IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 10, 2021, accepted October 31, 2021, date of publication November 11, 2021, date of current version November 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3127499

Hadoop Data Reduction Framework: Applying Data Reduction at the DFS Layer

RYAN NATHANAEL SOENJOTO WIDODO[1], HIROTAKE ABE[2], (Member, IEEE), AND KAZUHIKO KATO[2], (Member, IEEE)

[1]Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba 305-8557, Japan
[2]Department of Computer Science, University of Tsukuba, Tsukuba 305-8557, Japan

Corresponding author: Ryan Nathanael Soenjoto Widodo (ryannsw(at)osss.cs.tsukuba.ac.jp)

ABSTRACT Big-data processing systems such as Hadoop, which usually utilize distributed file systems (DFSs), require data reduction schemes to maximize storage space efficiency. These schemes have different tradeoffs, and there are no all-purpose schemes applicable to all data. Users must select a suitable scheme in accordance with their data. To accommodate this requirement, application software or file system (FS) have a fixed selection of these schemes. However, these provided schemes are insufficient for all data types, and when novel schemes emerge, extending the selection can be problematic. If the source code of the application or FS is available, the source code could potentially be extended with extensive labor, but could be virtually impossible without the code maintainers' assistance. If the source code is unavailable, there is no way to tackle the problem. This paper proposes an unexplored solution through a modular DFS design that eases data reduction scheme usage through existing programming techniques. The advantages of this presented approach are threefold. First, adding new schemes is easy and they are transparent to the application code requiring no extensions to it. Second, the modular structure requires minimal modification to the existing DFSs and performance overhead. Third, users can compile schemes separately from the DFS without the FS or DFS source code. To demonstrate the design's effectiveness, we implemented it by minimally extending the Hadoop DFS (HDFS) and named it the Hadoop Data Reduction Framework (HDRF). We designed HDRF to work with minimal overhead and tested it extensively. Experimental results indicate that it has negligible overhead over existing approaches. In a number of cases, it can offer up to 48.96% higher throughput while achieving the best result in storage reduction within our tested setups because of the incorporated data reduction schemes.

INDEX TERMS Data compression, data deduplication, distributed file system, Hadoop, HDFS.

I. INTRODUCTION

Big data processing systems utilize file systems (FSs) that are scalable such as distributed FSs (DFSs) to hold large datasets. These systems are often configurable to use data reduction schemes such as compression and deduplication to maximize storage space efficiency [1]–[3] at the cost of extra integration time and processing overhead. These schemes have unique characteristics that enable them to work better for specific dataset types. For example, most deduplication schemes work best on virtual machine images [4]–[8] and LogZip [9] performs better on log file data than general-purpose algorithms such as Lz4 and Gzip. In such cases, it is important to use the best data reduction scheme for each data type to maximize storage space efficiency.

In a number of big data systems with a DFS such as Hadoop [10], data reduction schemes can operate in three different layers (application, DFS, and FS) on the basis of how the DFS software works as a middleware as shown in Figure 1. At the application layer, the applications can directly generate and reduce the data before passing it to the underlying layers. At the DFS layer, the DFS software processes the data from the application with the schemes before storing it in the underlying FS. At the FS layer, the FS receives the data from the DFS software, processes it, and stores it in the storage drives. Each of these layers has its advantages and disadvantages depending on the ease of adding and enabling the schemes, and the processing overhead of the schemes.

The associate editor coordinating the review of this manuscript and approving it for publication was Victor Sanchez.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

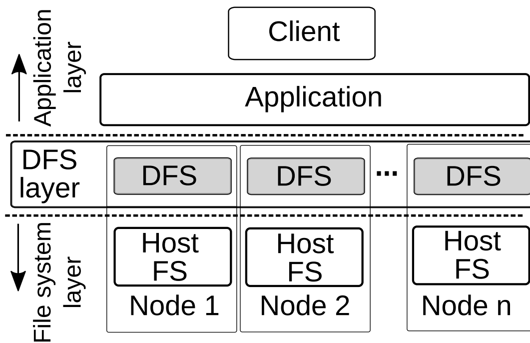

FIGURE 1. Possible locations for data reduction scheme applications in systems with DFS.

There are no perfect solutions that can combine all of the advantages without any of the disadvantages.

At the application layer, the applications perform the data reduction schemes, which can be a tedious process because users must enable the schemes for each application. Hadoop solves this issue by using Hadoop CompressionCodecs. The user can simply add new schemes as CompressionCodecs and integrate the base CompressionCodec library in the application. Next, the users can change the applications' configuration to enable the application to load the CompressionCodecs. However, this solution still depends on the support of CompressionCodecs in the applications. When the applications lack this support, users must modify each application to include the support, which is only possible when the application code is available, and even then, it could be a tedious process. In regard to overhead, because the application generates the data, directly reducing the data in the application also reduces the operational overhead in the lower layers. For example, the DFS layer can operate on the reduced data, minimizing the network traffic between the storage nodes.

Data reduction schemes at the FS layer operate conversely to those at the application layer because the FS works transparently from the point of view of the application. This transparent operation requires no change in the applications' code, which is simpler for end-users. However, adding new schemes to the FS code is more challenging because the FS code is often more complicated than the applications' code. Directly modifying the FS code without any help from the FS's developers might be challenging and may break the existing DFS setup. In regard to the operation overhead, the DFS and FS layers must process the original data with its larger size. When compared with the application layer-side data reduction, the DFS software may generate more network traffic. An example of this approach is by using ZFS, an FS that can perform compression and deduplication, as the base FS for Hadoop DFS [3], [11].

To the best of our knowledge, DFS layer-side data reduction is an unexplored domain because of a lack of research into the integration of data reduction schemes in the DFS software. However, by considering the structure of existing DFS designs and the role of the DFS in Hadoop, we can hypothesize and compare the schemes' operation in the DFS layer with the other layers. Adding new schemes to the DFS layer can be as challenging as modifying the FS code because of the complexity of the DFS code. However, enabling the schemes in the DFS layer is as easy as that in the FS layer because it lies below the application layer, enabling it to work on all applications' data. Additionally, network traffic reduction is possible because the DFS software is aware of the schemes in it.

Figure 2 summarizes the challenges for the application, FS, and DFS layers. Unlike the application and FS layer-based reduction, these challenges are solvable with a clever DFS design that is aware of and can use data reduction schemes. However, existing DFS designs do not enable the use of reduction schemes in the DFS layer because it may increase the software complexity, which in a number of cases is already more complex than the application code and underlying FS code. Additionally, directly modifying the existing DFS code may break the DFS functionality and cause data loss. We compiled a set of research questions for this study to explore the feasibility and characteristics of data reduction schemes in the DFS layer.

1) Is it possible to enable the data reduction schemes in the DFS software without affecting most of its functionality?
2) How easy is it to add and enable the schemes in the DFS software?
3) How big is the performance overhead for the schemes in the DFS software when compared with the existing solutions?

To confirm these hypotheses and to answer the research questions, this study proposes a new DFS design that can apply data reduction schemes at the DFS layer. The design uses existing DFS designs with slight modifications to the storage portion. The benefits of this design are threefold. First, users can easily add their favorite schemes to the DFS software through programming techniques like dependency injection or dynamic library linking. Second, it can enable schemes transparently on all applications without any modification on the application side. Third, the schemes are not limited to data reduction, which opens the possibility of data processing at the DFS layer.

As a proof of concept, we created a framework that enables data reduction schemes to work at an FS-level by directly providing HDFS blocks to the reduction scheme called the Hadoop Data Reduction Framework (HDRF) through HDFS source code modification. HDRF handles all communication between HDFS and a reduction scheme and enables the user to implement their schemes in Java or C++ with minimal knowledge of Hadoop Libraries. HDRF operates at the DFS level, which provides data reduction for all applications without depending on the application and FS layers. Our experimental results show that HDRF has minimal time and space overhead over a vanilla HDFS. It also enables data reduction schemes for all applications without any modification to the

	Application layer	File system layer	DFS layer
Difficulty of adding new schemes	○ Platform libraries	△ Difficult even with source code mod.	△ Difficult even with source code mod.
Application source code transparency	✖ Application code required	○ Application code not required	○ Application code not required
# of compression and network traffic reduction	○ Minimal # at lower layers	✖ No data reduction outside FS layer	○ Possible reduction in DFS and FS layer
Example	Hadoop Compression Codecs	HDFS on ZFS Lustre on ZFS	Proposed method HDRF

FIGURE 2. Comparison of the application, DFS, and FS layers for applying data reductions in a number of big data systems with a DFS.

application code. HDRF can enable CompressionCodecs on all applications, which can be impossible through the application layer without any application-side code changes.

