Optimizing Read Operations of Hadoop Distributed File System on Heterogeneous Storages

JONGBAEG LEE, JONGWUK LEE AND SANG-WON LEE
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, 16419, Korea
E-mail: jongbaeg.lee@gmail.com, {jongwuklee; swlee}@skku.edu

The key challenge in big data processing frameworks such as Hadoop distributed file system (HDFS) is to optimize the throughput for read operations. Toward this goal, several studies have been conducted to enhance read performance on heterogeneous storages. Recently, although HDFS has supported several storage policies for placing data blocks in heterogeneous storages, it fails to fully utilize the potential of fast storages (*e.g.*, SSD). The primary reason for its *suboptimal* read performance is that, while distributing read requests, existing HDFS only considers the network distance between the client and datanodes, thereby incurring more read requests to slower storages with more data (*e.g.*, HDD). In this paper, we propose a new data retrieval policy for distributing read requests on heterogeneous storages in HDFS. Specifically, the proposed policy considers both the unique characteristics of storages in datanodes and the network environments, to efficiently distribute read requests. We develop several policies including the proposed policy to balance these two factors such as random selection, storage type selection, weighted round-robin selection, and dynamic round-robin selection. Our experimental results show that the throughput of the proposed method outperforms those of the existing policies by up to six times in extensive benchmark datasets.

Keywords: Hadoop distributed file system, heterogeneous storage, data retrieval policy, MapReduce, load balancing

1. Introduction

Hadoop [1] is one of the representative distributed platforms for processing large-scale datasets. As a significant component of Hadoop, Hadoop distributed file system (HDFS) [2] is responsible for storing and managing data in a cluster of nodes. It is observed that, because the majority of HDFS applications follow write-once and read-many paradigm [3], their performances are heavily influenced by I/O throughput [4, 5]. Therefore, the optimization for read operations in the HDFS directly affects the overall throughput of Hadoop applications.

Toward this goal, a possible solution is to optimize I/O throughput on *heterogeneous storages*, which consists of different storage types such as solid state drive (SSD) and hard disk drive (HDD). Recently, several studies have been conducted to enhance I/O performance on heterogeneous storages. Specifically, they can be categorized into two types: (i) using fast storages as caches and (ii) maximizing overall data access throughput on heterogeneous storages. First, some caching methods use fast storages (*e.g.*, SSD) as caching areas for storing data on slow storages (*e.g.*, HDD) to avoid unnecessary I/O [6,

7, 8, 9]. Next, the overall performance is optimized by maximizing the I/O performance of the fast storages [10, 11, 12] . That is, as the I/O performance highly depends on the characteristics of the storages, it is essential to design an efficient data selection for accessing heterogeneous storages.

Although the HDFS has supported several storage policies for placing data blocks in heterogeneous storages, it fails to fully utilize the potential of fast storages (*e.g.*, SSD) when retrieving data blocks. For instance, the One_SSD policy [13] employs both SSD and HDD for write requests, while only network distance is used to select datanodes for read requests. As it simply ignores the I/O performance gaps across different storages, the read performance in the existing HDFS could be suboptimal.

In this paper, we propose a new data retrieval policy for distributing read requests on heterogeneous storages in HDFS. The proposed policy particularly considers both the unique characteristics of storages in datanodes and the network environments between client and datanodes. We aim to optimize read throughput by (i) maximizing the utilization of heterogeneous storages and (ii) balancing the selection of storage types at runtime. Thus, in addition to the traditional network-distance-based selection, we develop three additional policies for balancing two factors: storage type selection, weighted round-robin selection, and dynamic round-robin selection.

In particular, it is observed that preferentially reading data from fast storages incurs the *starvation* problem of slow storages. Thus, we dynamically balance the number of read requests according to the storage performance and check whether requests are skewed to one storage. This dynamic strategy can drastically mitigate different utilization gaps across heterogeneous storages.

To summarize, the contributions of this paper are as follows:

- We observed that the storage policies for HDFS on heterogeneous storages fail to leverage the full potential of fast storages. To address this problem, we describe four data retrieval policies, including HDFS's existing read policy, and propose a read-optimized data retrieval policy that not only maximizes the utilization of heterogeneous storages but also balances the selection of storage types at runtime (Section 3).

- We evaluate the performance of our method by comparing it against several data retrieval policies including that of existing HDFS. From extensive experiments using representative datasets, we confirm that our method outperforms existing policy by 2.2–6 times in terms of total throughput (Section 4). Moreover, we confirm that the method shows improved performance compared to other methods which take into account only the type of storage device.

The rest of the paper is organized as follows. Section 2 describes an overview of HDFS and its storage policies including a slow reading problem in heterogeneous storage policies. Section 3 discusses the impact of the network environment and storage types on performance and presents the design details of four data retrieval policies. In Section 4, we evaluate the performance of four data retrieval policies including the proposed policy using DFSIO and I/O-intensive HiBench workloads. Section 5 reviews the recent work enhancing the performance of Hadoop systems by increasing storage systems' performance. Finally, Section 6 summarizes the contributions of this paper.

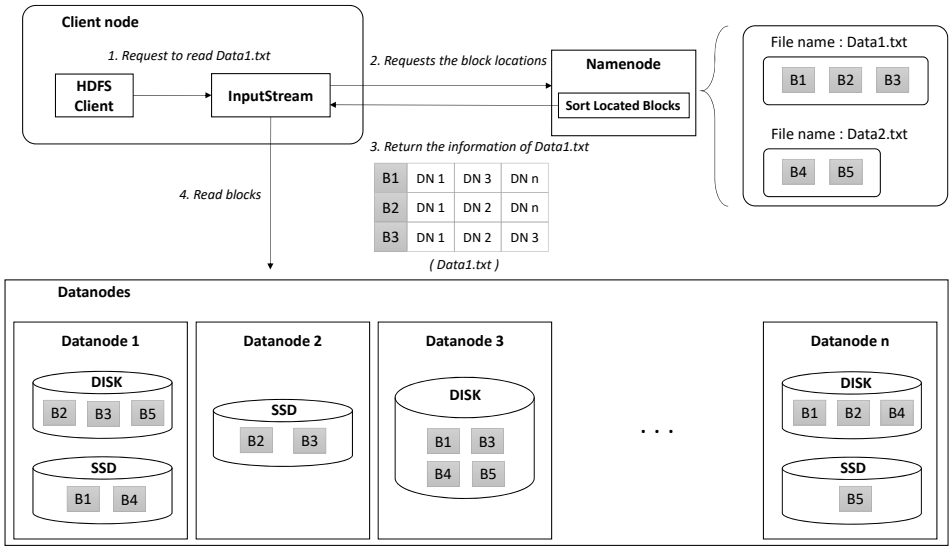


Fig. 1. Architecture of HDFS on heterogeneous storages

2. Background

2.1 Hadoop Distributed File System (HDFS)

HDFS is a distributed file system designed to store and access files on a cluster of commodity servers [2]. HDFS cluster is organized as master/slave architecture. The master, called *namenode*, stores the HDFS metadata and is responsible for controlling file access. The slave, called *datanode*, stores files managed by HDFS and is responsible for serving read/write operations on files. The nodes in HDFS are managed in units of rack according to the network distance. The bandwidth between nodes on the same rack is typically larger than that between the nodes on different racks.

