in turn push up the parallelism level. *aHDFS-Grouping* outperforms *aHDFS-Pipeline* at the map phase, because mappers at the downstream nodes of a pipeline has to wait for predecessor nodes to deliver *key-value* pairs. Delaying mappers in the predecessor nodes slows down the process of the entire pipeline.

Fig. 17b reveals that *aHDFS-Grouping*'s shuffle time is two times longer than that of *Baseline* when block size is 16 MB. Nevertheless, increasing block size helps in shortening *aHDFS-Grouping*'s shuffle time; in contrast, the increased block size slightly worsens the shuffle time of *Baseline*. Consequently, when the block size is larger than or equal to 32 MB, *aHDFS-Grouping* is superior to *Baseline* in terms of shuffle time. Fig. 17c shows that a large block size leads to long reduce execution time. *aHDFS-Grouping* exhibits small reduce time in the small-block-size case, because each node only shuffles one block to reducers, the processing time of which depends on block size.

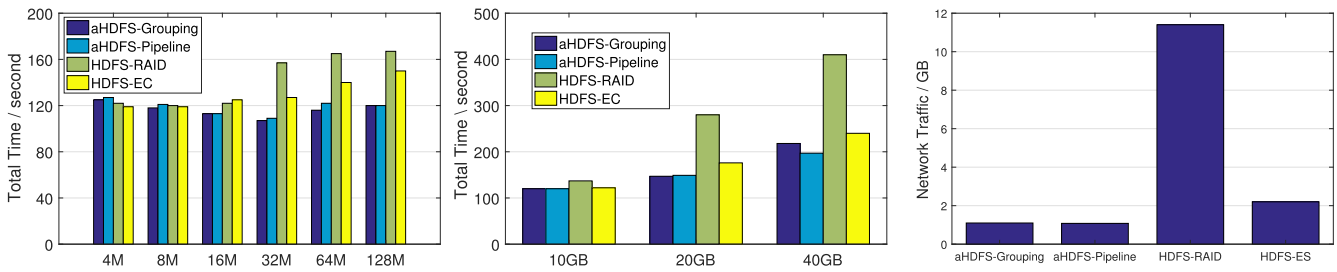
6.4 Putting it All Together

Now we provide an overall performance evaluation of aHDFS by comparing it with the two existing systems, namely HDFS-RAID [15] and HDFS-EC [20]. Fig. 18a shows the data archiving times of our aHDFS-Grouping and



(a) *Map Time* (b) *Shuffle Time* (c) *Reduce Time* (d) *Total Time*

Fig. 17. The execution time of the map, shuffle, and reduce phases along with the total execution time of the three systems under various block size.



(a) *Impacts of Block Size* (b) *Impacts of File Size* (c) *Network Traffic*

Fig. 18. An overall performance comparisons of aHDFS, HDFS-RAID, and HDFS-EC.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

aHDFS-Pipeline with the two alternatives under various block size. The experimental results show that when block size is larger than 32 MB, aHDFS improves the performance of HDFS-RAID and HDFS-EC by approximately 31.8 and 15.7 percent, respectively. If the block size is as small as 16 MB, aHDFS shorten the execution time of HDFS-RAID and HDFS-EC by 7.3 and 9.6 percent. No noticeable improvement is observed if the block size is below 8 MB. Two factors contribute to the performance improvements. First, aHDFS significantly shortens the shuffle and reduce phases in the large-block-size cases, because aHDFS combines all intermediate output Key-Values into a single Key-Value on each node. Second, aHDFS optimizes performance through the grouping (i.e., local key-value stores) and pipelining techniques (see Section 4).

Fig. 18b reveals the archiving performance of aHDFS, HDFS-RAID, and HDFS-EC as a function of file size. The performance trend observed in Fig. 18b is similar to that shown in Fig. 18a. Fig. 18c reveals that the grouping technique of aHDFS coupled with a pipeline significantly reduces network traffic for data archival workload. For example, aHDFS lowers the network traffic of HDFS-RAID and HDFS-EC by 90.5 and 50.9 percent, respectively.

7 Related Work

Data Archival Techniques. There are a handful of studies focusing on data archival techniques to save data storage space. For example, You et al. proposed the Deep Store-a large-scale archival storage system-to store immutable data efficiently and reliably for long periods of time [26]. Venti [27] is a network storage system intended for archival data. Schwarz et al. investigated a disk-scrubbing process in a large archival storage system where disks are periodically accessed to detect disk failures [28].

Erasure-Coded Data Archival. Erasure codes are widely adopted in cloud storage platforms [7], [29] and large-scale data centers [10], [11]. Huang et al. proposed an efficient scaling scheme called *Scale-RS* for Reed-Solomon-coded storage clusters [25]. Very recently, Huang et al. exploited pipelined encoding process by incorporating two data layouts (i.e., [D+P]cd and [3X]) to boost erasure-coded data archival performance on storage clusters [24]. Pamies-Juarez et al. developed two solutions-a decentralized erasure coding process [30] and *RapidRAID, pamies2013rapidraid*.