Our previous work [12] demonstrated that HDRF can apply data reduction at the DFS layer on all applications. However, because of time and space limitations, we were unable to include a feature called block mirroring and do extensive testings with newer hardware. In this paper, we extensively discuss the design and implementation, add a feature called block mirroring, and present our experimental testings' result with both the old and new hardware. In summary, the contributions of this paper are as follows:

1) States the issues with existing methods for data reduction schemes in DFSs.
2) Proposes a DFS design that enables data reduction schemes in the DFS layer.
3) Implements a proof of concept of the design based on HDFS.
4) Evaluated the performance overhead over various workloads on different cluster setups and compared it with the existing solutions.

The rest of the paper is organized as follows. Section II explores the existing approaches, challenges, and our solutions. Section III discusses the design and implementation of the proposed DFS design. In Section IV, we compare the design with existing solutions over various workloads. Finally, we summarize our paper in Section V.

II. BACKGROUND AND RELATED WORK

This section discusses the challenges of existing DFS designs for data reduction schemes and our motivation to solve the challenges.

A. DATA REDUCTION SCHEMES

Data reduction schemes like compaction, compression, and deduplication can solve the data growth, which is a challenge for most storage systems, especially those that handle large data like cloud storage systems [6], [13]–[15], virtual machine images storage systems [4]–[8], and big data platforms [16]–[20]. These schemes work by reorganizing the data more efficiently to minimize potential redundancies at the cost of computation, and thus minimizing the storage footprint [20]. In a number of cases where the disk is significantly slower than the processing unit, the reduction in I/O operations can improve the system performance from storing or reading fewer data. In this paper, we focus on lossless compression and deduplication.

Lossless compression is a reversible process that removes redundancy within a file or data block. This type of scheme can be found in many applications, including memory compression [21]–[23] and file compression (e.g. Snappy [24], Lz4, Bzip2, and Gzip). For file compression, most lossless compression algorithms are based on a dictionary coder algorithm.

A dictionary coder operates by finding redundancies within a file. Let file f be a set of n blocks b, $f = \{b_1, b_2, \ldots, b_n\}$, and each b is composed of m words w, $b = \{w_1, w_2, \ldots, w_m\}$. The dictionary coder uses a data structure called a dictionary d, which is a subset of b, that maintains i unique w, $d \subseteq b$. When it encounters a redundant w, the dictionary coder replaces it with a pointer p to the corresponding unique w in the dictionary. In summary, the dictionary coder produces a compressed file f' with a total size of $f' = \sum b'$, where $b' = d + \sum_{x=1}^{m-i} p_x$ and $b' \ll b$ for best-case scenarios. However, when $m - i$ is close to 0, b' can be larger than b because of the overhead of d.

Deduplication is also a lossless process that maps files into smaller files called chunks with chunking algorithms [25], [26] and stores unique chunks in containers. Deduplication finds duplicate chunks by comparing the chunks' fingerprints, which are obtainable through mathematical hash functions like SHA1. Storage-saving is achieved by replacing duplicate chunks with pointers to the existing chunks. To reconstruct the original file back, deduplication systems use the file's recipe, which lists the chunks that correspond to the file. In comparison to compression, deduplication can remove redundancies on a larger scale such as those among files and storage devices. However, deduplication is usually more memory-intensive because it needs to compare a larger number of chunks. Deduplication works well for datasets that share similar parts like virtual machine images [27] and large storage systems [28].

Deduplication is similar to a dictionary coder, but it works with a set of files, $\{f_1, f_2, \ldots, f_m\}$ and $f = \{b_1, b_2, \ldots, b_n\}$, and at a more coarse granularity. It exploits the possibility of duplicate blocks b, which are also called chunks in deduplication, within f and among files, $f_a \cap f_b \cap f_c = duplicate\ b$. Because comparing blocks with bit-by-bit comparison is slow when considering the size of b, deduplication uses a mathematical hash function to produce a hash h_b or the fingerprint of each chunk and uses it in the comparison process. If multiple b have a matching hash value, then only a single instance of b and its hash h_b is stored in a data structure or database DB, $DB = \sum(b + h_b)$. p. Finally, it replaces the file with a recipe r that contains a list of hashes h_b of the chunks in the original file, $f = \{b_1, b_2, \ldots, b_n\} \rightarrow r = \{h_{b_1}, h_{b_2}, \ldots, h_{b_n}\}$. In the best-case scenario, the total size, which is $sumr + DB$,

is significantly smaller than *sumf*. However, in a number of cases where the number of redundant *b* is too small, the space overhead of the *DB* and *r* might be larger than the total size of redundant *b*.

Several studies proposed the use of data deduplication in a DFS through the application layer [29]–[31]. These approaches are more beneficial for data size reduction because data deduplication works best with a large dataset. However, data read processing through the DFS is a challenge because the deduplicated data is not the same as the original data. Directly processing the deduplicated data with a platform-provided data access method like Hadoop MapReduce and Apache Spark may result in an inaccurate output. In such cases, the application may not be able to leverage the distributed computing capability of the DFS and it requires extra processing to revert the deduplication process to produce an accurate result.

B. DISTRIBUTED FILE SYSTEM (DFS)

A DFS is a cluster of storage nodes that is scalable both vertically and horizontally. Each storage node can be expanded vertically by attaching more storage devices. Once the limit of vertical scaling is reached, the DFS can still grow through horizontal scaling or by adding more storage nodes to the cluster. This approach enables the DFS to keep up with data growth and increase storage I/O performance through parallelization.

Although DFSs are scalable in every direction, data growth is still a problem for all storage devices. To resolve this problem, DFSs can be combined with data reduction techniques like compression and deduplication. Because the DFS is software or middleware that connects the application to another FS, it can be split into three layers on the basis of the location of the data reduction, which are the application, DFS, and FS layers, as shown in Figure 1. At the application layer [16], [32]–[34], the application must support data reduction schemes to benefit from more efficient space utilization. At the FS layer [3], [35], this is no longer an issue because the data processing of the FS operates separately from the applications in their layer.

C. HADOOP AND HADOOP DISTRIBUTED FILE SYSTEM

Hadoop [10] is a commonly used big data platform that uses the Hadoop DFS (HDFS) [11] to store datasets in blocks. HDFS has two nodes: the name node, which handles the files' metadata and blocks' location, and the data node, which stores fixed-sized blocks. HDFS saves these blocks through another underlying FS like EXT4, NTFS, or ZFS. To ensure the blocks' reliability and availability, it uses a block replication scheme, which puts several copies of the same block in different data nodes to prevent losing blocks during data node failures. This replication process occurs when a node receives a packet from a client or another node.

HDFS, unlike other DFSs like Lustre [36], supports the MapReduce programming paradigm to parallelize data processing in the cluster, and thus enabling data nodes to perform data processing or compute. The MapReduce code can also be combined with Hadoop CompressionCodecs to optimize the storage usage in HDFS. The supported codecs are Gzip, Bzip2, Lz4, Snappy, and Zlib. Adding new reduction schemes as a codec is also possible. However, these codecs must be included in the application code to enable them. A simpler approach would be to use an FS like ZFS to enable compression or deduplication in HDFS or other DFSs, which do not need any modification in the application code.

Another concern of using the codecs in combination with MapReduce is split-ability, which is the possibility of independently decompressing each block. It is crucial because MapReduce may operate parallelly on each block. However, a number of codecs produce non-splittable blocks, which may require other blocks to decompress. In such cases, the MapReduce application cannot run in parallel and the data node may need to retrieve blocks that it does not have from other nodes. The FS-based approach has no such issue because each HDFS block is compressed independently from the DFS layer.

D. RELATED WORK ON DATA REDUCTION SCHEMES IN DFS AND HDFS

At the application layer, the users can add a scheme into the application code or enable it through a platform-specific API. For example, client-side compression can achieve a compression ratio of 1.5 in Lustre with Lz4 [37] and users of Hadoop can use Hadoop CompressionCodecs to enable schemes in their MapReduce [38] applications or directly add the scheme code to the application when using Hadoop [16], [29]–[34] or Lustre [36]. Applying the scheme at this layer also provides benefits for a reduction in network traffic at other layers, and thus speeding up the data transfer in the lower layers through the network. Additionally, the receiving node does not need to recompress the data independently when replication is set to above 1-time for HDFS as shown by Widodo *et al.* [12]. However, application code modification, which may not be possible if the code is not available, is mandatory to enable the scheme.

Several studies [29]–[31] have proposed data deduplication applications at the application layer that store the data in HDFS. Ranjitha *et al.* [29] and Sun *et al.* [30] studied deduplication applications that manage the output data in HDFS. Zhang *et al.* [31] explored the possibility of deduplication through the MapReduce programming model that passes the data to HDFS. The main drawback of these studies is that the deduplication cannot be independently reversed by each data node. These applications must revert the deduplicated data back to the original data and store it back in HDFS or other storage solutions before processing it to ensure consistent results, increasing the cost of data processing for Hadoop clusters.

Another common approach to enable data reduction schemes in a DFS is through the FS layer [3], [35]. For example, Zhou *et al.* [3] used ZFS as the FS for HDFS and enabled deduplication and compression through ZFS configurations.