In HDFS, a file is split into one or more *blocks* with each block stored in one or more *datanodes*. The same block is replicated and stored in multiple nodes. (By default, the block size is 128MB and the replication factor is 3.) It thus allows fault tolerance and parallel read through data replication.

Next, we explain how read/write operations are performed in HDFS. Read /write operations essentially consist of the interactions across a client node, a namenode, and datanodes [14]. A client node, which can be a namenode, a datanode, or another node outside the cluster, requests a read/write operation. At the client node, *HDFS Client* object requests operations from/to the file system, *InputStream/OutputStream* object retrieves the metadata from the namenode, and read/write blocks from/to the datanode are operated.

The read operation is divided into two steps: *open* and *read*. Figure 1 details the overall process by which a client reads 'Data1.txt' that is stored in HDFS. First, the open step consists of acquiring the file's metadata, opening the file, and preparing to read it. In this step, the HDFS Client object creates an InputStream object that helps us to search and opens the file. The InputStream object, when opening the file, receives metadata about the file 'Data1.txt' through the namenode. The metadata includes block information on which block the file is composed of(Data1.txt consists of B1, B2, B3.). As each HDFS block is replicated on several datanodes, the information of the datanode with the replication

Policy type	Policy name	Description
Homogeneous	Hot(default)	**All** replicas are stored in **DISK**
	Cold	**All** replicas are stored in **ARCHIVE**
	All_SSD	**All** replicas are stored in **SSD**
Heterogeneous	Warm	**One** of the replicas is stored in **DISK** **The others** are stored in **ARCHIVE**
	One_SSD	**One** of the replicas is stored in **SSD** **The others** are stored in **DISK**
	Lazy_Pesist	**One** of the replicas is stored in **RAM_DISK** **The others** are stored in **DISK**

Table 1. Descriptions of storage policies in HDFS

location is included in one block information in the form of an array (B1 is stored in datanode 1, datanode 3, and datanode n.).

Next, the read step consists of InputStream connecting to the datanode where the block is stored and reading the block. Among multiple replications, reads are performed on the datanode that is positioned at the front of the location array of replications. In existing HDFS, as the array is sorted according to the network distance between the client and the datanode, it reads the block on *nearest* datanode based on the client's logical *network distances* from the client. If there are several replications at the same network distance, HDFS randomly selects and reads the block.

The write operation is divided into two steps: *create* and *write*. In create step, HDFS Client requests to create or open a file for writing. To write a file, the namenode performs various checks, such as checking permission for the write operation. Further, the namenode provides the location of the datanodes where the replications are to be stored. If the replication factor is 3, the first replication is stored on the local machine if the client is on a datanode or on a random datanode. The second replication is stored on a datanode on a different rack. Finally, the third replication is stored in another node in the same rack as the first replication. After obtaining the information from the namenode, in the write step, the client writes the block to the first datanode. Then, the first datanode writes the block replications to other datanodes. Finally, the write is completed when the first datanode receives acknowledgments from other datanodes.

2.2 Storage Policies on HDFS

Currently, HDFS supports four storage types including *DISK*, *SSD*, *ARCHIVE*, and *RAM_DISK*. DISK represents the hard disk drive, and SSD represents the solid-state drive. ARCHIVE represents an archive storage device with large capacity and small computing power. Finally, RAM_DISK represents an in-memory storage device. HDFS also supports six rules, called storage policies, that determine the storage type in which each replication is stored. Table 1 depicts the storage policies for block replications in HDFS [13]. According to the storage types used to store the replications, it can be classified into *homogeneous* and *heterogeneous* cases.

The storage policies for homogeneous storages store data in clusters with the same storage types, whereas those for heterogeneous storages store data in clusters with two different storages. In heterogeneous storage policies, the policies store one replication in the fast storage and the remaining replications in the slow storage. As they store one of the replications in the fast storage, it is possible to improve overall throughput on

heterogeneous storages.

However, HDFS clusters using the storage policies for heterogeneous storages have been observed to not *fully* achieve performance improvement. Figure 4(a) shows the storage utilization obtained while running the DFSIO benchmark. (In Section 4, we will explain the description for benchmark datasets in detail). In this experiment, we observed the performance of the HDFS cluster using One_SSD policy. In an experimental setting, all datanodes are at the same network distance from the client, and each datanode uses one SSD and one DISK. The experimental result shows that the utilization of DISK is high, while that of SSD is too low. This problem also occurs when using the other heterogeneous storage policies (*i.e.*, Warm and Lazy_Pesist).

This is because HDFS uses the network-distance-based data retrieval policy that reads a replication from the nearest datanode among multiple block replications. In a single rack setting with the same network distance between datanodes, the network-distance-based policy randomly selects any of the replications in the cluster. When a cluster of hardware in HDFS consists of homogeneous storages, this approach is effective for optimizing overall throughput. (In this study, the throughput is measured in bytes per second for read/write operations.)

However, when HDFS consists of heterogeneous storages, network-distance-based selection could not make full use of the fast storage devices. In a heterogeneous storage environment, as the throughput of storages is different, the time required to complete a read request differs depending on the storage type. If the same number of reads is assigned to both storage types, the fast storages complete the read requests and wait for the slow storages to finish reading. The heterogeneous storage policies aggravate the problem. As existing HDFS assumes that storages are homogeneous, it distributes the read requests across datanodes according to the amount of the data stored without considering the storage characteristics. As the heterogeneous storage policies store fewer blocks in the fast storages, fewer reads are allocated to the fast storages. Although fast storages can handle more reads than slow storages at the same time, they become idle after completing a small number of allocated reads. As a result, the throughput of fast storages cannot be fully exploited, and the overall performance degrades.