Unlike the aforementioned *Scale-RS* and *RapidRAID* that are designed for clusters and RAID storage, our *aHDFS* aims to improve data archival performance for Hadoop storage clusters.

Archiving Data in the Hadoop Distributed File System. Although there are a large body of studies aims at data archival in data centers, little attention has been paid to speed up archival process in HDFS. A few pilot projects were kicked off to address this problem. For example, Gupta et al. developed an active data archival scheme for data warehouses incorporating HDFS to efficiently store data [32]. While Gupta's approach is focused on the performance optimization of querying archived data, our *aHDFS* pays attention to improving archiving performance through parallel computing and pipelining. In particular, our *aHDFS* incorporates parallel archiving and pipeline archiving that take full advantage of the MapReduce programming model and the 3X-replica data layout to boost data archival performance on Hadoop clusters.

TABLE 4
Comparison of Archival Systems

Storage Systems	Archival	Erasure Code	HDFS	Pipeline
DeepStore [26]	✓	×	×	×
Venti [27]	✓	×	×	×
ScaleRS [25]	×	✓	×	×
[D+P]cd [24]	✓	✓	×	✓
RapidRAID [31]	✓	✓	×	×
HDFS-RAID [15]	×	✓	✓	×
HDFS-EC [20]	×	✓	✓	×
aHDFS	✓	✓	✓	✓

Xia et al. presented a new erasure-coded storage system called HACFS as an extension to HDFS. Adapting to dynamically changing workloads, HACFS switches on the fly between two different erasure codes, namely, (1) fast code optimizing recovery performance and (2) compact code reducing storage overhead [20]. From the perspective of implementation, *aHDFS* differs from HACFS in that *aHDFS* implements the data archival functionality on top of HDFS at the application level rather than at the system level.

A Comparison Table. Some of the above-mentioned storage systems are summarized in Table 4. We compare the existing systems with our *aHDFS* from four major aspects: (1) data archival, (2) erasure codes, (3) HDFS, and (4) Pipelined encoding.

8 Conclusions and Future Work

This paper presents an erasure-coded data archival system-*aHDFS*-in the realm of Hadoop cluster computing. We proposed two archiving strategies called *aHDFS-Grouping* and *aHDFS-Pipeline* to speed up archival performance in Hadoop distributed file system or HDFS. Both the archiving schemes adopt the MapReduce-based grouping strategy, which wraps up multiple intermediate *key-value* pairs sharing same *key* into one *key-value* pair on each node. *aHDFS-Grouping* transfers the single *key-value* pair to a reducer, whereas *aHDFS-Pipeline* delivers this *key-value* pair to the subsequent node in the archiving pipeline.

We implemented these two archiving strategies, which were compared against the conventional MapReduce-based archiving strategy referred to as *Baseline*. The experimental results show that *aHDFS-Grouping* and *aHDFS-Pipeline* can improve the overall archival performance of *Baseline* by a factor of 4. In particular, *aHDFS-Grouping* and *aHDFS-Pipeline* speed up *Baseline*'s shuffle and reduce phases by a factor of 10 and 5, respectively. In addition, *aHDFS-Grouping* and *aHDFS-Pipeline* significantly lower the network I/O traffic by 87 and 89 percent, respectively.

As a future research direction, we will develop a data reconstruction system to deal with block failure issues on Hadoop clusters. We plan to apply the grouping and pipelining strategies to the reconstruction system to speed up the reconstruction process. To optimize reconstruction performance, we will investigate a way of choosing intermediate parity blocks to be kept in the local key-value store.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENTS

This work was supported in part by was supported by the U.S. National Science Foundation under Grants IIS-1618669, CNS-0917137, and CCF-0845257. Jianzhong Huang's work is supported by the National Science Foundation of China under Grant No.61572209 and the Fundamental Research Funds for the Central Universities under HUST: 2015MS006.

