Hindawi Publishing Corporation
Scientific Programming
Volume 2016, Article ID 1808396, 12 pages
http://dx.doi.org/10.1155/2016/1808396

Research Article

MHDFS: A Memory-Based Hadoop Framework for Large Data Storage

Aibo Song,[1] Maoxian Zhao,[2] Yingying Xue,[1] and Junzhou Luo[1]

[1]*School of Computer Science and Engineering, Southeast University, Nanjing 211189, China*
[2]*College of Mathematics and Systems Science, Shandong University of Science and Technology, Qingdao 266590, China*

Correspondence should be addressed to Aibo Song; absong@seu.edu.cn

Received 22 February 2016; Accepted 17 April 2016

Academic Editor: Laurence T. Yang

Hadoop distributed file system (HDFS) is undoubtedly the most popular framework for storing and processing large amount of data on clusters of machines. Although a plethora of practices have been proposed for improving the processing efficiency and resource utilization, traditional HDFS still suffers from the overhead of disk-based low throughput and I/O rate. In this paper, we attempt to address this problem by developing a memory-based Hadoop framework called MHDFS. Firstly, a strategy for allocating and configuring reasonable memory resources for MHDFS is designed and RAMFS is utilized to develop the framework. Then, we propose a new method to handle the data replacement to disk when memory resource is excessively occupied. An algorithm for estimating and updating the replacement is designed based on the metrics of file heat. Finally, substantial experiments are conducted which demonstrate the effectiveness of MHDFS and its advantage against conventional HDFS.

1. Introduction

Recent years have seen an astounding growth of enterprises having urgent requirements to collect, store, and analyze enormous data for analyzing important information. These requirements continuously step over a wide spectrum of application domains, ranging from e-business and search engine to social networking [1, 2].

With the significant increment of data consumption in various applications, from the level of GB to PB, there is an urgent need for such platforms with superior ability to store and process this exploding information. As the most used commercial computing model based on the Internet, cloud computing is currently employed as the primary solution and can provide reliable, scalable, and flexible computing abilities as well as theoretically unlimited storage resources [3]. With necessary components available, such as networks, computational nodes, and storage media, large-scale distributed clouding platforms can be conveniently and quickly built for various data-intensive applications.

Conventional could computing systems mainly rely on distributed disk storages and employ the management subsystems to integrate enormous machines and build the effective computing platform. Typical cloud computing platforms include Apache Hadoop [4], Microsoft Azure [5], and Google MapReduce [6, 7], among which the open-source Hadoop system gains particular interests in practice. Generally, these platforms all provide high throughput in data access and data storing for clients by effectively managing distributed computer resources, which are proved to be appropriate to store and process large amount of data in real-world applications.

However, as the evolvement of modern computer hardware, the capacity of various memory media is continuously increasing, with the decline of their prices. Considering this situation, researchers and engineers have focused their effort on the employment and management of memory resources to step across the bottleneck of I/O rate in conventional distributed cloud computing systems.

Current memory-based distributed file systems are mostly designed for the real-time applications, including HANA [8] and Spark [9, 10] as the typical representatives. Their goal is to speed up the process of data writing and reading from different perspectives. HANA is the platform

developed by SAP Inc. and is widely used in fast data processing by employing a specifically designed framework of memory-based distributed storing. Even though HANA platform is widely adopted, it is mainly designed to handle the structured data instead of semistructured data. Since there are quite a number of applications based on semistructured or even unstructured data, HANA is not suitable for them. Meanwhile, HANA is usually memory-intensive due to its high occupation of memory space. Spark is a recently involved cloud computing system based on memory that employs the basic concept of resilient distributed datasets (RDD) to effectively manage the data transformation and storage. Spark is a fast and general-purpose cluster computing system which provides high-level APIs and an optimized engine that supports general job executions. It is highly dependent on the third-party component Mesos [11], whose authority is to manage and isolate various memory resources. This causes the unreliability and absence of customization for Spark. Additionally, Spark is based on its self-developed data model called RDD (resilient distributed datasets) that differs significantly from the conventional HDFS data model. Thus, the problem of compatibility is also the crucial part for its limited employment on existing applications. Developers have to design their data structures and architectures from scratch.

Based on the above analysis, this paper mainly focuses on improving the current infrastructure of HDFS. We extended the storage media of HDFS from solely disk-based to memory-based, with disk storage as addition, and designed a proper strategy to preferentially store data into the memory. Meanwhile, we proposed an algorithm of memory data replacement for handling the overload of limited memory space and effectively increase the average I/O rate as well as overall throughput of the system.

The main difference of our MHDFS framework compared to Spark and HANA includes two aspects. Firstly, we developed the framework based on native Hadoop, which provides consistent APIs and data structures. So existing applications based on Hadoop can conveniently migrate to MHDFS with little changes. Secondly, we designed the memory data replacement module as the secondary storage. So MHDFS can automatically handle the situation when memory space is nearly occupied.

The remainder of this paper is organized as follows. Section 2 presents a brief description of our proposed MHDFS system and demonstrates the key architecture. Section 3 gives the strategies of allocating and deploying distributed memory resources. Section 4 introduces fault tolerance design based on memory data replacement. Then, in Section 5, experiments on different size of data are conducted to evaluate the effectiveness of our proposed model. Finally, we present related studies in Section 6 and conclude our work in Section 7.

2. Architecture

Based on the previous analysis, we now present MHDFS, a memory-based Hadoop framework for large data storage. MHDFS is an incremental system of the native HDFS. Other than the normal modules, it includes two other modules, the memory resource deployment module and the memory data replacement module. The architecture of MHDFS is shown in Figure 1. In MHDFS, name node accepts clients' requests and forwards them to assigned data nodes for specific writing or reading jobs. Data nodes are based on both memory and disk spaces, where memory space is selected as the preferred storing media against disk through the help of memory resource deployment module. Meanwhile, for handling the problem of limited memory space, memory data replacement module is involved as the middleware and swaps files into disk when necessary. And it is proved to be useful to the robustness of the whole system.