This setup provides a transparent data reduction to the application and DFS layers. However, there is no reduction in network traffic because once the data leave the ZFS, the data is back to its original structure and size.

Data reduction schemes at the DFS layer can solve these challenges when the DFS is designed to run with the schemes. However, enabling them at the DFS layer has yet to be done because of a number of possible reasons. First, it adds complexity to the DFS, which increases the cost of development and the chance of breaking other components in the DFS. Second, on the basis of our experience, directly modifying the DFS source code to enable the schemes can be complex and time-consuming. Because of these reasons, it is difficult to compare reduction schemes at the DFS layer to the other approaches on the basis of the benefits and development costs.

In this paper, we proposed and tested a new DFS design that eases the data reduction schemes implementation and usage in a DFS. Users can use and enable the schemes through programming techniques like dynamic library linking or dependency injection. The DFS design is also transparent to all applications, enabling data reduction schemes to run on all applications' data. With this DFS design, users can add and enable these schemes at the DFS layer without modifying the DFS code.

III. DESIGN AND IMPLEMENTATION

This section provides the details of the proposed DFS design and the implementation of the design in HDFS.

A. THE DFS DESIGN

Implementing reduction schemes in the existing DFS design requires careful handling in several aspects. First, the ease of use from the user's perspective. Second, the placement of these schemes in the DFS may affect DFS functions like the metadata system and data integrity checker. Third, the DFS may not benefit from the network traffic reduction because of its smaller data size.

To overcome these challenges, we proposed a DFS design that supports data reduction schemes and extends existing DFS designs as shown in Figure 3. The key features of the design are: (1) it uses existing programming techniques to link the schemes' libraries to the DFS; (2) it runs the scheme close to the host FS to minimize the changes to other DFS components; (3) it can provide reduced and original data streams to the DFS and schemes to avoid reprocessing within the DFS.

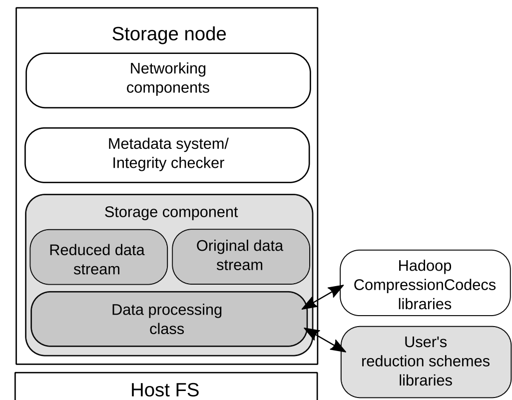

FIGURE 3. Proposed DFS design. The data reduction scheme is placed close to the host FS to minimize the changes in existing DFS designs. The grayed parts are the new additions to the existing DFS components.

1) EASE OF USE OF DATA REDUCTION SCHEMES

Adding and enabling data reduction schemes in the application and FS layers have their limitations. At the application layer, they depend on the capability of each application and its availability of the source code. For example, in Hadoop, the users can add new schemes through CompressionCodecs and enable them in the code or configurations when the application supports it. However, these can be difficult depending on the application's source code's availability. At the FS layer, enabling the data reduction schemes is not an problem because it is transparent to the applications. However, the adding part is like the application layer because it depends on the source code's availability. Even when available, modifying their code is difficult because its complexity is commonly higher than the applications' code.

The design of our DFS solves these problems through common programming techniques and by exploiting the nature of the DFS layer. The design uses the same approach as Hadoop CompressionCodecs, which connects to the application through independent Java jar files. Users can add new data reduction schemes to the DFS by extending a template data reduction class or by adding it through CompressionCodecs and extending the template class. The application can load these schemes through jar libraries. As for enabling these schemes, the design benefits from the nature of the DFS layer, which works transparently to the application layer like the FS layer.

2) SCHEMES PLACEMENT IN THE DFS

The implementation of the schemes in the DFS may harm a number of DFS functionalities depending on the placement. For example, running the schemes before the metadata system can generate errors because of the metadata mismatch between the client's and DFS's generated metadata. The design prevents such issues by activating the schemes before storing the data in the underlying FS, similar to FS-based reduction schemes like ZFS. This design also minimizes the changes over the existing DFS design because once the data leaves the storage part, it is back to its original form.

3) NETWORK TRAFFIC REDUCTION

One of the benefits of using data reduction schemes at the DFS layer is that it can also reduce network traffic, speeding up DFS operations that use the network. However, when these schemes work at the storage part of the system, there is no reduction in the network traffic because the networking part of the DFS usually operates above the storage components. Additionally, the DFS may also suffer from extra

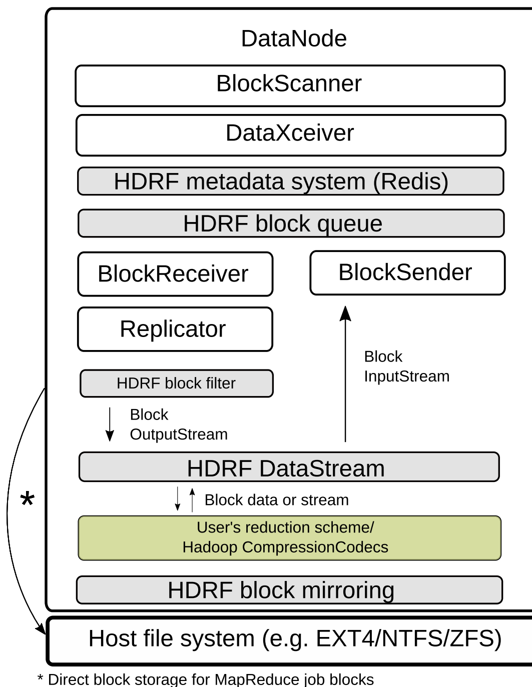

FIGURE 4. Proposed design on HDRF. The grayed parts are HDRF's components.

reprocessing during data replication. This reprocessing cost can be expensive for older nodes with low processing capability. To solve this design constrain, the proposed DFS design provides a selection of data streams, which contains the reduced and original data streams. Other DFS components can access both versions of the data outside of the storage portions of the DFS through these streams.

B. IMPLEMENTATION

As a proof of concept, we implemented the proposed design in HDFS, which is the default DFS for Hadoop, and named it HDRF. The implementation uses Hadoop 3.1.0 as the base and requires around 1000 lines of additional code and modification. Additionally, it works well with HDFS and MapReduce applications without any configuration or modification to the application code.

1) HDRF

As the name implies, HDRF is a framework that enables data reduction schemes in HDFS. HDRF operates within the DFS layer and works with all applications without any modification to their code. The data reduction schemes can be connected to HDRF through Java's dynamic library linking. This approach is similar to Hadoop CompressionCodecs, which works at the application layer, however, it works transparently with all applications. It records separate metadata for each block in Redis [39], [40] to ensure block integrity is not affected by the schemes. As shown in Figure 4, it requires a number of changes in HDFS's source code and has a number of features to minimize its processing overhead.

HDRF supports two types of streams: direct block array and processed data. The block array stream buffers the whole block data in the memory and can provide faster data access to the application by prefetching the block data in the memory. The processed data stream is similar to HDFS's file streams, which reads the data by chunks. Unless the user's data processing application requests the block array stream, HDRF uses the processed data stream because it is closer to the default code in the vanilla HDFS and minimizes the memory usage.

2) LOCAL METADATA SYSTEM

HDRF maintains a local metadata system in each node. This system functions as the translator for the addresses of each HDFS block in HDRF. When HDFS checks the length of the HDFS block file, HDRF queries this system and returns the stored length. HDRF also uses this metadata system to get the directory of the processed block from HDFS's blocks. The key for the HDFS block is its ID, and the value is the its length concatenated with the location of the processed block.

3) DATA INTEGRITY CHANGES IN HDFS

HDRF works within the storage part of HDFS by replacing HDFS's storage stream with its own stream, resulting in empty block files. Without any changes to the data integrity checker of the data node in HDFS, this will cause errors because of the non-matching length between the block file, which is zero bytes, and the metadata. HDRF prevents these errors by replacing the length check by retrieving length information from its metadata system and passing it to the data node length checker. There is no change to the data integrity check in the data node because HDFS performs it after HDRF produced the original blocks' data.

4) DEDUPLICATION THROUGH HDRF

To prove that users can implement their reduction scheme through HDRF, we created a simple deduplication scheme that splits the input data into small chunks, fingerprints the chunks with a hash function, matches the chunks with the hashes to find duplicates, and stores the chunks in a drive. The chunks metadata is stored in Redis. The scheme uses external libraries for chunking [26], hashing [41], and communicating with Redis [40]; the metadata database. The chunking algorithm is a content-dependent chunking (CDC) algorithm, which can automatically align the cut-point on the basis of the input data. However, it is still vulnerable to byte change, which may shift the cut-point or chunk boundary. In such cases, the affected chunk will be treated as a new chunk because it does not match the old chunk's hash. The deduplication scheme is connected to HDRF through the data reduction abstract class of HDRF. This scheme groups the chunks and store them in a large chunk called Superchunk to avoid random writes, which can be detrimental to most storage devices.