3. Proposed Method

In this section, we describe four data retrieval policies, including HDFS's network-distance-based selection, and present the proposed policy that dynamically considers not only the network environment but also the storage characteristics to improve the HDFS read performance with heterogeneous storages. To describe the design of policies in detail, we discuss the impact of network environment between client and datanode, one of the factors affecting the read performance, present how to consider the network environment when reading for each storage type, and provide the detailed description of four data retrieval policies.

The key idea of the proposed method is as follows:

- **Maximizing the utilization of heterogeneous storages**: When selecting a datanode on heterogeneous storages, we consider the network environment as well as the storage characteristics to maximize the utilization of heterogeneous storages.
- **Balancing the selection of storage types**: We prevent the starvation of slow storage by statically and dynamically controlling the selection of storage types to alleviate heavy accesses of fast storage devices.

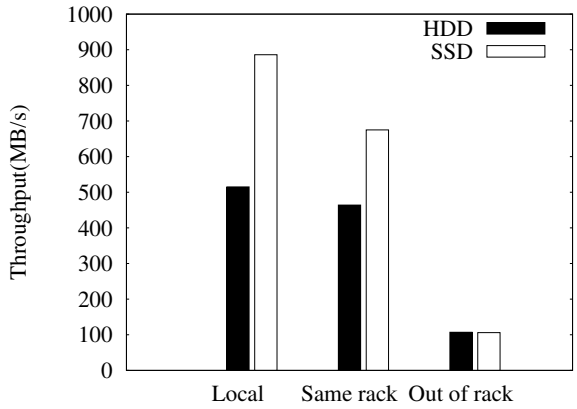


Fig. 2. HDFS Performance by network distance

3.1 Network Environment Between Nodes

The network environment between the client and the datanode is an important factor for optimizing read performance. HDFS is composed of a two-level network topology consisting of multiple racks of nodes. The client and the datanode may be the same node, different nodes in the same rack, or nodes in different racks. In the existing HDFS, the client is designed to read from the datanode closest to the network because of the slow performance caused by network traffic between racks by saturating the rack-to-rack switch [15].

In addition to the logical distance between the client and data nodes, network bandwidth also affects HDFS' read performance. When a client reads a block from a datanode, the amount of data transferred from the datanode to the client is limited to the network bandwidth between the client and the datanode. If the network bandwidth between two nodes is smaller than the storage's maximum throughput, the read operation cannot fully utilize the storage. In a typical HDFS cluster configuration, the network bandwidth inside the rack is larger than the bandwidth between the other racks; therefore, slow storages in the same rack may provide better performance than fast storages in other racks.

For example, Figure 2 illustrates the read throughput of each storage according to the network distance between the client and the datanode. In this experiment, all nodes are connected in the LAN. The nodes in the same rack are connected through 10Gbps Ethernet. In the case of rack-to-rack connection, we assume that they are connected in the WAN. To simulate a WAN environment, we limited the rack-to-rack network bandwidth by connecting them through 1Gbps Ethernet (In this simulation, we do not address the other network issues except bandwidth.). The result shows that the closer the network distance is, the higher the throughput is measured. Furthermore, it is slower to access SSD on the out of rack than to access DISK nearer to SSD despite SSD being considerably faster than DISK. Thus, when reading from fast storage, considering both the network bandwidth and distance provides higher performance while reducing unnecessary network traffic.

To accurately consider the network environment, we pre-configure the network bandwidth between the nodes, the network bandwidth between racks and storage's maximum throughput information in addition to the logical network distance. The four data retrieval policies described in Section 3.2 select either fast storage or slow storage to read

a block according to each rule. Whether to read a block from the selected storage device is determined by considering the pre-configured network environment, and how the network environment is considered depends on the storage type. First, when trying to read from slow storage, the policies read a block from the closest datanode to obtain high performance without saturating the rack-to-rack switch. Next, when trying to read from fast storage, the policies determine whether to read a block according to the network environment between the client and the datanode. In particular, the following factors are considered.

1) If the client and the datanode are the same, read a block stored in the fast storage.
2) If the datanode is another node in the same rack as the client, the policies compare the network bandwidth inside the rack with the slow storage's maximum throughput.
   2.1) If the network bandwidth inside the rack is larger than the slow storage's throughput, read a block stored in the fast storage.
   2.2) Otherwise, read a block from the closest datanode.
3) If the client and data nodes are in different racks, read a block from the closest datanode.

3.2 Storage Selection Policies

Notation	Description
N	Number of blocks to read
$BlockSize$	Block size in HDFS
Rep	Number of replications
$Throughput_{fast}$	The maximum throughput of fast storages
$Throughput_{slow}$	The maximum throughput of slow storages

Table 2. Notations to describe the performance of each storage selection policies

We present several data retrieval policies, including network-distance-based selection, storage type selection, weighted round-robin selection, and dynamic round-robin selection, to consider the storage characteristics. Table 2 shows the notations used to describe the performance of HDFS cluster for each data retrieval policy. Because HDFS reads from the datanode located at the first of the replication array from the namenode, each policy uses different methods for sorting the replication array. The policies implemented on the *sortLocatedBlocks* function of the *DatanodeManager* class that is responsible for sorting the replication array.

Network-distance-based selection: This method, which is the existing HDFS's read policy, selects any data replication regardless of the storage type. This policy randomly selects storage by sorting the replications in an arbitrary order. Therefore, more read requests are processed on the storage where more blocks are stored. For example, in Figure 1, the replications of block B1 are stored in datanode 1, datanode 3, and datanode N. Thus, this policy selects any of the three datanodes.

In the homogeneous storages, if the blocks are evenly distributed, the network-distance-based selection evenly distributes read operations across storages. Therefore, the expected time to read N blocks from the HDFS cluster is $(N \times BlockSize)/Throughput$.

However, in the heterogeneous storages, the number of blocks to be stored and the performance is uneven depending on the storages. In this case, slower storages store more blocks and process more requests, which degrades overall performance. Specifically, the network-distance-based selection stochastically requests slow storages to read ($N \times (Rep-1)/Rep$) blocks while requesting fast storages to read the ($N \times 1/Rep$) blocks. While reading more blocks from the slow storages, the fast storages become idle after completing the allocated read. Therefore, the total execution time is equal to the time of reading ($N \times (Rep-1)/Rep$) blocks from the slower storages.