2.1. Memory Resource Deployment Module. This module is designed for configuring and allocating available memory space for deploying MHDFS. Based on the analysis on the historical physical memory usage of each data node, we estimate the size of available memory space, that is, the remaining memory space not occupied by ordinary executions of a machine. Then, the estimated memory space will be mapped to a file path and used for deploying MHDFS. Data will be preferentially written into the memory space instead of the conventional HDFS disk blocks.

2.2. Memory Data Replacement Module. This module is designed for swapping data between memory and disk when RAM space is occupied by executing jobs. In this module, a reasonable upper bound is set to start the data replacement process. In addition, a submodel based on file accessed heat is proposed to find the best swapping data block. Intuitively, the file heat is related to the access frequency of a file; that is, files being accessed frequently during recent period will generally have higher heats. Details will be presented in Section 3. In this submodel, heat of each file will be evaluated and files in the memory will be ranked according to their heat. Then, files with low heat will be swapped out to the local disk so as to ensure the available memory space for subsequent job executions.

3. Strategy of Memory Resource Deployment

3.1. Allocation of Memory Resources. In order to allocate available memory resources for deploying HDFS, we utilized the records of physical memory on each data node and estimate the reasonable memory spaces. The key strategy is stated as follows.

We denote the total memory capacity of each data node as M_T, which is related to the configuration of the physical machine. To calculate the available memory space for deploying MHDFS, we monitor the usage information of data node within a time duration of ΔT and record the maximal RAM usage as M_h. Since the memory usage is fluctuating continuously with enormous job executions, we give a relax to the local computing usage and set the maximal usage as $\delta \times M_h$. The value of δ is determined by the running status of different clusters, but generally we could set it as 120% to satisfy the local job executing requirements. Finally,

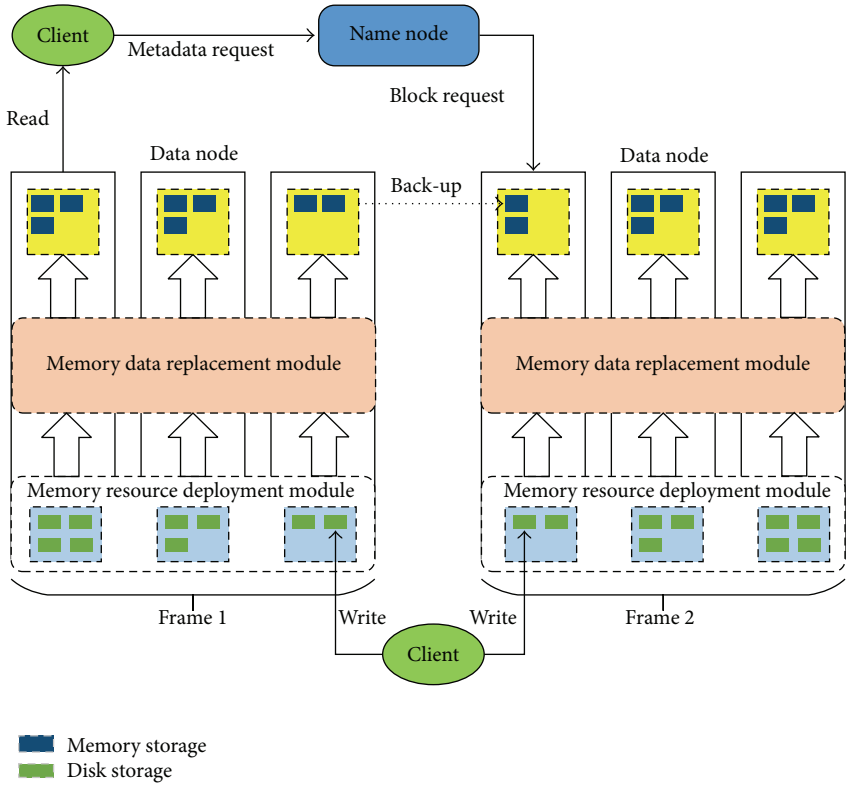

Figure 1: The architecture of MHDFS.

the remaining memory space is allocated as the resources for MHDFS, which is called the maximum available memory storage space (M_a):

$$M_a = M_T - M_h * 120\%. \quad (1)$$

If the calculated M_a is less than or equal to zero, the historical maximal memory usage on the node is convinced to exceed 83.3%, which further indicates the heavy load of physical memory. In this case, due to the lack of enough available memory spaces, this data node is decided to be unsuitable for allocating memory resources for MHDFS. Intuitively, when the total size of written data exceeds M_a, MHDFS will prevent subsequent data writing into the memory of this node.

Since HDFS requires the storage paths mapping to specific directories, we have to map the available memory space to a file path. For the sake of convenience, *Ramfs* is used as the developing tool [12]. It is a RAM-based file system integrated into the Linux kernel and works at the layer of virtual file system (VFS).

According to the maximal available memory space M_a proposed above, we use *Ramfs* to mount the memory space to a predefined file path, so that data could be written into the memory accordingly. *Ramfs* takes the advantage of dynamically allocating and utilizing memory resources. To be more specific, if the amount of written data is less then M_a, say M_d ($M_d < M_a$), *Ramfs* only occupies the memory space of size M_d while the remaining are still preserved for ordinary job executions. With the size of written data increasing continuously, *Ramfs* enlarges the total occupied memory space to meet system requirements. The size of the remaining memory space could be marked as

$$M_p = M_T - M_d, \quad (2)$$

where M_T is the total memory space of the data node and M_d is the actual used memory space by MHDFS.

Ramfs ensures M_p always being positive along with the process of data reading and writing, which improves the effectiveness of memory utilization of the physical machine. And the mechanism of dynamically allocating memory resources further reduces its inferior effect to the capability of local job executions. Practically, we can set parameters in *hdfs-site.xml* file and configure the file path mapped from memory to another storage path of data block.

3.2. Management of Memory Resource. Conventionally, data is usually stored in the unit called data block in HDFS. Traditional data storing process in HDFS could be summarized as the following steps: (1) client requests for writing data, (2) name node takes charge of assigning data nodes for writing and recording the information of assigned nodes, and (3) specific data nodes store the writing data to its self-decided file paths.