REFERENCES

[1] J. Wang, P. Shang, and J. Yin, "Draw: A new data-grouping-aware data placement scheme for data intensive applications with interest locality," in *Cloud Computing for Data-Intensive Applications*. Berlin, Germany: Springer, 2014, pp. 149–174.
[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
[3] D. Borthakur, "The hadoop distributed file system: Architecture and design, 2007," Apache Software Foundation, Forest Hill, MD, USA, 2012.
[4] B. Calder, et al., "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 143–157.
[5] S. S. Miller, M. S. Shaalan, and L. E. Ross, "Correspondent-centric management email system uses message-correspondent relationship data table for automatically linking a single stored message with its correspondents," U.S. Patent 6 615 241, Sep. 2, 2003.
[6] D. Robinson, *Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services Enables You to Reach Business Goals Faster*. Queensland, Australia: Emereo Pty Ltd, 2008.
[7] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 20–20.
[8] C. Huang, et al., "Erasure coding in windows azure storage," in *Proc. Usenix Annu. Tech. Conf.*, 2012, pp. 15–26.
[9] A. Thusoo, et al., "Data warehousing and analytics infrastructure at Facebook," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1013–1020.
[10] D. Ford, et al., "Availability in globally distributed storage systems," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 61–74.
[11] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.
[12] R. T. Kaushik and K. Nahrstedt, "T: A data-centric cooling energy costs reduction approach for big data analytics cloud," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 52.
[13] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[14] Colossus: Successor to the Google file system (GFS). (2012) [Online]. Available: http://www.highlyscalablesystems.com/3202/colossus-successor-to-google-file-system-gfs/
[15] M. Sathiamoorthy, et al., "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.
[16] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 331–342.
[17] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, Art. no. 19.
[18] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[19] Z. Ren, J. Wan, W. Shi, X. Xu, and M. Zhou, "Workload analysis, implications, and optimization on a production hadoop cluster: A case study on taobao," *IEEE Trans. Services Comput.*, vol. 7, no. 2, pp. 307–321, Apr.–Jun. 2014.
[20] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 213–226.
[21] J. Yin, J. Wang, J. Zhou, T. Lukasiewicz, D. Huang, and J. Zhang, "Opass: Analysis and optimization of parallel data access on distributed file systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 623–632.
[22] M. Tang, B.-S. Lee, X. Tang, and C.-K. Yeo, "The impact of data replication on job scheduling performance in the data grid," *Future Generation Comput. Syst.*, vol. 22, no. 3, pp. 254–268, 2006.
[23] J. Huang, F. Zhang, X. Qin, and C. Xie, "Exploiting redundancies and deferred writes to conserve energy in erasure-coded storage clusters," *ACM Trans. Storage*, vol. 9, no. 2, 2013, Art. no. 4.
[24] J. Huang, Y. Wang, X. Qin, X. Liang, S. Yin, and C. Xie, "Exploiting pipelined encoding process to boost erasure-coded data archival," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 2984–2996, Nov. 2015.
[25] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie, "Scale-RS: An efficient scaling scheme for RS-coded storage clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 6, pp. 1704–1717, Jun. 2015.
[26] L. L. You, K. T. Pollack, and D. D. Long, "Deep store: An archival storage system architecture," in *Proc. 21st Int. Conf. Data Eng.*, 2005, pp. 804–815.
[27] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proc. Conf. File Storage Technol.*, 2002, pp. 89–101.
[28] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng, "Disk scrubbing in large archival storage systems," in *Proc. IEEE Comput. Soc. 12th Annu. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2004, pp. 409–418.
[29] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, "Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 163–176.
[30] L. Pamies-Juarez, F. Oggier, and A. Datta, "Decentralized erasure coding for efficient data archival in distributed storage systems," in *Distributed Computing and Networking*. Berlin, Germany: Springer, 2013, pp. 42–56.
[31] L. Pamies-Juarez, A. Datta, and F. Oggier, "RapidRAID: Pipelined erasure codes for fast data archival in distributed storage systems," in *Proc. IEEE INFOCOM*, 2013, pp. 1294–1302.
[32] R. Gupta, H. Gupta, U. Nambiar, and M. Mohania, "Efficiently querying archived data using hadoop," in *Proc. 19th ACM Int. Conf. Inf. Knowl. Manage.*, 2010, pp. 1301–1304.



Yuanqi Chen received the BS degree in computer engineering from Anhui University, in 2008 and the PhD degree in computer science from Auburn University, in 2016. His research interests include energy-efficient computing, storage systems, big data, and parallel and distributed systems.



Yi Zhou (S'16) received the BEE and MSEE degrees in electronic engineering all from Beijing University of Technology, Beijing, in 2006 and 2010, respectively. He is currently a doctoral student in the Department of Computer Science and Software Engineering, Auburn University. Prior to joining Auburn University in 2013, he has been a software engineer in Alcatel-Lucent Technologies (China) Co.,Ltd. for four years. His research interests include energy-saving techniques, database systems, and parallel computing. He is a student member of the IEEE.



Shubbhi Taneja (S'15) received the BE degree from Maharishi Dayanand University, Haryana, India, in 2012. She is working toward the doctoral degree in computer science in the Samuel Ginn College of Engineering, Auburn University, in 2013. Her main area of research include parallel and distributed systems, storage systems, energy-efficient computing, and performance evaluation. Her other areas of interest include web development and STEM programs for K-12 students. She is a student member of the IEEE, the ACM, and the IEEE Computer Society.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.



Xiao Qin (S'00-'04M-SM'09) received the BS and MS degrees in computer science from Huazhong University of Science and Technology, in 1992 and 1999, respectively, and the PhD degree in computer science from the University of Nebraska-Lincoln, in 2004. He is currently a professor in the Department of Computer Science and Software Engineering, Auburn University. Prior to joining Auburn University in 2007, he had been an assistant professor in the New Mexico Institute of Mining and Technology (New Mexico Tech) for three years. He won an NSF CAREER award in 2009. His research is supported by the US National Science Foundation. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation. He is a senior member of the IEEE.



Jianzhong Huang received the PhD degree in computer architecture, in 2005 and completed the post doctoral research in information engineering with Huazhong University of Science and Technology (HUST), in 2007. He is currently an associate professor in the Wuhan National Laboratory for Optoelectronics at HUST. His research interests include computer architecture and dependable storage systems. He is a member of the China Computer Federation (CCF).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:03 UTC from IEEE Xplore. Restrictions apply.