Storage type selection: This method preferentially selects a replication stored in a fast storage type. This policy aligns the block stored in fast storage on the front of the replication array. In block B1 shown in Figure 1, the block replication on datanode 1 is stored in the SSD, and the remaining replications are stored in DISKs. In that case, the replication of datanode 1 is sorted first, and the block on the datanode is read. Thus, storage type selection can increase the utilization of fast storages, and the expected read throughput is $Throughput_{fast}$. The result of the DFSIO benchmark in Section 4 shows that the utilization of SSD, compared to the existing HDFS, is increased by an order of magnitude. However, this policy only reads from fast storages, thereby incurring the starvation problem of slow storages.

Weighted round-robin selection (WRR): This method selects fast storage and slow storage in an interleaved manner. For example, as shown in Figure 1, if blocks B1 and B2 are read, then it reads B1 from DISK and reads B2 from SSD, allowing us to distribute the read operation regardless of the amount of stored data. To further optimize this, we first obtain the weight used to adjust the number of interleaving of storage selection. The weight indicates how fast storage can handle more reads than the slower storage. The higher the performance gap between the storage devices, the higher the value assigned. The weight is calculated according to the following equation.

$$weight = \frac{Throughput_{fast}}{Throughput_{slow}} \tag{1}$$

Next, we obtain the ratio of the amount of data read from fast storage to that from slow storage. We record the number of blocks read from each storage type during execution and calculate the ratio of the number of blocks read from fast storage to that from slow storage for every read. As HDFS reads files in fixed-size blocks, the ratio of the number of blocks read has the same meaning as the ratio of the amount of data read. The ratio of the amount of data read is calculated according to the following equation.

$$ratio = \frac{the\ number\ of\ blocks\ read\ from\ fast\ storages}{the\ number\ of\ blocks\ read\ from\ slow\ storages} \tag{2}$$

Through the weight and the ratio, WRR selection distributes the reads according to each storage's performance. While reading a file, if the ratio is less than or equal to the weight, it reads from the fast storage because it suggests that the fast storage is not fully utilized. On the other hand, if the ratio is larger than the weight, it reads from the slow storage as it indicates that the fast storage is fully utilized while slow storage is less utilized. Thus, WRR selection reads from the slow storage once while reading the *weight* times from the fast storages.

For example, if we assume that the read throughput of fast storage is W times faster than that of slow storage, it suggests that the fast storage can read W times more blocks than slow storage at the same time, and the weight is set to W. The value of W may vary depending on the environment of the cluster. Therefore, when reading a file composed of

W + 1 consecutive blocks, *W* blocks are read from the fast storage while one block is read from the slow storage for balanced reading.

In this case, WRR selection requests slow storages to read $(N \times 1/(W+1))$ blocks while requesting fast storages to read the $(N \times W/(W+1))$ blocks. Theoretically, the time to read $(N \times 1/(W+1))$ blocks from slow storages is same as the time to read $(N \times W/(W+1))$ blocks from fast storages. Therefore, both storages do not become idle and finish reading within $(N \times 1/(W+1))/Throughput_{slow}$(this value is equal to $(N \times W/(W+1))/Throughput_{fast}$).

Dynamic round-robin selection (DRR): This method operates similar to WRR selection except that it dynamically adjusts the weight. As WRR selection statically distributes read requests to storages, it results in an imbalance in distribution when the weight is incorrect. DRR selection mitigates the problem by monitoring the storage utilization at runtime to ensure that the read distribution is balanced and adjusting the weight if it is not balanced.

Before running HDFS, DRR selection statically configures threshold value for each storage type to determine whether to adjust the weight. In addition, it initially creates a background process that measures the system statistics and records the storage utilization of each datanode. Then, it periodically checks whether the average utilization of each storage type is updated during read. If the average utilization of fast or slow storages is measured to be lower than the threshold, it increases the weight to allow more access to fast or slow storages, respectively.

Depending on the threshold, this can control the aggressiveness of changing the weight. If the threshold is set low, the initial weight changes less aggressively. In contrast, if the threshold is set high, the weight increases aggressively if the value is for fast storages, while the weight decreases aggressively if that is for slow storages. Thus, it not only exploits the advantages of WRR selection but also flexibly handles the imbalance of read distribution that occurs during operation.

4. Experiment

4.1 Experimental Setup

Except for several experiments, our HDFS cluster consists of one namenode and three datanodes, with each node containing two 24-core CPUs (Intel Xeon CPU E5-2670 2.30GHz) and 64GB DRAM. All machines are connected through 10Gbps Ethernet in the LAN. The version of Hadoop is 2.6.0. The HDFS block size is 128 MB, and the replication factor is 3.

We used the One_SSD policy as the baseline method. We used one Samsung 850 Pro SSD 256GB and one Western Digital 7200RPM HDD SATA 1TB on each datanode. In our experimental environment, the throughput of an HDFS cluster with a single SSD is 300MB/s and that of a single HDD is about 60MB/s.

4.2 Results

We evaluate the performance of four data retrieval policies with three MapReduce [16] workloads as follows.

- *DFSIO*: This tests the method by which HDFS handles a large number of operations that simultaneously perform read or write. DFSIO measures the throughput of the cluster as a test result.

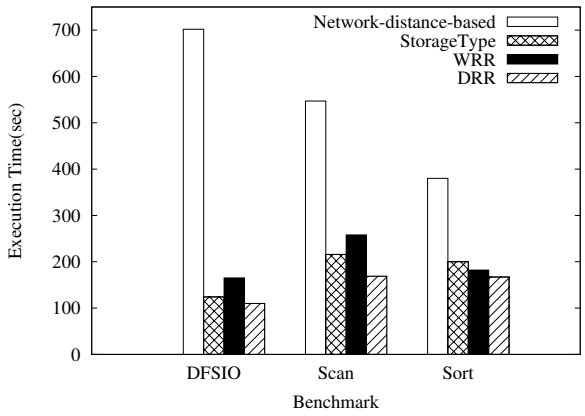


Fig. 3. Benchmarks execution time for each data retrieval policy

- *Scan* : This tests a Hive [17] query that creates two tables, reads one of the tables and copies it to the other table.
- *Sort*: This tests a MapReduce workload that reads textual input data from HDFS, sorts and merges the results and stores them in HDFS.

DFSIO is a file system benchmark of Hadoop, and Scan and Sort are I/O-intensive workloads provided by HiBench [18]. For the performance evaluation, the amount of data used to measure the performance is 128GB, 34GB and 30GB for DFSIO, Scan and Sort, respectively.