The data storage structure of the data node could be demonstrated as in Figure 2. Each FSVolume represents a data storage path and the FSVolumeSet manages all FSVolumes which is configured in the directory of ${*dfs.data.dir*}. Generally, there is more than one data storage path ${*dfs.data.dir*} on each data node; HDFS employs

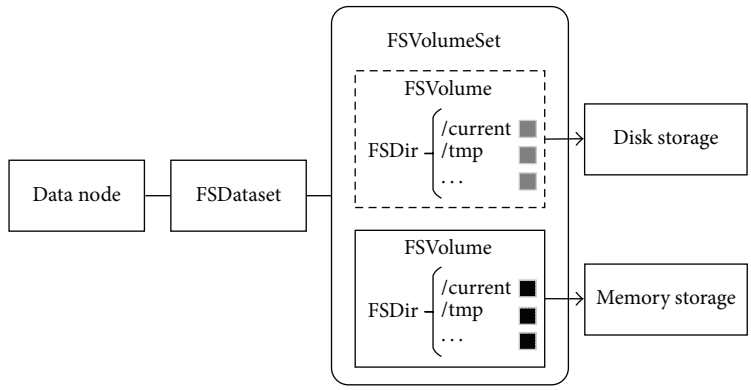

Figure 2: The data storage structure of MHDFS.

the strategy of round robin to select different paths so that data at each path could be balanced. The general idea of round-robin path selection is to use variable *curVolume* to record current selected storage path increasing it by one after a block of data is written each time. Then *curVolume* points to the next storage path in the volumes array. When each data block is written, storage path is changed and finally, after *curVolume* reaches the end of volumes array, it will be redirected to the head.

To ensure data written into the memory preferentially, we rewrite the policy of selecting storage path in HDFS. We assign different file paths with different priorities, sort them ordered by priority, and then store the file paths into the *volumes* array of the data node. Considering Figure 2, for example, with the memory-based storage path in array volumes[0] and the disk-based storage path in array volumes[1], we check the paths in the array from start when data is written. MHDFS will continuously check the paths subsequently until a satisfied one is found out.

4. Strategy of Memory Data Replacement

4.1. File Replacement Threshold Setting. Obviously, memory resource of physical machines is limited compared with disk spaces. When the remaining memory space of the data node is below a specific threshold, the process of replacing data stored in memory into disk will be triggered. In MHDFS, we design a strategy of threshold measuring and replacement trigger.

The threshold setting is specific to the user and application, whereas it is usually acceptable to set as the 10% of the maximal available storage space of the data node, which means when there is less than 10% of maximal available space remaining, MHDFS should start the process of replacement.

Based on the assumption that the maximum available memory on a data node DN is M_a and the already used memory space on DN is M_u, the trigger condition could be set that the remaining memory space is less than 10% of M_a. The data replacement trigger threshold on the DN node is denoted as T_{start}, and its formula could be wrote as follows:

$$T_{\text{start}} = (M_a - M_u) - 10\% * M_a. \quad (3)$$

After the writing operation has finished on DN, the detection of the threshold T_{start} will be conducted.

Replacing process is constantly conducted with low heat file swapped out from the memory until (4) is satisfied, which indicates that the available memory space on DN is sufficient again:

$$T_{\text{stop\_s}} \leq \frac{M_a - M_u}{M_a}. \quad (4)$$

Generally, the threshold for stopping the replacement process is very specific to different applications and configurations of clusters, which is decided by the user himself. In this paper, according to the environment of our experimenting cluster, the default stopping threshold is set as $T_{\text{stop\_s}} = 40\%$; that is, when the available memory on DN exceeds $40\% * M_a$ after the involvement of memory data replacement, the replacement process will be suspended and data will be still kept retaining in memory to ensure the efficiency of subsequent data accesses.

4.2. Measurement of File Heat. In order to make the file replacement effective, a strategy has to be proposed to identify the files to be replaced. Intuitively, if a file is accessed frequently by the applications, it is more preferable to retain in memory, while the less frequently visited files could be swapped into the disk to save memory space.

A frequency-based heat parameter is proposed in MHDFS to handle the problem stated above. The heat parameter will be stored in the metainformation of each file in the memory, which records the access frequency and access volume. Basically, there are three rules:

(1) Files with more frequent access have higher heat.

(2) Files with larger amount of data access have higher heat.

(3) Files with shorter survival time have higher heat.

We designed the heat metric based on the theory of time series which can evaluate the accessing status of each file. The frequency of the file being accessed is recorded within time interval. The heat is updated after the file is accessed. The

Input:
Operation: Type of Operate of Write and Read
RDatNum: Number of byte of requesting to access the file, or Number off byte of writing into file
⟨*Fseq, Heat, Life, AccessT, InVisitT*⟩: File List, Current Heat, Life Time, Access Time, Access Interval
Output: Result: Result of Update
(1) **Function** UpdateHeatDegree
(2) Seq = Fseq;
(3) //Get current time;
(4) Time = getTime();
(5) **if** (operation == read) **then**
(6) factor = (Time − Access$T$)/Invisit$T$ + (Time − Access$T$);
(7) New_heat = (factor ∗ RDatNum + (1 − factor) ∗ heat)/(Time − Life);
(8) Invisit$T$ = Time − Access$T$;
(9) Access$T$ = Time;
(10) Heat = New_Heat;
(11) **return** Success, ⟨Fseq, Heat, Life, Access$T$, InVisit$T$⟩
(12) **end if**
(13) **if** (operation == write) **then**
(14) Access$T$ = Time;
(15) Invisit$T$ = Time − Access$T$;
(16) **if** (operation == create) **then**
(17) Fseq = new Fseq();
(18) New_heat = 0;
(19) Life = Time;
(20) Heat = New_heat;
(21) **return** Success, ⟨Fseq, Heat, Life, Access$T$, InVisit$T$⟩
(22) **end if**
(23) **if** (operation == add) **then**
(24) **return** Success, ⟨Fseq, Heat, Life, Access$T$, ReSet$T$⟩
(25) **end if**
(26) **end if**
(27) **if** (operation == delete) **then**
(28) delete(Fseq);
(29) **return** null
(30) **end if**
(31) **return** Fail, ⟨Fseq, Heat, Life, Access$T$, ReSet$T$⟩
(32) **End**

Algorithm 1: Memory file heat update.

shorter the interval is, the greater impact the access has, and vice versa.