Figure 5 illustrates the structure of the key and value for the file's and chunk's metadata. This structure provides a maximum of 4 GB for block size, 256 MB for Superchunk size, and 16 MB for chunk length. "# of copy" is the number

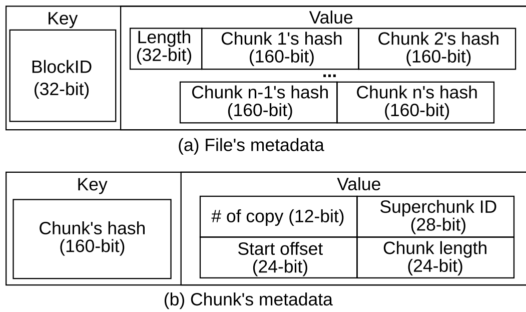

FIGURE 5. Key and value for: (a) file's metadata and (b) chunk's metadata.

of duplicates for a chunk, and the maximum is 4096. With these tables, the deduplication scheme can rebuild the file with a small amount of overhead. In our experiment, these tables amount to less than 5% of the original data size.

C. FEATURES OF HDRF

HDRF has several features to make it easier for the user to implement their preferred data reduction schemes in HDFS and minimize its performance overhead.

1) EASE OF ADDING NEW DATA REDUCTION SCHEMES

HDRF has two approaches for adding new data reduction schemes. First, through user configuration and dynamic linking, which is similar to Hadoop CompressionCodecs in the application layer. Second, by extending the data reduction abstract class of the HDRF. The former is similar to how Hadoop CompressionCodecs works in the application layer. The latter is more challenging because it needs to be compiled together with HDFS and HDRF. However, this solution is still easier than modifying the FS source code because the user can simply follow the abstract class to implement their new data reduction schemes.

2) TRANSPARENT DATA REDUCTION

Application layer-side data reductions have several benefits over other approaches. For example, data size reduction is more effective at the application layer with schemes like deduplication, which works better with a large dataset. Additionally, adding new schemes is easier through a few extra lines in the code or through dynamic library linking, which is used by Hadoop CompressionCodecs. Although the data reduction library is only compiled once, the user must change the code for each application to enable data reduction. In this case, DFS- and FS-side data reduction is easier for the user because it works for all applications with the tradeoff of less effective data reduction. However, adding a new data reduction scheme to the FS and DFS can be challenging depending on the availability of the source code and its complexity. In this aspect, HDRF combines the benefits of the application layer- and the FS layer-side data reductions. Adding new data reduction to HDRF is possible through dynamic library linking and changes in the configuration like the application layer-side data reduction, and it works transparently for all applications like the FS layer.

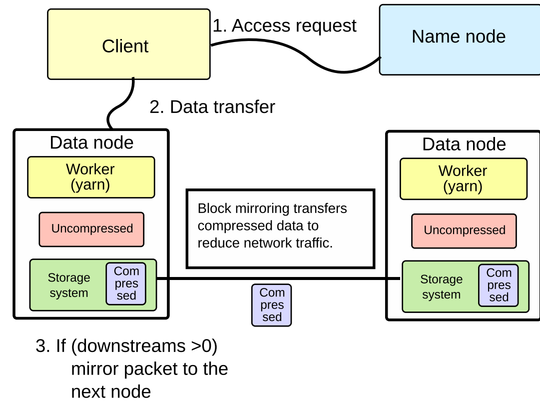

FIGURE 6. Block mirroring in HDRF. It checks the number of nodes in the downstreams and send the reduced data to the target nodes.

3) BLOCK MIRRORING

When the user applies data reduction schemes at the application layer, the lower layers like the DFS and FS layers can benefit from the reduced data size, and thus minimizing data transfer time during DFS operations that use the network. For example, we can reduce the data transfer time in HDFS by applying compression when replication is in use as shown in Figure 11. However, such benefits may not exist when applying them at the lower layers because the scheme must reconstruct the data back to its original form to make the process transparent at the upper layers. This limitation increases the data transfer time when compared with application layer-based schemes. An solution to this is to send the reduced data instead of the original data, which is possible for the DFS layer.

To minimize the network traffic, HDRF has a block mirroring feature, which disables the block replication part of HDFS and sends the reduced block data to other nodes. It uses scheme stacking to read the data from the data reduction scheme and send it to another node through TCP communication at a latency like that of the vanilla HDFS and CompressionCodecs. Figure 6 illustrates how the block mirroring works in HDFS. During a block write, HDRF performs a check on the downstream number. If the number is more than 0, it will start block mirroring and contact the target nodes. Once the process is completed, the block is registered and accessible in the other nodes.

4) SEAMLESS DATA ACCESS

A number of data reduction schemes can only process complete data, while others can process data block by block. To support these various methods of processing, HDRF has two block access modes: direct block array and direct stream access. In the block array mode, HDRF buffers the block data into an array and passes it into the scheme. HDRF can also revert the block array into a stream and pass it to HDFS for read-operations. However, block array mode requires higher memory consumption to buffer the block data. In the direct stream access mode, the HDRF directly passes

HDFS's stream data to the reduction scheme to minimize memory consumption and possibly latency from buffering the data. Additionally, HDRF can provide either the original or processed block data to the user's scheme for scheme stacking.

5) SCHEME STACKING

Additional processing of the blocks is often crucial to prepare them before or after they have been processed by the data reduction scheme. Although preprocessing can be done in the application layer through MapReduce code, this approach is inefficient because it must be enabled multiple times when applied to different applications. HDRF supports such processing through schemes stacking, which can connect a preprocessing scheme to a data reduction scheme or vice versa. This feature is possible because HDRF enables each scheme to request the original data or the scheme's processed data.

6) BLOCK FILTER

Processing MapReduce job files can be wasteful for the node's resources because MapReduce job files are often short-lived and discarded after job completion. Additionally, a number of data types may not benefit from data reduction schemes. For example, data reduction schemes may have a low reduction ratio when processing encrypted data.

HDRF has a block filter that searches for user-defined keywords in the content of the first received packet for each block. These blocks are then stored without any additional processing, and thus minimizing the node's resource usage. With the block filter, users can exclude blocks through HDRF configuration files and potentially save some performance.

7) BLOCK QUEUE

A vanilla HDFS can parallelly process multiple block requests, which speeds up the block-read process by not waiting for other requests' completion. However, not all data reduction schemes can work in parallel. For example, a number of deduplication schemes are limited to one thread to maximize their deduplication ratio [42]. In such cases, users can use the block queue system to limit parallel block accesses.

8) FAILURES HANDLING

System crashes and hardware failures are common and expected in production systems. HDFS handles this through block reporting. When a block is corrupted or has a mismatched length, the data node will report the block to the name node. Additionally, the name node will attempt to replace the block with one from other available nodes. HDRF takes the same approach as HDFS with a few changes. For the block size, a number of HDFS blocks in HDRF have 0-byte lengths. In this case, we made changes to the HDFS block length management system to check the length from the HDRF's metadata.

Another possible failure point of HDRF is the metadata server, which is isolated in each data node. When the metadata server fails, HDRF will report that the current node is broken to the name node. The name node will attempt to recover the data through the block replication of HDFS, which is similar to how HDFS handles node failures. HDRF handles other types of data corruption, node failures, and other unexpected events similarly to HDFS.

IV. EXPERIMENTAL RESULTS

This study answers the research questions presented in the Introduction by proposing and comparing HDRF to two existing setups, which are data reduction in the application and FS layers, respectively. For data reduction in the application layer, we used a vanilla HDFS setup where HDFS uses an ext4 FS and enables data reduction schemes through Hadoop CompressionCodecs in the applications. For data reduction in the FS layer, we used a ZFS setup where HDFS runs on top of ZFS and enables the schemes through ZFS.

We evaluated the proposed design by extending HDFS's code, which may affect its functionality when running various workloads. To show that the implementation can operate without any issue, we ran various workloads and observed HDFS's log in each data node. We ensured no errors such as mismatch checksum, missing blocks, or other related Java errors occurred in all setups. Additionally, for data processing results, we checked and compared the output.

Supporting new data reduction schemes is important to maximize the efficiency of the storage solutions. HDRF solves this through a data processing module that can load these schemes. HDRF can load any schemes that extend HDRF's data reduction scheme class or Hadoop CompressionCodecs class, which is similar to how Hadoop CompressionCodecs operates in the application layer. To demonstrate the operation of the solution, we implemented a local data deduplication scheme described in Section III-B, which extends HDRF's data reduction scheme. We also tested it in a series of tests to compare it with ZFS's deduplication scheme.