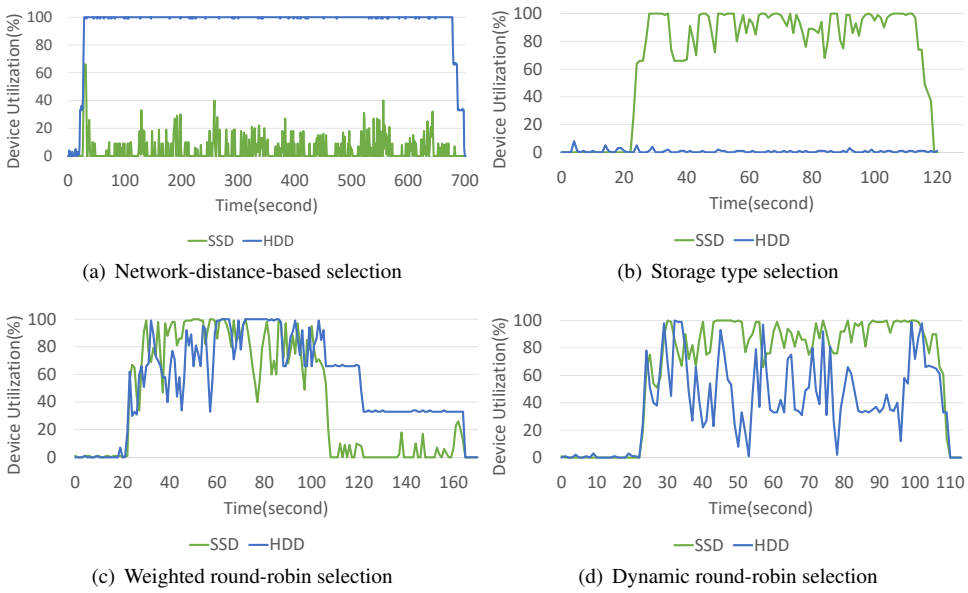
4.2.1 Execution time of benchmarks for the datasets

Figure 3 illustrates the execution time of benchmarks for each data retrieval policy. It is evident that network-distance-based selection provides the worst performance while storage type selection, WRR selection and DRR selection provide significantly higher performance than network-distance-based selection on all benchmarks.

This is because, when using the One_SSD storage policy, network-distance-based selection is more likely to read blocks stored in DISK despite SSD being able to handle more reads because only one replication is stored in the SSD. On the other hand, storage type selection, WRR selection, and DRR selection improve the overall performance by utilizing the SSD through a larger number of blocks read from SSD than from DISK (In this experiment, we established the weight of WRR selection to five because the performance of SSD and DISK used in this experiment is five times different.). In particular, because DRR selection utilizes both storages, it is expected to improve the throughput by 4 times the network-distance-based selection and 1.2 times the storage selection. The experimental results show that it performs best on all benchmarks and improves DFSIO by 6.4 times, Scan by 3.2 times, and Sort by 2.3 times compared to network-distance-based selection.

4.2.2 Evaluation details for benchmarks and data retrieval policies

For further analysis, we measured the device utilization of SSD and DISK with the linux iostat command which reports the device's input and output statistics for each benchmark dataset and data retrieval policy. We also calculated the average device utilization



(a) Network-distance-based selection

(b) Storage type selection

(c) Weighted round-robin selection

(d) Dynamic round-robin selection

Fig. 4. Storage utilization of each policy measured by DFSIO

by dividing the accumulated utilization values by a specific time period. (In DFSIO, the value is divided by the total execution time. In Scan and Sort, the value is divided by the time the read operations are performed because the writes are still in progress after the read operations are complete.)

DFSIO: Figure 4 compares the storage utilization for each data retrieval policy during DFSIO. As seen in Figure 4(a), in network-distance-based selection, the overall utilization of DISK is high while that of SSD is low. The average utilization of SSD is measured as 5% and that of DISK is measured as 97%. Next, Figure 4(b) shows that, in the storage type selection, only the SSD is used and the average utilization of SSD is measured as 90%, whereas DISK is not used. For WRR selection, as shown in Figure 4(c), the average utilization of the SSD is measured as 67% and that of DISK is measured as 69%. Finally, for DRR selection, as seen in Figure 4(d), the average utilization of SSD is measured as 90% and that of DISK is measured as 51%. From the results, we can made two key observations.

- **The utilization of fast storage significantly affects overall throughput.** The network-distance-based selection with the lowest SSD utilization shows the worst performance. WRR selection is better than network-distance-based selection because of its higher SSD utilization than, but it is slower than storage type selection and DRR selection because of its lower SSD utilization.

- **The policy that utilizes both storages shows better performance.** Storage type selection and DRR selection achieved high performance through high SSD utilization. However, as DRR selection monitors storage utilization and changes the weight dynamically, both storages show high utilization. This allows it to achieve higher performance than the storage type.

Additionally, we evaluated the data retrieval policies in various configures by using DFSIO. First, to observe the WRR selection's behavior according to weights, we mea-

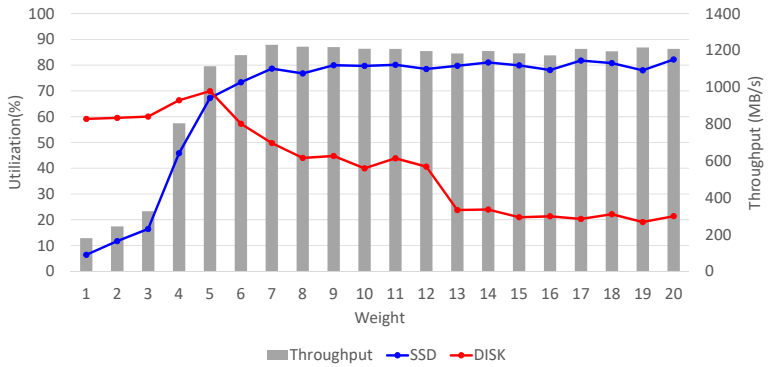


Fig. 5. Storage utilization and throughput of WRR selection by weight

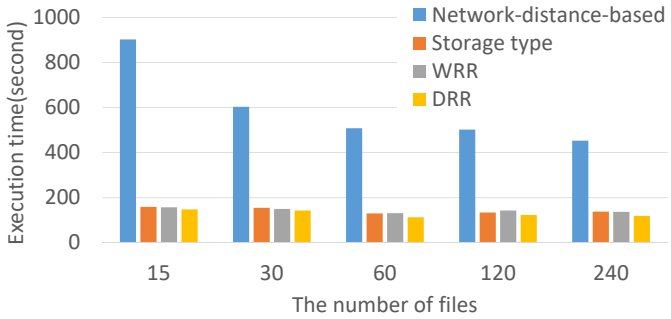


Fig. 6. DFSIO Performance by network distance

sured the utilization of SSD and DISK and the overall throughput by changing the weight from 1 to 20, as seen in Figure 5. As the weight increases, the utilization of SSD increases while that of DISK gradually decreases. As SSD performs considerably faster than DISK, the throughput of clusters increases with the increased utilization of SSD. Further, higher throughput is measured if the utilization of DISK is higher when SSD utilization is the same.