We denote $H_i(f)$ as the head of the file on data nodes in the memory storage and propose the following equation:

$$H_0(f) = 0$$
$$H_i(f) = \frac{\beta * S_i + (1-\beta) * H_{i-1}(f)}{T_i}, \tag{5}$$

where $\beta = \Delta t_i/(\Delta t_i + \Delta t_{i-1})$.

Apparently, the equation is consistent with the above three principles. When a file is accessed frequently, $H_{i-1}(f)$ gives more weight to $H_i(f)$, and if bytes of accessed data S_i are large or file survival time T_i is small, $H_i(f)$ will also generate a larger result. β is an adjustment factor based on the time series, which mainly works for adjusting the relationship between the current heat and historical heat. A larger adjustment factor results in a greater impact of data accessed amount to the file heat, while a smaller one gives more weight to the historical heat value.

4.3. File Heat Update. As can be seen from Section 4.2, file heat updating process is specific to different data operations. In this paper, we categorize various data operations into three groups: (1) writing operation, including creating and adding, (2) reading operation, and (3) deleting operation. Considering the three kinds of operations, the updating strategy is demonstrated as pseudocodes in Algorithm 1.

The updating of file heat information includes adding, updating, and deleting. Initially, the heat of a file is zero which clearly means that newly created file with no reading access never has a positive heat. And if a file has never been read since creation, the associated heat value will remain unchanged. This indicates that reading operation is closely

related to the increase of file heat which is confirmed to our common sense. The updating process can be summarized as follows.

(1) For reading operations, accessed data size and accessing time will be recorded, and our algorithm will update the file heat as well as the access interval parameter according to (5).

(2) For writing operations, we consider them as two cases: (a) creating operation: a file sequence number and file feat are initialized, with both creating time and access time as current system time and access interval as zero; (b) adding operation: adding time is recorded and access interval is updated, without updating the file heat. This is consistent with the above analysis that only reading operations can directly impact the altering of file heat.

(3) For deleting operations, file heat record will be deleted based on the defined file sequence number.

4.4. File Replacement. Based on the measurements of threshold setting and file heat updating strategy stated above, we designed the module of memory file replacement. The process can be summarized in Figure 3.

For the sake of simplicity, we set the starting threshold of replacement as 10% of total available memory space and the stopping threshold as 40% of total available memory space. The process could be summarized as the following steps:

(1) Check the remaining memory, and if remaining memory space is less than 10% of M_a, replacement process shall be started.

(2) Sort all files in memory by their file heats in descending order.

(3) Generally, each file may contain a number of data blocks. Replacement process starts from the data block with the longest survival time until all the data blocks are moved out.

(4) When all data blocks in a file have been replaced into the disk, the corresponding file heat is deleted and replacement process carries on to replace the data block in the file with secondary lowest file heat.

(5) Suspend replacement process when the remaining space reaches 40% of M_a again.

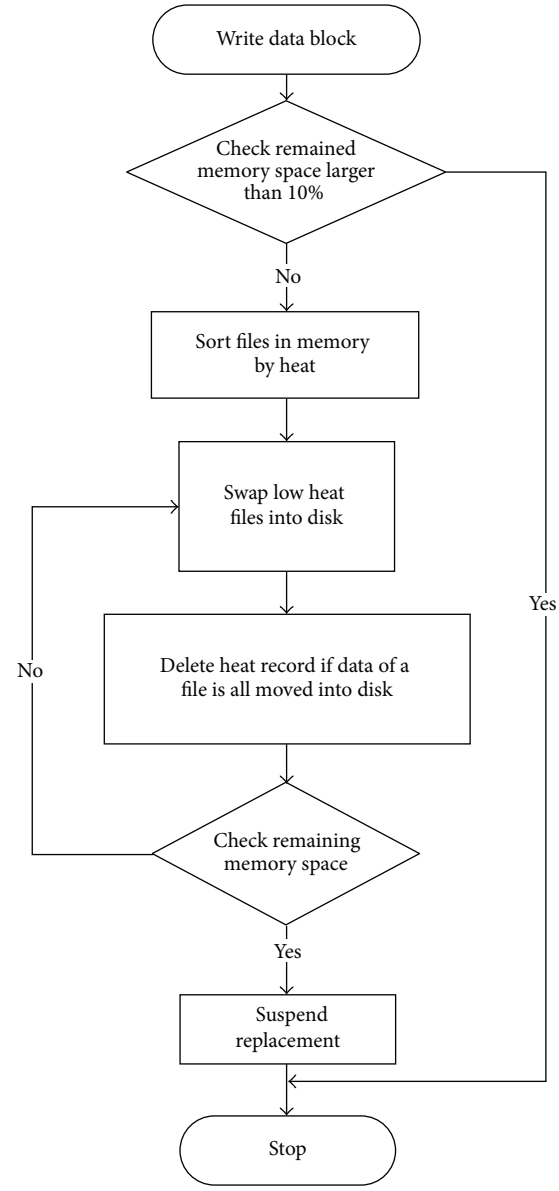

Figure 3: Process of memory file replacement.

5. Experiment

In order to showcase the performance of our proposed MHDFS model, we did comparison experiments on stimulated datasets. Reading and writing performance is evaluated on different size of data, say, 2 GB and 8 GB. *Ramfs* is utilized as the developing tool for the mapping of memory space.

5.1. Experimental Setup. Experimental cluster is set up with nine nodes: one for the Hadoop name node and resource manager, and the others for the data nodes and data managers. Each node is a physical machine with dual 3.2 GHz Xeon EM64 CPU, 4 GB MEM, and 80 GB HDD, connected via Gigabit Ethernet. We use Red Hat Enterprise Linux Server Release 6.2 as the host operating system and the version of Hadoop distribution is 2.6.0.