The difficulty of enabling the data reduction schemes may vary among setups. Enabling these schemes for the vanilla HDFS setup is the most challenging because a number of applications do not support them. As for the ZFS setup, enabling the schemes is as easy as changing configurations of ZFS because the data processing in ZFS is transparent to the applications. To confirm the difficulty of enabling these schemes for HDRF, this study used different applications on the three different setups, discussed the difficulty of enabling these schemes on these setups based on our experience, and observed the data processing and storage overhead.

The cost of operation is also important because if it is too big, it might deter users from using the setups. To show this cost, we ran several applications and data access tests to measure and compare the overhead of the setups. The goal for HDRF is to have a negligible difference from the best approaches.

Applications: The Hadoop applications that we used for the data access tests are Hadoop DistCP and Hadoop Streaming, which are both MapReduce-based and cannot and can use

TABLE 1. Specification for the test nodes.

	Cluster A	Cluster B	Name node
CPU	Xeon E-2224 (4c4t)	E3-1220L V2 (2c4t)	i3-7100 (2c4t)
Memory	32-GB DDR4	8-GB DDR3	24-GB DDR4
Storage	Samsung 983 DCT 960-GB (800 MBps write, 2.5 GBps read)	Micron CT250MX500SSD1 250-GB (400 MBps write, 500 MBps read)	Sandisk SDSSDH32 2-TB (400 MBps write, 500 MBps read)
Network	10-Gbps	1-Gbps	10-Gbps
Number of nodes	14	6	1

TABLE 2. Datasets used in the storage overhead tests.

Datasets	Content	Size
Wikipedia	A compilation of Wikipedia dumps from the first five dumps in 2021 (20210101, 20210120, 20210201, 20210220, 20210301) [43]	380-GB
PhysioNet	A compilation of medical datasets provided by PhysioNet[44], [45], [46], [47]	96.9-GB
Cocoimages	A compilation of 2017 unlabeled and train JPEG-formatted images from cocodataset [48].	36.8-GB

Hadoop's CompressionCodecs, respectively. With Hadoop DistCP, a user can define the number of mappers and reducers in the configuration when running. Hadoop Streaming can read and process the input data by using an input format before storing it in HDFS. The default input format is TextInputFormat, which reads a file line by line. However, it is not as useful for our testing because it adds a line number at the beginning of each line, increasing the storage space and making it difficult for us to compare it with Hadoop DistCP. To solve this, we made a custom input format based on TextInputFormat but without the extra line numbering. The number of mappers and reducers for Hadoop Streaming is defined by the split size. In the tests, we maintain the number of splits to match the number of mappers for Hadoop DistCP to ensure the fairness of the tests.

Additionally, we included Intel's big data benchmark called Intel HiBench, which can test the performance of the cluster when running MapReduce and Spark workloads. The workload that we used is Wordcount, which loads and reads the datasets directly from HDFS and then store the results back in HDFS. HiBench can generate the dataset by using a MapReduce application with a configurable size.

Cluster Specifications: Table 1 lists the specification of the clusters. We used two clusters, A and B. Cluster A has more nodes, higher compute power, more I/O performance, and more importantly, a higher network speed. Cluster B consists of nodes with older hardware and a slower network speed, which is only 1-Gbps maximum. We ran all tests on Cluster A except for the network scaling test, which was run on both clusters to show the impact of replication scaling on older hardware. All tests were run three times and the results provided here are the average of the three runs.

Datasets: The evaluation used three different types of datasets to measure the storage overhead of the tested setups. The first dataset is the Wikipedia dump dataset with a size of almost 400 GB and represents a human-readable dataset. The second dataset is the medical dataset from PhysioNet [44] with a size of almost 100 GB and contains log data from machines like electrocardiograms (ECGs) and patient activity records. The third dataset is an image dataset from Cocoimages with a size of around 35 GB and contains JPEG-compressed training and unlabeled images for machine learning applications. Table 2 shows the details of the datasets. We chose these datasets to show the storage overhead of HDFS when storing various dataset types. This test uses 3-times replication, which is the default for HDFS.

The Structure of the Tests: This study answers the research questions presented at the beginning of this Section through several tests with the described applications, clusters, and datasets. Because directly answering the first (1) and second (2) questions is difficult, we structured the tests to observe the overhead of the tested setups and answered the questions through the data presented in the results. The test structure is as follows. First, this study measures the throughput of the clusters when uploading and downloading the dataset from the cluster in Subsection IV-A. Second, this study evaluates the storage consumptions in Subsection IV-B. Next, this study shows the performance of the tested setups in MapReduce and Spark workloads. Finally, this study discusses the network overhead when replication is enabled in Subsection IV-D.

This study shows the ease of adding new data reduction schemes in Subsections IV-A and IV-B through the implementation of the data deduplication scheme in HDRF. The ease of enabling the schemes can be observed in Subsection IV-A where we attempted to enable data reduction schemes in Hadoop DistCP and Hadoop Streaming and in IV-C where we attempted to use Lz4 on the tested setups for MapReduce and Spark workloads. The overhead is shown throughout the test results by comparing the results from the different setups.

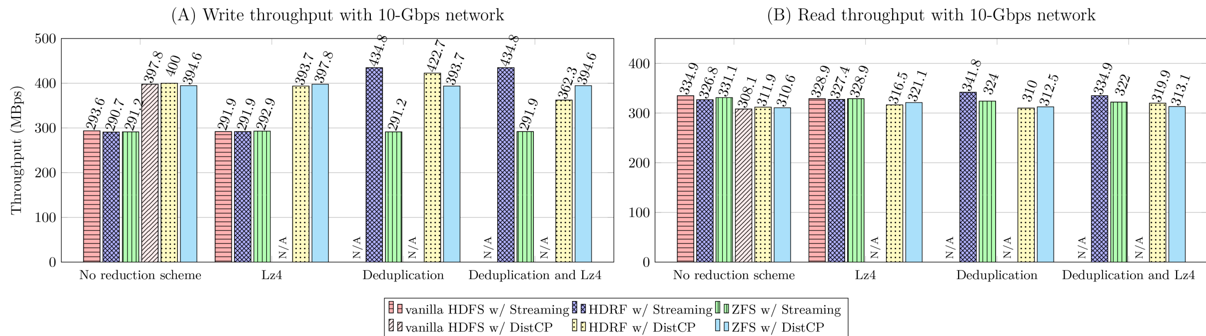

FIGURE 7. Throughput for dataset transfer between the client and Cluster A. DistCP on the vanilla HDFS cannot use any CompressionCodecs, and the vanilla HDFS does not officially support deduplication. Higher values are better.

A. DATA TRANSFER OVERHEAD TEST

To confirm the overhead of HDRF in data transfer, which is a basic and crucial task for DFSs, we compared HDRF with the vanilla HDFS and ZFS in a data transfer test. This test uses 50 GB of the Wikipedia dump dataset in the form of split 1-GB files to measure the data transfer speed for the tested setups. For the test configuration, we chose 1-time replication to show the compute overhead without being affected by the network during the replication. We simulated the 1-Gbps network by using Wonder Shaper 1.4.1 [49]. The applications used in these tests are Hadoop Streaming and DistCP.

Figure 7 presents the throughput recorded when transferring the dataset between the cluster and the client. In this test, we used the deduplication scheme described in Subsection III-B. Adding the scheme to HDRF is fairly easy through the provided abstract class. Throughout this test, we noticed no errors when the HDRF and ZFS setups ran the schemes in both Hadoop DistCP and Streaming. In contrast, for the application layer setup, the vanilla HDFS, we were unable to enable Hadoop CompressionCodecs on Hadoop DistCP through the command-line interface. DistCP ignored the CompressionCodec configuration in the command line and stored the data as if no CompressionCodec was specified.

For the write throughput, the differences between the vanilla HDFS, HDRF, and ZFS are less than 1% with the same reduction scheme, which shows that the overhead of HDRF is insignificant. Additionally, HDRF with deduplication enabled had close to a 50% higher throughput compared with that of the vanilla HDFS with Hadoop Streaming. The reason is that the deduplication scheme in HDRF uses the block array mode, which buffers the data in the memory before storing it in the disk, and thus maximizing the client's disk read bandwidth and network throughput. For the read overhead, the results depicted in Figure 7 show that the overhead is almost non-existent because the results for the vanilla HDFS, HDRF, and ZFS are negligible. In the best-case scenario, HDRF can eliminate the storage footprint by more than 50% while transferring the data at a higher throughput when compared with the vanilla HDFS without any reduction scheme.

Another observation we made from Figure 7 is that DistCP is up to 25% faster than Hadoop Streaming for write operation with most schemes because DistCP does not have extra processing. Additionally, DistCP reads and writes the data in chunks unlike Hadoop Streaming, which reads and writes in lines. However, with deduplication, the difference is less significant because HDRF absorbs small writes in memory for block array stream mode, used by the deduplication scheme. These results suggest that HDRF can provide many options to the users to minimize the data transfer time when applying data reduction schemes.