Specifically, the result shows the best performance when the weight is 7 (SSD utilization is 80%, DISK utilization is 50%.). When the weight is 20, the overall throughput decreased by about 2.5% because DISK utilization is reduced by about 30% while SSD utilization is similar. Because DISK is much slower than SSD, DISK utilization does not significantly affect the overall performance. However, as the weight increases, it is clear that the performance will be saturated with storage type selection, resulting in 10% performance degradation. The experiment shows that because WRR selection operates on static weights, it cannot cope with low utilization of a particular storage, making it difficult to obtain high storage utilization on both storages. In addition, it also shows that setting a higher threshold for slow storage has a greater adverse effect on performance by aggressively lowering the weight than setting a higher threshold for fast storage in DRR selection.

Second, we measured the performance of each data retrieval policy by increasing the number of map tasks that concurrently perform DFSIO reads, from 15 to 240. DFSIO benchmark adjusts the number of map tasks by changing the number of files. Figure 6 shows the execution time for reading 120 GB of data, depending on the number of files.

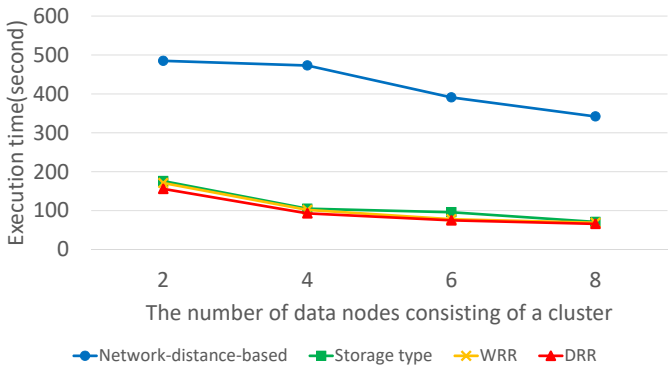


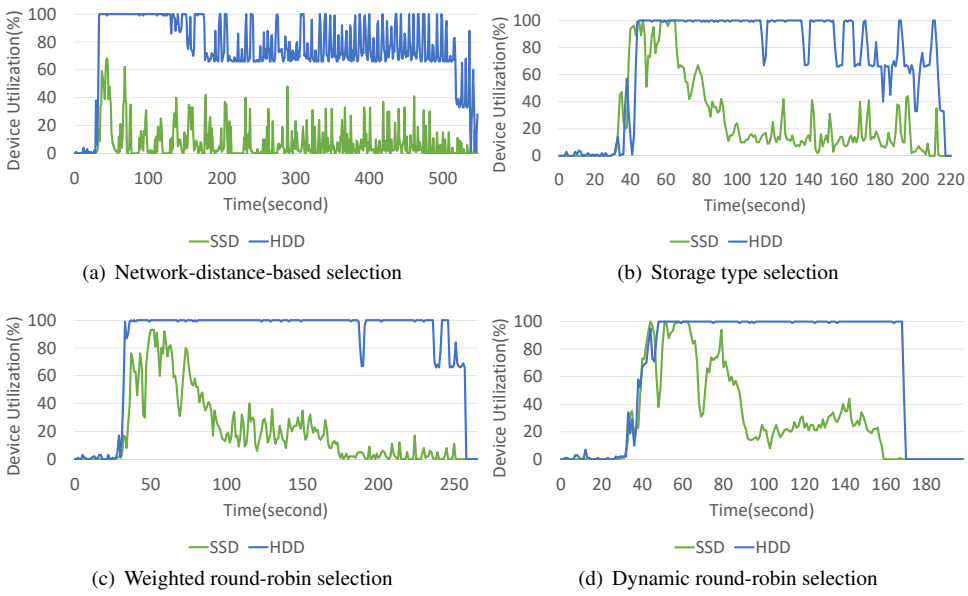
Fig. 7. DFSIO performance by the number of datanodes

The experiment result shows that when the number of map tasks is greater than or equal to 60, there is no performance difference according to the number of map tasks in all data retrieval policies. However, when the number of map tasks is less than or equal to 30, the performance of network-distance-based selection is degraded because not only the number of tasks processing reads is small, but also the tasks spend time waiting for IO. On the other hand, storage type selection, WRR selection, and DRR selection maintained their performance because they take advantage of fast IO performance and spend less time waiting for IO even with small map tasks.

Next, we measured the amount of rack-to-rack data movement across the network. We used a two-rack cluster consisting of a rack consisting of two nodes and a rack consisting of one node (we configured the other settings as default.). We measured the amount of data transferred between racks for each data retrieval policy, with and without network considerations. The result shows that when the network is not considered, about 33GB of data when using storage type selection, about 27GB of data when using WRR selection, and about 28GB of data when using DRR selection was transferred between the racks. On the other hand, when considering the network, there was no data transfer between the racks. Based on the result, we can see that our methods mitigate the saturation of the switch between racks.

Lastly, we measured the performance of each data retrieval policy by changing the number of datanodes consisting of the cluster to 2, 4, 6, and 8. In this experiment, all datanodes use one SSD and one DISK. In the case of a cluster consisting of nodes with different computing powers, the performance degradation occurs [19]. Therefore we only used DFSIO workload in this experiment. The result in Figure 7 shows that the performance of all data retrieval policies improves as the number of datanodes increases. Specifically, as the number of nodes increased from 2 to 8, network-distance-based selection improved by 41%, storage type selection by 147%, WRR selection by 147% and DRR selection by 136%.

Scan: As Scan reads a table and copies it to another table, both reads and writes are performed simultaneously. Writes are slower than reads because they create multiple replications while reads retrieve one of the replications. Moreover, writes lower fast storage utilization because fast storage has to wait for the replications to be written to the slow storages. In Scan, reads and writes are performed simultaneously in the front part, and the remaining writes are performed after reads are completed. As a result, SSD utilization is lowered after a certain time because the reads are almost completed and the remaining writes are performed.