For conducting the experiments, we map the memory of name node to the file storage path using *Ramfs* and configure two file paths (one for the disk storage and the other for the memory storage) in the Hadoop configuration file. After recompilation, HDFS can identify both the file paths and dynamically assign priority to them.

5.2. Performance Comparison on Single-Node Cluster

5.2.1. I/O Test with Small Datasets (Less Than 2 GB). Data reading and writing operations are conducted on MHDFS

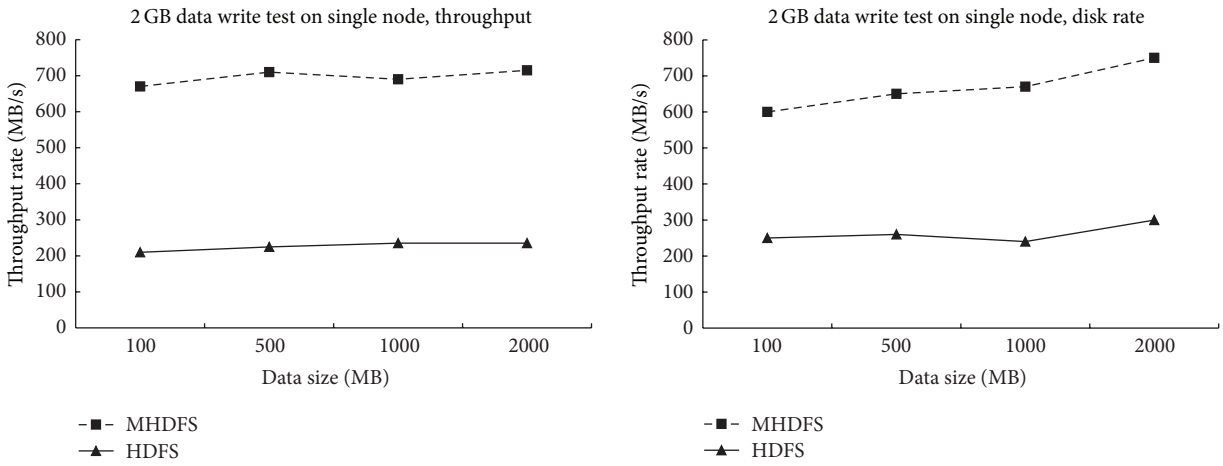

Figure 4: HDFS/MHDFS 2 GB data write test on single-node cluster.

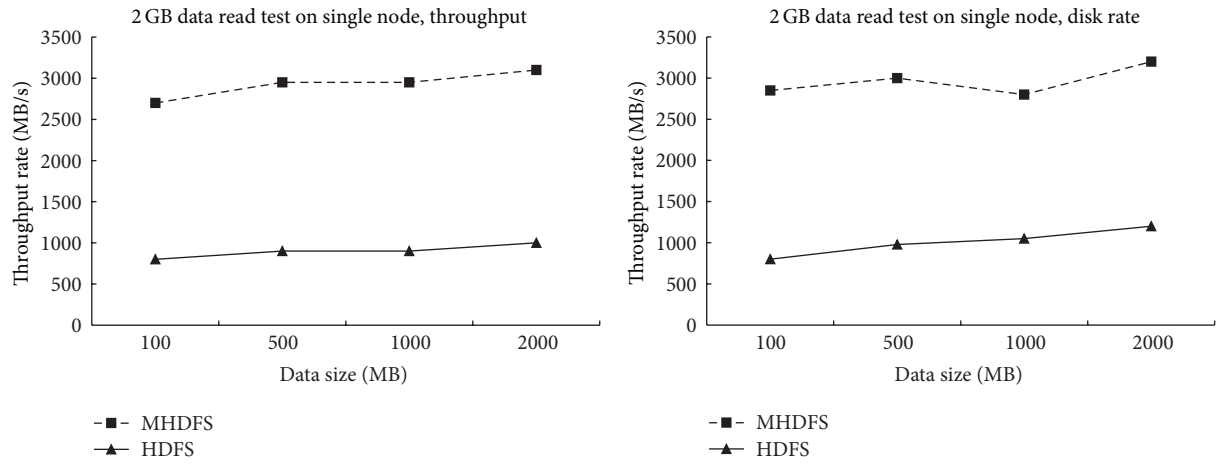

Figure 5: HDFS/MHDFS 2 GB data read test on single-node cluster.

and HDFS based on a single-node cluster and performance is recorded to evaluate the effectiveness of our proposed model. Data size increases continuously from 500 MB to 2 GB. The evaluating results are shown in Figures 4 and 5.

Generally, the throughput and average disk write rate of both models increase with the increment of data size. The throughput of HDFS is floating around 300 MB/s, whereas MHDFS reaches up to 700–800 MB/s, almost three times faster than HDFS. The result shows that MHDFS can significantly improve the performance of writing data compared with conventional HDFS.

Considering the reading performance shown in Figure 5, the throughput of HDFS is floating around 900 MB/s, while MHDFS is up to 3000 MB/s, which is also three times of HDFS. Internally, MHDFS is able to respond to data reading requests in time, since the data read by MHDFS is preferentially stored in memory, rather than disk. Besides, the relatively good performance of HDFS is highly related to the use of SSD, which further indicates the significant improvement of MHDFS against HDFS in various conditions.

5.2.2. I/O Test with Large Datasets (from 2 GB to 8 GB). In this subsection, we evaluate our proposed MHDFS model with larger size of datasets. Intuitively, when the written data exceeds a certain size, memory space is occupied, and subsequent writing will be directed to file paths in the disk.