B. STORAGE OVERHEAD TEST

In this test, we uploaded three different datasets into Cluster A with Hadoop Streaming and measured the storage space usage of the tested setups. We chose Hadoop Streaming because it supports Hadoop CompressionCodecs Lz4, which is important when comparing the HDRF and ZFS setups. We then evaluated the storage overhead by comparing the space usage for the three different setups. We did not test the vanilla HDFS with deduplication because of the strict specification of Hadoop CompressionCodecs that requires the scheme to have stream support, which is missing from the described deduplication scheme in Subsection III-B.

Figure 8 shows the results for the storage used by each setup with no processing, Lz4 compression, deduplication, and the combination of the two. From this figure, we learned four facts. First, the overhead of HDRF is small and less than 1% of that of the vanilla HDFS across the three datasets with Lz4. Second, even though Hadoop CompressionCodecs and ZFS use the same Lz4 algorithm, they use different configurations. The Lz4 CompressionCodec in the vanilla HDFS and HDRF has a better compression ratio than that in ZFS. Third, HDRF's deduplication can reduce up to 15% of the dataset storage consumption, which is better than the deduplication scheme used in ZFS because HDRF's scheme uses a CDC algorithm. Fourth, neither Lz4 nor deduplication performs well across any of the datasets. Lz4 performs best with the human-readable text in the Wikipedia datasets with over 50% data reduction and reduces around 14% of the storage footprint for the log data in the PhysioNet datasets. However, in the image dataset, neither Lz4 nor deduplication could reduce the dataset further. Even worse, deduplication increases the storage footprint because of the deduplication metadata. This last information also

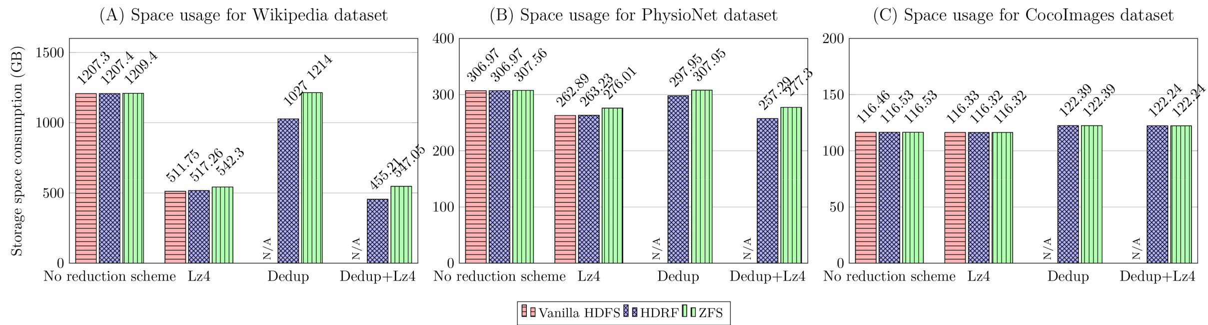

FIGURE 8. The storage space required to store the datasets with Cluster A. No processing means the dataset is directly stored as is. Dedup means deduplication is applied to the dataset. The vanilla HDFS does not officially support deduplication. Lower values are better.

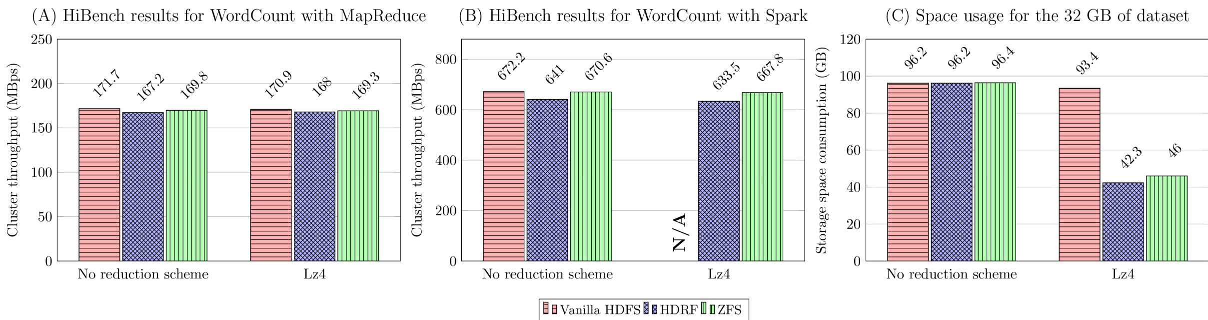

FIGURE 9. Throughput for the HiBench WordCount workload with Cluster A. Higher values are better.

shows the importance of the ease of adding new reduction schemes, to enable users to use a new reduction scheme to match their dataset's type.

C. DATA PROCESSING TEST

This test utilizes a commonly used big data benchmark tool from Intel called HiBench. It can measure the performance of the cluster when running MapReduce and Spark workloads. For this test, we generate a dataset with HiBench and run a test for data processing. We chose WordCount as our data processing task and ran it for all tested setups with 3-times replication, which is the default for HDFS. The test has two phases: prepare, which generates 32 GB of a ''huge'' dataset, and process, which uses MapReduce to perform the word count. The number of executors, mappers, and reducers is set to 28. The prepare phase uses MapReduce to generate random words and store them in HDFS.

With this test, we observed that HiBench's Wordcount MapReduce workload was unable to use Hadoop CompressionCodecs. We tried the configuration ''hibench.compress .profile enable'' and ''hibench.compress .codec.profile 'lz4''' but it still consumes the same storage space. However, adding compression options in the MapReduce command line reduces the dataset from 96 to 93 GB, which is still far larger than the other setups. In comparison, enabling Lz4 in HDRF can reduce the dataset to 42 GB, which is over 50% reduction in storage footprint. This observation indicates that HDRF is transparent to all applications without the hassle of configuring each application.

A similar situation also occurred when running the Spark workload counterpart. When we prepared the dataset with Hadoop CompressionCodecs, the process phase crashed stating that the Lz4 native library could not be loaded. We debugged the application for hours attempting to fix the issue and noticed that IOCommon failed to load the compressed dataset. Additionally, we confirmed with the ''hadoop checknative -a'' command that Lz4 is indeed installed properly in our cluster and is available natively. HDRF and ZFS can run the MapReduce and Spark workloads with Lz4 enabled without any issue. This experience on enabling Hadoop CompressionCodecs on HiBench shows that enabling the data reduction schemes can often be challenging and frustrating.

As for the overhead, the results shown in Figure 9A indicates that the overhead for data processing applications like MapReduce is around 3% for HDRF, and it can perform as well as the vanilla HDFS with and without compression. With the Spark workload, HDRF is around 5% slower than the vanilla HDFS because of the extra processing by the framework when loading the data from HDFS. The overhead with Spark is larger because Spark is much faster than MapReduce at processing the dataset, increasing the significance of the dataset load operation from HDFS. With a more compute-intensive workload, this overhead may become smaller.

D. NETWORK OVERHEAD TEST

One of the issues for lower-layer data reduction like HDRF and ZFS is the lack of network traffic reduction and the need to reprocess the data at each node during block replications, and thus increasing the data transfer time. For the application layer data reduction, these issues are not a concern because block replication occurs in the DFS layer and the data there is already reduced, eliminating the need to reprocess the data at the replication's destination nodes. To confirm

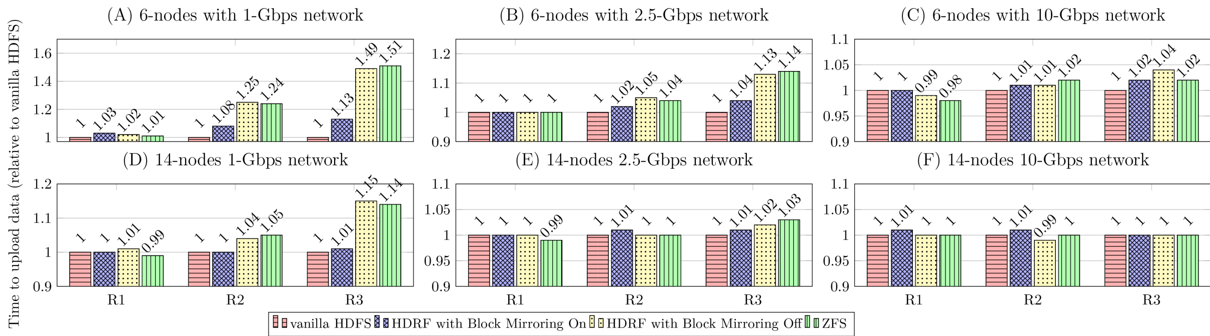

FIGURE 10. **Data upload time from the client to the cluster with replication scaling on Cluster A with 1-Gbps, 2.5-Gbps, and 10-Gbps network and Lz4 compression. The number behind R indicates the number of replication factors for each block. For example, R2 means 2-times replication. The result is relative to that of the vanilla HDFS. Lower values are better.**