(a) Network-distance-based selection (b) Storage type selection

(c) Weighted round-robin selection (d) Dynamic round-robin selection

Fig. 8. Storage utilization of each policy measured by scan

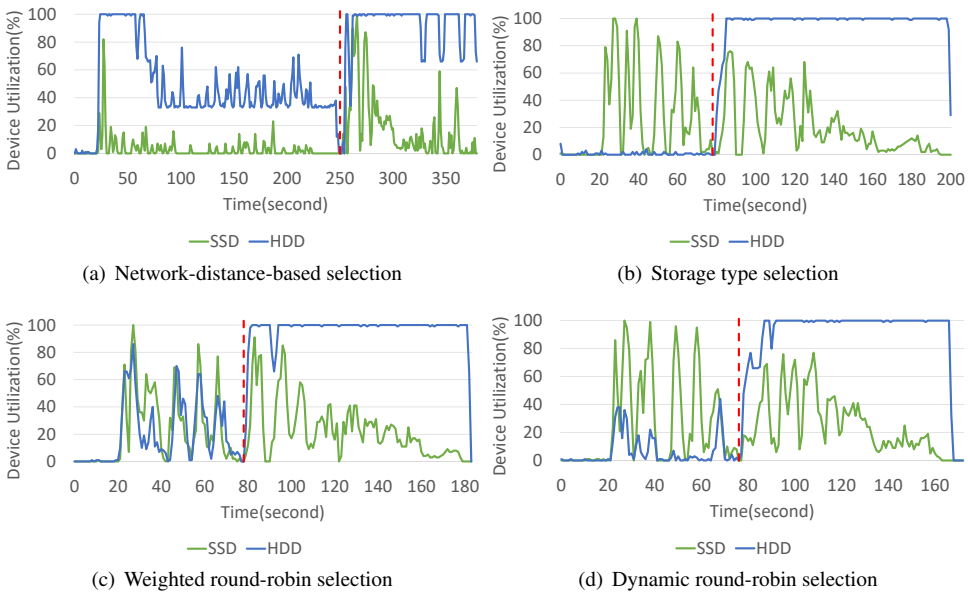
Figure 8 depicts the storage utilization of each data retrieval policy over time during Scan. During the read portion of the Scan, for network-distance-based selection, the average utilization of SSD is measured as 8.1% and the utilization of DISK is measured as 80%. This policy shows the slowest performance with lowest SSD utilization. For storage type selection, the average utilization of SSD is measured as 73%, whereas that of DISK is measured as 82%. Because all blocks are read from the SSD, fast performance and high SSD utilization is observed. Note that the utilization of DISK is also measured to be high because the blocks are written to both storages. For WRR selection, the average utilization of SSD is 64% and that of DISK is 99%. Finally, for DRR selection, the utilization of SSD is measured as 79% and that of DISK is measured as 84%. Both policies show high performance because they utilize both storages. DRR selection particularly shows the best performance by utilizing both storages most efficiently.

Sort: Sort reads blocks from HDFS, sorts them, and stores the final result in HDFS. As intermediate data is stored in temporary space, unlike Scan, the read and write parts can be distinguished. Furthermore, performance enhancement according to data retrieval policy is obtained from improved performance of reading part.

Figure 9 presents the storage utilization of HDFS cluster during Sort. The red dotted line in the figures distinguishes the read and write parts of the Sort. The left side of the line shows reading and the right side of the line shows writing.

As a result of the reading part of the Sort, the average utilization of SSD for network-distance-based selection is measured as 3% and that of DISK is measured as 52%. Next, for storage type selection, the average utilization of SSD is measured as 43%, whereas DISK is not used. For WRR selection, the average utilization of SSD is 37% and that of DISK is 31%. Finally, for DRR, the average utilization of SSD is measured as 43% and that of DISK is measured as 10%.

Network-distance-based selection shows the slowest performance as it cannot efficiently utilize fast storage in the reading part. On the other hand, storage type selection,



(a) Network-distance-based selection

(b) Storage type selection

(c) Weighted round-robin selection

(d) Dynamic round-robin selection

Fig. 9. Storage utilization of each policy measured by sort

WRR selection, and DRR selection provide fast readings and reduce the time to read. In the case of Sort operation, CPU usage is larger than the other benchmarks in our study, and as temporary data is stored separately, the storage utilization is lower than other tasks. Nevertheless, it is found that the improved I/O performance greatly influences the overall performance.

5. Related Work

Recently, several studies have been conducted to enhance the performance of Hadoop systems by increasing storage systems' performance. They can be classified into researches on homogeneous and on heterogeneous storage environments. Research on heterogeneous storage can also be categorized as one using fast storage as a cache and the other maximizing the throughput of heterogeneous storage system.

Researches on homogeneous storage environment: Researches of homogeneous storage environments improve performance by optimizing I/O of existing storage system [20] or by replacing slower storages with faster storages [21, 22, 23]. Themis [20] improves the performance of MapReduce as it optimizes existing storage systems by minimizing the number of I/O operations through elimination of task-level fault tolerance and dynamic and flexible memory management.

On the other hand, [21, 22, 23] improve the overall performance by replacing HDDs with SSDs. Specifically, [21] shows that replacing local storage device with HDD to SSD greatly improves the performance of I/O-intensive MapReduce job. [23] shows that the performance of Hadoop cluster composed of SSD is better though the throughput of SSD and of HDD is same. Lastly, [22] shows that leveraging SSD as temporary storage, especially for storing intermediate data, is the most cost-efficient.

Optimizing the existing storage system is effective in a homogeneous storage envi-

ronment although there is a limitation in solving the problem occurring in the heterogeneous storage environment described in Section 2. Though replacing slow storages with fast storages improves performance, our method is more cost-effective as it can achieve high performance while storing small amounts of data in fast storages.

Using fast storages as a cache: The studies of Hadoop with heterogeneous storage environment include researches using fast storages as a cache. HyCache [24] and mpCache [25] add SSDs to the distributed file system and provide methods for using SSDs as caching areas for existing storage systems. Triple-H [26] runs in a heterogeneous storage environment using RAM_DISK and SSD-based buffer caches.