Figure 6 shows the results of performance evaluation with data size varying from 2 GB to 8 GB. Generally, after the data size reaches over 2 GB, the throughout and average I/O rate both have declined due to the lack of memory space. In detail, the value of throughput and average I/O rate of writing and reading have declined from 800 Mps to 600 Mps and 3000 Mps to 2000 Mps, respectively. Even though both MHDFS and HDFS get declined performance with data increases, MHDFS still achieves higher results against HDFS. This is consistent with our expectation and can be explained as follows. When memory space is insufficient, data replacement operation will be started and data can only be stored into disk during the replacing period. This may lead to the declined performance of MHDFS for a while. After the complete of replacement, data is still written into the available

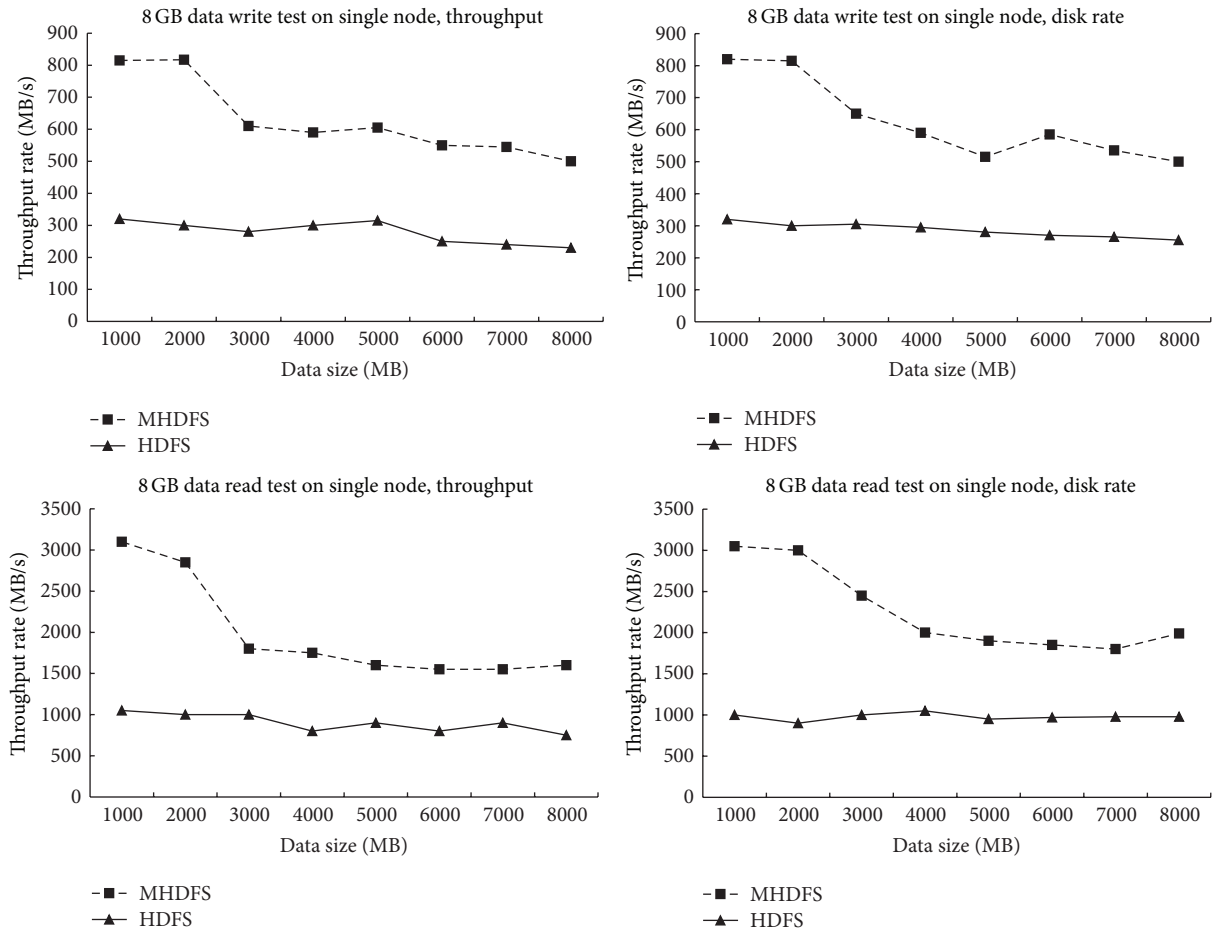

Figure 6: HDFS/MHDFS 8 GB data read/write test on single-node cluster.

memory space and reaches a relatively stable performance with subsequent writing requests.

5.3. Performance Comparison on Multinode Cluster. For comparing the performance of MHDFS and conventional HDFS in real-world distributed environment, we conducted experiments on the multinode cluster. We configure different memory resources for MHDFS on each node to test the triggering and swapping strategy. The configuration is shown in Table 1.

Generally, nine nodes are selected for the experiment: one for master node and the others for data nodes. The total memory allocated for MHDFS is 10100 MB, where master node is allocated 2 GB for ensuring the execution of the whole cluster. Memory allocation on each data node is according to the historical maximal memory usage individually. We use the Linux command *top* to periodically monitor the ordinary memory usage of each machine and estimate the maximal one. And for testing the effectiveness of our memory data replacement strategy, the experiment data size is designed to exceed the available total memory space.

5.3.1. I/O Test with Small Datasets (Less Than 10 GB). We evaluate the performance of MHDFS and HDFS on small datasets, say, less than 10 GB, when memory space for the whole cluster is not fully occupied. In this case, MHDFS will not trigger the process of data swapping and disk storing. The amount of data is increasing from 1 GB to 10 GB, every 2 GB as the step. The results are shown in Figures 7 and 8.

Table 1: Memory storage resources on each node.

Node	Historical maximal memory usage	Allocated memory size (MB)
Master	/	2000
$N1$	59.7%	1250
$N2$	62.6%	1000
$N3$	65.0%	900
$N4$	61.9%	1050
$N5$	68.1%	750
$N6$	58.9%	1200
$N7$	62.7%	1000
$N8$	64.1%	950

As can be seen from Figure 7, the writing throughput and average writing rate of MHDFS are 45 Mbps and 50 Mbps, respectively, which only has a slight improvement over HDFS