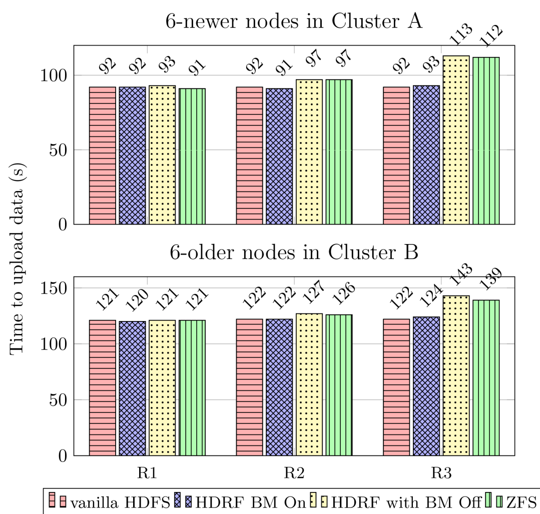

FIGURE 11. **Time to upload with replication scaling for six nodes in Clusters A and B on a 1-Gbps network. BM stands for Block Mirroring. Lower values are better. The results are not scaled to the baseline to show the difference between Cluster A and Cluster B.**

this hypothesis, we scaled the network speed and replication factor of HDFS with Clusters A and B. To show the overhead of replication, each node has the dataset stored locally in the solid state drive (SSD) to eliminate the overhead of the network transfer from the client to the data nodes. The dataset is the same as that used in the data transfer test, which is the first 50 GB of the Wikipedia dump.

Figure 10 depicts the results for the network scaling test on Cluster A. According to the results, the impact of replication is more significant with a slower network and fewer nodes. With slower networks, setups that reduce the data at lower layers like HDRF and ZFS significantly consumed more time to finish because they transfer the data without any data reduction. Additionally, they need to recompress the data at each replication's destination nodes, which is inefficient compared with the vanilla HDFS that reduces the data at the application layer. With fewer nodes, the effect is more significant because the amount of data is the same, meaning each node now must process more data. With block mirroring, HDRF can almost match the data transfer time of the vanilla HDFS with faster networks because it transfers the reduced data instead of the original data, and thus minimizing the data transfer time and eliminating the need to reprocessing at the destination nodes. With the slow 1-Gbps network, there is around 13% of data transfer overhead for HDRF with block mirroring enabled over the vanilla HDFS, which is significantly smaller than that without block mirroring.

As data processing also depends on the node capability of data processing, we also compared the newer nodes in Cluster A with the older nodes in Cluster B. In this test, we used the first 20 GB of the Wikipedia dump. As shown in Figure 11, the data transfer overhead is more significant in the newer nodes at around 23% when compared with the older nodes at around 17% overhead for ZFS and HDRF without block mirroring. The faster disks in the newer nodes decrease the amount of time to read and write the data to HDFS, increasing the importance of the data processing capability for a task like data compression. With the block mirroring, the overhead over the vanilla HDFS is smaller because the destination nodes do not need to reprocess the data.

E. DISCUSSION

With these tests, we answered the research questions presented at the Introduction, which are:

1) Is it possible to enable the data reduction schemes in the DFS software without affecting most of its functionality?
2) How easy is it to add and enable the schemes in the DFS software?
3) How big is the performance overhead for the schemes in the DFS software when compared with the existing solutions?

We answered the first question by adding the support for data reduction schemes to an existing DFS called HDFS with around 1000 lines of codes. We also tested it in various workloads and confirmed that there are no issues with its operation. We answered the second question by adding a deduplication scheme to HDFS through HDRF and enabling schemes such as Lz4 and deduplication in various applications. During the implementation and experiments, we observed that adding new data reduction schemes is fairly easy through the provided abstract class. Additionally, HDRF supports Hadoop CompressionCodecs, making scheme implementation as easy as those in the application layer. Enabling the schemes is

similar to the FS layer approach. HDRF's data processing is transparent like that of ZFS in the FS layer. To answer the third question, we compared HDRF with the existing approaches in various workloads and observed the overheads. The data transfer and storage space overhead is fairly low at around 1%. With the Spark and MapReduce workloads, the overhead is around 5%. These results indicate that the proposed DFS design can combine the benefit of existing approaches at a fairly low overhead cost.

With the test in IV-B, this work also demonstrates that not all data reduction schemes are equal. A number of schemes work better than others for certain data types. For example, HDRF's deduplication schemes can reduce more redundancy than those of ZFS. Additionally, deduplication and compression do not perform well on datasets with log or image files. This result shows the importance of having easy access to the DFS to add and enable the schemes to the DFS.

V. CONCLUSION AND FUTURE WORK

This study explores the application of data reduction schemes in big data systems with DFSs such as Hadoop. Existing approaches have challenges in terms of the ease of adding or enabling schemes in the system. These challenges might discourage users from using these schemes to maximize their storage space efficiency. This study proposes a new DFS design that can combine the benefits of existing approaches by applying the data reduction schemes directly in the DFS layer. This advantages of this approach are threefold. First, adding a new scheme can be as easy as doing so in the application layer. Second, it can apply the data reduction scheme transparently regardless of the application. Third, the overhead is negligible compared with that of the other approaches.

To show the effectiveness of the proposed design, we implemented it in an open-source DFS called HDFS and compared it with a vanilla HDFS and the data reduction scheme in the FS layer through ZFS. Our results and experience dealing with the tested setups show that adding a new scheme through HDRF is as simple as through the vanilla HDFS or even easier; enabling the schemes is as easy as the FS layer approach because it does not require any changes on the application side. Our results also indicate that HDRF has a small overhead of around 1% for data transfer and storage space overhead, and less than 5% for compute workload.

The implementation currently only exists for Hadoop with HDFS. Because other systems may have a different slightly different design or structure, it is important to check the feasibility of the design in other systems. Additionally, HDRF has other potentials such as the data processing capability of HDRF, which can improve the compatibility of HDFS with other storage devices without relying on other solutions with high processing overhead such as FUSE [50].

ACKNOWLEDGMENT

The authors would like to thank editors, reviewers, and readers who have given their time and valuable comments. Throughout their contribution, the authors were able to improve this work to its current state.

REFERENCES

[1] D. Zhao and I. Raicu, "Exploring data compression in distributed file systems," Dept. Comput. Sci., Inst. Technol., Illinois Inst. Technol., Chicago, IL, USA, Tech. Rep., 2013. Accessed: Mar. 25, 2021. [Online]. Available: http://datasys.cs.iit.edu/reports/2013_IIT-CS554_FusionFScompression.pdf

[2] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan, "Azure data lake store: A hyperscale distributed file service for big data analytics," in *Proc. ACM Int. Conf. Manage. Data*, May 2017, pp. 51–63.

[3] R. Zhou, M. Liu, and T. Li, "Characterizing the efficiency of data deduplication for big data storage management," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2013, pp. 98–108.

[4] J. Wei, H. Jiang, K. Zhou, and D. Feng, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol. (MSST)*, May 2010, pp. 1–14.

[5] N. M. Tyj and G. Vadivu, "Adaptive deduplication of virtual machine images using AKKA stream to accelerate live migration process in cloud environment," *J. Cloud Comput.*, vol. 8, no. 1, pp. 1–12, Dec. 2019.

[6] R. Kaur, I. Chana, and J. Bhattacharya, "Data deduplication techniques for efficient cloud storage management: A systematic review," *J. Supercomput.*, vol. 74, no. 5, pp. 2035–2085, May 2018.

[7] C. Lin, Q. Cao, J. Huang, J. Yao, X. Li, and C. Xie, "HPDV:A highly parallel deduplication cluster for virtual machine images," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 472–481.

[8] J. Xu, W. Zhang, Z. Zhang, T. Wang, and T. Huang, "Clustering-based acceleration for virtual machine image deduplication in the cloud environment," *J. Syst. Softw.*, vol. 121, pp. 144–156, Nov. 2016.

[9] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: Extracting hidden structures via iterative clustering for log compression," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 863–873.

[10] Apache Software Foundation. *Apache Hadoop*. Accessed: Oct. 25, 2019. [Online]. Available: https://hadoop.apache.org/

[11] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project*, vol. 53, nos. 1–13, p. 2, 2008.

[12] R. N. S. Widodo, H. Abe, and K. Kato, "HDRF: Hadoop data reduction framework for Hadoop distributed file system," in *Proc. 11th ACM SIGOPS Asia–Pacific Workshop Syst.*, Aug. 2020, pp. 122–129.

[13] B. Nicolae, "High throughput data-compression for cloud storage," in *Proc. Int. Conf. Data Manage. Grid P2P Syst.*, Springer, 2010, pp. 1–12.

[14] C. Tu, E. Takeuchi, C. Miyajima, and K. Takeda, "Continuous point cloud data compression using SLAM based prediction," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2017, pp. 1744–1751.