These studies improve performance by avoiding access to slower storage devices through caching methods. However, they are less efficient than our method in terms of space utilization because of the redundant data in the storage tiers as well as to the block replication of HDFS. Moreover, while caching methods utilize only fast storage, and there is overhead of managing the caching area, our approach utilizes both fast and slow storage, resulting in more opportunities for performance improvements.

Optimizing the heterogeneous storage environment: There are several researches focusing on improving performance by optimizing storages in heterogeneous storage environments. [23, 27, 28] describes the performance of heterogeneous storage environments with SSDs added to clusters of HDDs. In [23], using default configuration, the bandwidth of added SSD is not fully utilized because less data is stored on the SSD than on the HDDs. In [27] and [28], the performance of a cluster is measured by changing the ratio of data stored on the HDDs and SSDs. Further, [27] suggests that the system that stores more data in the SSD is less sensitive to parameters, and [28] indicates decreased energy consumption.

In the above studies, the performance of cluster is improved when more data is stored in the SSDs. However, because the price per capacity of an SSD is higher than that of an HDD, storing more data on the SSD is inefficient in terms of cost. On the other hand, our method optimizes read through new data retrieval policy for providing high performance while storing a small amount of data in SSD.

In other studies, fast storage and slow storage are classified into tiers, and tier-based data access schemes are proposed [29, 30]. [29] improves the performance of SQL query processing engine called SQL-on-Hadoop running on HDFS using heterogeneous storage environment and utilizes the SSD in HDFS by allowing the SQL-on-Hadoop system to read data on SSDs(it is similar to the storage type selection in our study). The method used in [30] divides similar storage devices into tiers and proposes block placement and retrieval policies to improve storage utilization. Among the methods proposed in this study, the hybrid policy stores and reads blocks considering the performance of the network and the storage(it is similar to the WRR selection of our study).

Both methods improve application performance by utilizing fast storages. However, while [29] utilizes only fast storages, our proposed method can deliver better performance because it uses both types of storages. In addition, our method modifies the structure of the distributed file system, providing the advantage of improving performance regardless of the type of application.

In case of [30], it effectively utilizes all devices in a heterogeneous storage environment. However, performance degradation may occur when the weights are set incorrectly. Our proposed method dynamically checks and changes the weight, thus working more flexibly and effectively.

6. Conclusion

In this paper, we proposed a new data retrieval method for HDFS with heterogeneous storages to optimize the read performance. Our method efficiently utilized both storages by considering not only network distance but also the performance of storage. In addition, it dynamically rebalances the number of read requests according to the performance of each storage. We evaluated the performance of our method using three well-known workloads. All benchmark results demonstrate that our method is able to deliver 2.2-6 times better performance than the existing HDFS. In addition, our method shows improved performance compared to other methods considering the type of storage device.

REFERENCES

1. "Apache Hadoop," http://apache.hadoop.org.
2. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.
3. D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, Vol. 11, no. 2007, 2007, p. 21.
4. T. W. Wlodarczyk, Y. Han, and C. Rong, "Performance analysis of hadoop for query processing," in *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*. IEEE, 2011, pp. 507–513.
5. F. Pan, Y. Yue, J. Xiong, and D. Hao, "I/o characterization of big data workloads in data centers," in *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, 2014, pp. 85–97.
6. T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 327–338.
7. J. Kwak, E. Hwang, T.-k. Yoo, B. Nam, and Y.-r. Choi, "In-memory caching orchestration for hadoop," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 94–97.
8. D. Lee, C. Min, and Y. I. Eom, "Effective flash-based ssd caching for high performance home cloud server," *IEEE Transactions on Consumer Electronics*, Vol. 61, no. 2, 2015, pp. 215–221.
9. A. Wang and C. McCabe, "In-memory caching in hdfs: Lower latency, same great taste," *Hadoop Summit*, Vol. 2014, 2014.
10. T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang, "hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems," *Proceedings of the VLDB Endowment*, Vol. 5, no. 10, 2012, pp. 1076–1087.
11. Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 2011, pp. 227–236.
12. H. Payer, M. Sanvido, Z. Z. Bandic, and C. M. Kirsch, "Combo drive: Optimizing cost and performance in a heterogeneous storage device," in *First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Vol. 1, no. 1, 2009, pp. 1–8.

13. "Hadoop Archival Storage, SSD & Memory," https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html, 2014.
14. T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.
15. L.-Y. Ho, J.-J. Wu, and P. Liu, "Optimal algorithms for cross-rack communication optimization in mapreduce framework," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 420–427.
16. J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, Vol. 51, no. 1, 2008, pp. 107–113.
17. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, Vol. 2, no. 2, 2009, pp. 1626–1629.
18. S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.
19. C. B. VishnuVardhan and P. K. Baruah, "Improving the performance of heterogeneous hadoop cluster," in *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*. IEEE, 2016, pp. 225–230.
20. A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat *et al.*, "Themis: an i/o-efficient mapreduce," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 13.
21. S.-H. Kang, D.-H. Koo, W.-H. Kang, and S.-W. Lee, "A case for flash memory ssd in hadoop applications," *International Journal of Control and Automation*, Vol. 6, no. 1, 2013, pp. 201–210.
22. S. Moon, J. Lee, and Y. S. Kee, "Introducing ssds to the hadoop mapreduce framework," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 272–279.
23. K. Kambatla and Y. Chen, "The truth about mapreduce performance on ssds," in *28th Large Installation System Administration Conference, LISA*, 2014, pp. 109–118.
24. D. Zhao and I. Raicu, "Hycache: A user-level caching middleware for distributed file systems," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1997–2006.
25. B. Wang, J. Jiang, and G. Yang, "mpcache: Accelerating mapreduce with hybrid storage system on many-core clusters," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 220–233.
26. N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. K. Panda, "Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 101–110.
27. D. Wu, W. Luo, W. Xie, X. Ji, J. He, and D. Wu, "Understanding the impacts of solid-state storage on the hadoop performance," in *Advanced Cloud and Big Data (CBD), 2013 International Conference on*. IEEE, 2013, pp. 125–130.
28. I. Polato, D. Barbosa, A. Hindle, and F. Kon, "Hybrid hdfs: decreasing energy consumption and speeding up hadoop using ssds," *PeerJ PrePrints*, 2015.
29. M. Kim, M. Shin, and S. Park, "Take me to ssd: a hybrid block-selection method on hdfs based on storage type," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 965–971.

30. K. Krish, A. Anwar, and A. R. Butt, "hats: A heterogeneity-aware tiered storage for hadoop," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 502–511.