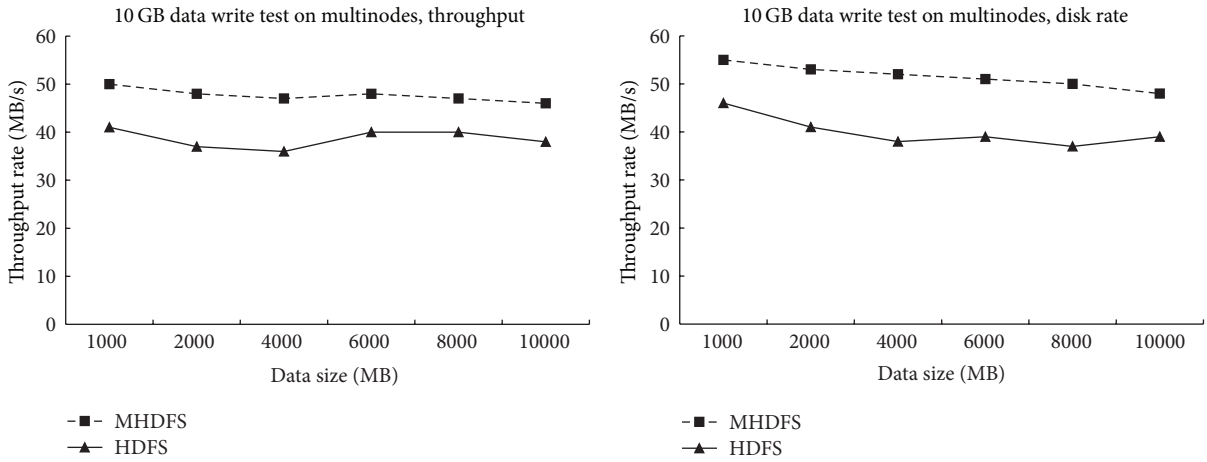

Figure 7: HDFS/MHDFS 10 GB data write test on multinode cluster.

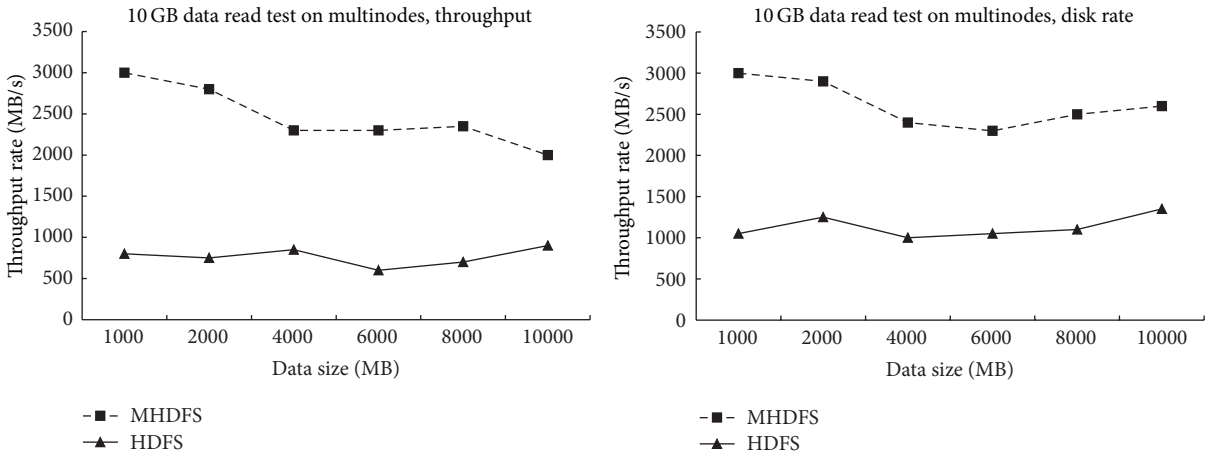

Figure 8: HDFS/MHDFS 10 GB data read test on multinode cluster.

compared with results in single-node environment. This is due to the fact that writing data into the cluster requires simultaneously copy writing; that is, for each writing, three copies of the data should be stored in different nodes. With 100 Mbps as the bandwidth of each node, the crucial cost of writing is actually the network overhead which cannot be improved by MHDFS. In fact, MHDFS is designed to leverage the memory space for handling disk writing overhead and thus can provide little help to other factors like network transferring.

Figure 8 presents the results of reading test. The reading throughput and average reading rate of MHDFS reaches 2600 Mbps and 2650 Mbps, respectively, which outperforms two and half times compared to HDFS. Undoubtedly, this is consistent with our theory that when reading data from cluster, data as well as its copies are mostly stored in memory and this provides very fast reading operations. Interestingly, however, with the increase of data size, the reading performance gets a few declines. This is because, for the single-node environment, all data can be read from memory while, for multinode clusters, some data must be transferred via network from other data nodes. The additional cost of network transfer leads to the slightly decline when large amount of data is requested.

5.3.2. I/O Test with Large Datasets (from 10 GB to 25 GB). In this subsection, we evaluate our proposed MHDFS model with larger size of datasets in the environment of multinodes cluster. In this situation, MHDFS will employ the strategy of data replacement. Figures 9 and 10 demonstrate the writing and reading results.

From Figure 9, it is clear that when writing data exceeds 10 GB, that is, the allocated memory space for MHDFS, the writing performance declines due to the involvement of data replacement, even if it still outperforms HDFS with 5% to 15%.

Remarkably, different with the cases in single-node cluster, the adjective impact of data replacement is much slighter due to the fact that, in single-node situation, memory space is totally prohibited for writing when data replacement is conducted, while, for multinode cluster, replacing on one node will not affect another node's writing process; that is, data replacement seldom occurs on all nodes in the whole cluster.