[15] K. Hossain, M. Rahman, and S. Roy, "IoT data compression and optimization techniques in cloud storage: Current prospects and future directions," *Int. J. Cloud Appl. Comput.*, vol. 9, no. 2, pp. 43–59, Apr. 2019.

[16] K. Rattanaopas and A. S. Kaewkeeree, "Improving Hadoop MapReduce performance with data compression: Study using wordcount job," in *Proc. 14th Int. Conf. Electr. Eng./Electron., Comput., Telecommun. Inf. Technol. (ECTI-CON)*, Jun. 2017, pp. 564–567.

[17] M. Usama and N. Zakaria, "Chaos-based simultaneous compression and encryption for Hadoop," *PLoS ONE*, vol. 12, no. 1, Jan. 2017, Art. no. e0168207.

[18] A. Ashu, M. W. Hussain, D. S. Roy, and H. K. Reddy, "Intelligent data compression policy for Hadoop performance optimization," in *Proc. Int. Conf. Soft Comput. Pattern Recognit.* New York, NY, USA: Springer, 2019, pp. 80–89.

[19] M. Kuhn, J. Kunkel, and T. Ludwig, "Data compression for climate data," *Supercomput. Frontiers Innov.*, vol. 3, no. 1, pp. 75–94, Jan. 2016.

[20] S. De Agostino, "The greedy approach to dictionary-based static text compression on a distributed system," *J. Discrete Algorithms*, vol. 34, pp. 54–61, Sep. 2015.

[21] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proc. 32nd Int. Symp. Comput. Archit. (ISCA)*, Jun. 2005, pp. 74–85.

[22] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 638–649.
[23] M. Zarubin, P. Damme, T. Kissinger, D. Habich, W. Lehner, and T. Willhalm, "Integer compression in NVRAM-centric data stores: Comparative experimental analysis to DRAM," in *Proc. 15th Int. Workshop Data Manage. New Hardw. (DaMoN)*, 2019, pp. 1–11.
[24] *Google Snappy*, Google, Mountain View, CA, USA, 2019.
[25] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 1337–1345.
[26] R. N. S. Widodo, H. Lim, and M. Atiquzzaman, "A new content-defined chunking algorithm for data deduplication in cloud storage," *Future Gener. Comput. Syst.*, vol. 71, pp. 145–156, Jun. 2017.
[27] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proc. SYSTOR Israeli Experim. Syst. Conf. (SYSTOR)*, 2009, pp. 1–12.
[28] Y. Shin, D. Koo, and J. Hur, "A survey of secure data deduplication schemes for cloud storage systems," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 1–38, Feb. 2017.
[29] S. Ranjitha, P. Sudhakar, and K. S. Seetharaman, "A novel and efficient deduplication system for HDFS," *Proc. Comput. Sci.*, vol. 92, pp. 498–505, Jan. 2016.
[30] Z. Sun, J. Shen, and J. Yong, "A novel approach to data deduplication over the engineering-oriented cloud systems," *Integr. Comput.-Aided Eng.*, vol. 20, no. 1, pp. 45–57, Jan. 2013.
[31] D. Zhang, C. Liao, W. Yan, R. Tao, and W. Zheng, "Data deduplication based on Hadoop," in *Proc. 5th Int. Conf. Adv. Cloud Big Data (CBD)*, Aug. 2017, pp. 147–152.
[32] K. Meena and J. Sujatha, "Reduced time compression in big data using MapReduce approach and Hadoop," *J. Med. Syst.*, vol. 43, no. 8, pp. 1–12, Aug. 2019.
[33] Y. Ji, H. Fang, H. Yao, J. He, S. Chen, K. Li, and S. Liu, "FastDRC: Fast and scalable genome compression based on distributed and parallel processing," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.* New York, NY, USA: Springer, 2019, pp. 313–319.
[34] S. Huang, J. Xu, R. Liu, and H. Liao, "A novel compression algorithm decision method for spark shuffle process," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 2931–2940.
[35] W. Fuzong, G. Helin, and Z. Jian, "Dynamic data compression algorithm selection for big data processing on local file system," in *Proc. 2nd Int. Conf. Comput. Sci. Artif. Intell. (CSAI)*, 2018, pp. 110–114.
[36] Lustre. *Lustre*. Accessed: Oct. 25, 2019. [Online]. Available: http://lustre.org/
[37] J. Kunkel, "Analyzing data properties using statistical sampling—Illustrated on scientific file formats," *Supercomputing Frontiers Innov.*, vol. 3, no. 3, pp. 19–33, Oct. 2016.
[38] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[39] S. Sanfilippo, "Redis is an open source (BSD licensed)," in *Memory Data Structure Store, Used as a Database, Cache and Message Broker*. Mountain View, CA, USA: Redis Ltd., 2019. Accessed: Oct. 25, 2019. [Online]. Available: https://redis.io/
[40] Jonathan Leibiusky. (Accessed:Oct. 25, 2019). *Jedis is a Blazingly Small and Sane Redis Java Client*. Accessed: 2019. [Online]. Available: https://github.com/redis/jedis
[41] Project Nayuki. (Accessed:Oct. 25, 2019). *Native Hash Functions for Java*. Accessed: 2019. [Online]. Available: https://github.com/nayuki/Native-hashes-for-Java
[42] H. Fan, G. Xu, Y. Zhang, L. Yuan, and Y. Xue, "CSF: An efficient parallel deduplication algorithm by clustering scattered fingerprints," in *Proc. IEEE Int. Conf Parallel Distrib. Process. With Appl., Big Data Cloud Comput., Sustain. Comput. Commun., Social Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, Dec. 2019, pp. 602–607.
[43] *Wikimedia Downloads: XML Dump for English Wikipedia*, Wikimedia, San Francisco, CA, USA, 2021.
[44] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, "PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. 215–220, 2000.
[45] M. G. Terzano, L. Parrino, A. Smerieri, R. Chervin, S. Chokroverty, C. Guilleminault, M. Hirschkowitz, M. Mahowald, H. Moldofsky, A. Rosa, R. Thomas, and A. Walters, "Erratum to 'Atlas, rules, and recording techniques for the scoring of cyclic alternating pattern (CAP) in human sleep' [sleep med. 2(6) (2001) 537–553]," *Sleep Med.*, vol. 3, no. 2, p. 185, Mar. 2002.
[46] G. B. Moody and R. G. Mark, "A database to support development and evaluation of intelligent intensive care monitoring," in *Proc. Comput. Cardiol.*, Sep. 1996, pp. 657–660.
[47] F. Andreotti, J. Behar, S. Zaunseder, J. Oster, and G. D. Clifford, "An open-source framework for stress-testing non-invasive foetal ECG extraction algorithms," *Physiolog. Meas.*, vol. 37, no. 5, pp. 627–648, Apr. 2016.
[48] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Computer Vision—ECCV*. New York, NY, USA: Springer, 2014, pp. 740–755.
[49] B. Hubert, J. Geul, and S. Séhier. (Accessed: Mar. 25, 2021). *Wondershaper: Command-Line Utility for Limiting an Adapter's Bandwidth*. Accessed: 2021. [Online]. Available: https://github.com/magnific0/wondershaper
[50] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 59–72.

RYAN NATHANAEL SOENJOTO WIDODO received the B.E. degree in electrical engineering from Petra Christian University, Indonesia, in 2013, and the M.Sc. degree in ubiquitous IT from Dongseo University, South Korea. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Graduate School of System Information Engineering, University of Tsukuba. His research interests include data deduplication and compression, cloud storage, distributed storage systems, persistent memory, and data structures.

HIROTAKE ABE (Member, IEEE) received the B.E., M.E., and Ph.D. degrees from the University of Tsukuba, Japan, in 1999, 2001, and 2004, respectively. From 2004 to 2007, he was part of the research staff of Japan Science and Technology Agency. From 2007 to 2010, he was an Assistant Professor with the Toyohashi University of Technology, Japan. From 2010 to 2012, he was an Assistant Professor with Osaka University, Japan. He is currently an Associate Professor with the University of Tsukuba. His research interests include system software, distributed systems, and computer security.

KAZUHIKO KATO (Member, IEEE) received the B.E. and M.E. degrees from the University of Tsukuba, Japan, in 1985 and 1987, respectively, and the Ph.D. degree from The University of Tokyo, Japan, in 1992. From 1989 to 1993, he was a Research Associate with the Department of Information Sciences, Faculty of Sciences, The University of Tokyo. He is currently a Professor with the Department of Computer Science, Graduate School of System Information Engineering, University of Tsukuba. His research interests include operating systems, distributed systems, and secure computing. He received the distinguished paper awards from JSSST and IPSJ, in 2004 and 2005, the software paper award from JSSST, respectively. He received the IPSJ fellow, in 2017, the JSSST Achievement Award, in 2018, and the Minister of Education, Culture, Sports, Science and Technology Award for Science and Technology, in 2020.