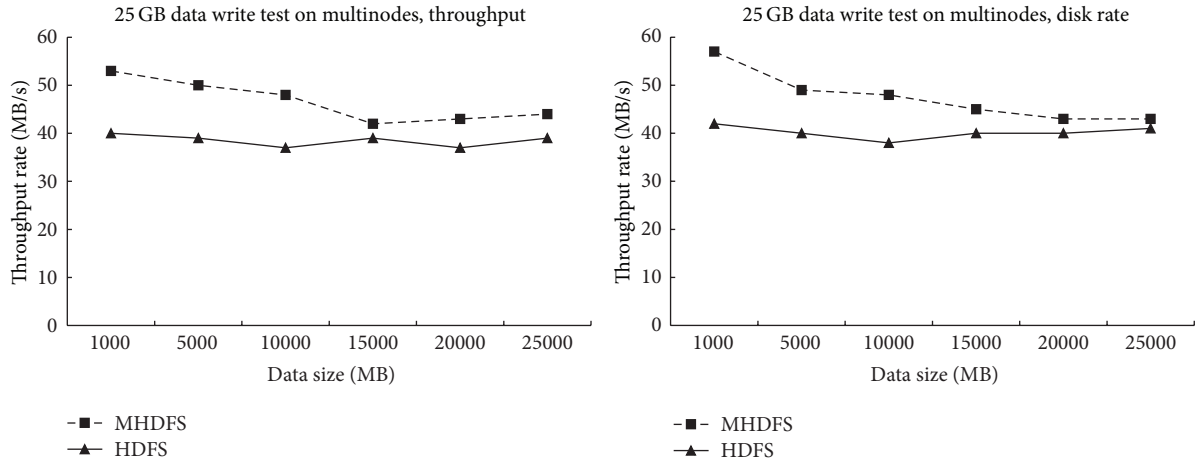

Figure 9: HDFS/MHDFS 25 GB data write test on multinode cluster.

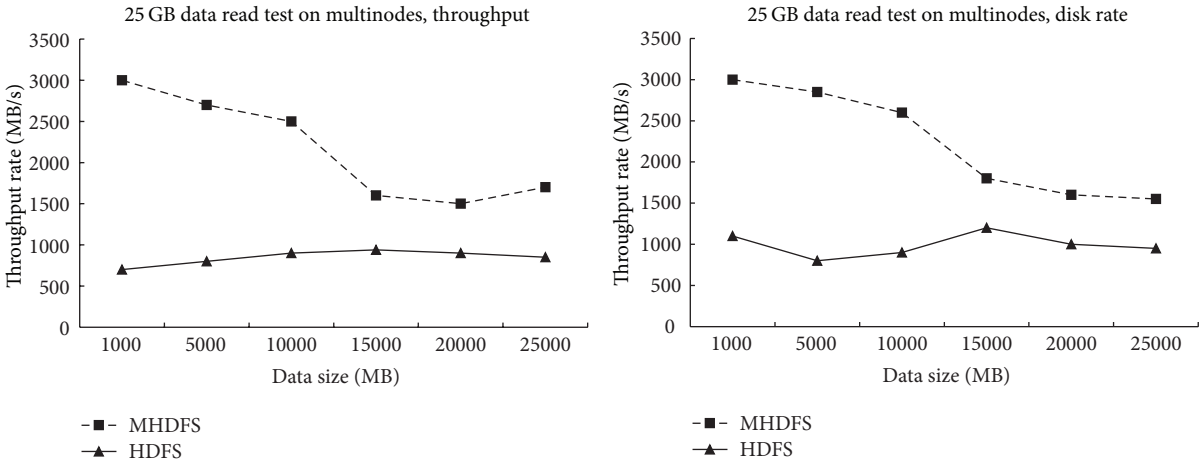

Figure 10: HDFS/MHDFS 25 GB data read test on multinode cluster.

The reading test result is undoubtedly consistent with other evaluations according to Figure 10, and, due to the factor of data locality, reading performance in multinode cluster declines faster than that in single-node situation. In detail, for single-node clusters, data is only read from local machine while, for multinode clusters, data shall occasionally be obtained from other machines via network.

5.4. A Real Case Benchmark. In this section, we benchmark our MHDFS framework compared with native HDFS based on the word-count case, that is, to evaluate the average performance when processing real-world applications. We evaluate the total executing time of the word-count cases as MapReduce jobs under different input sizes, and both the single-node and multinode clusters are examined (for the single-node case, we only use the name node as a stand-alone cluster). Figures 11 and 12 present the results of MHDFS and HDFS on various data sizes, respectively.

It is obvious that, on both single-node and multinode clusters, MHDFS achieves significant performance against traditional HDFS. This proves the improvement of our memory-based Hadoop framework that can accelerate the execution of MapReduce jobs with the advantage of storing the input data and intermediate results into the memory space. Generally, the MapReduce process of word count can

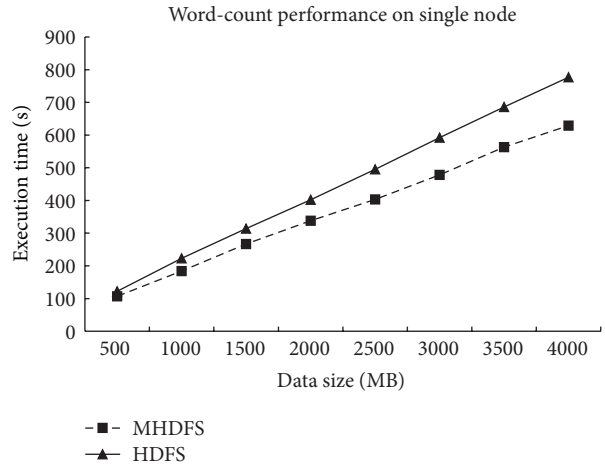

Figure 11: HDFS/MHDFS word-count test on single-node cluster.

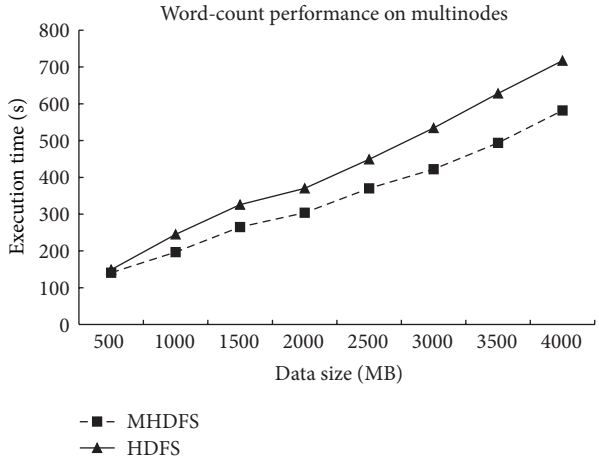

Figure 12: HDFS/MHDFS word-count test on multinode cluster.

generate double the sizes of its input; thus, for smaller size of input, MHDFS is quite close to HDFS, while, for larger size of input, MHDFS can notably benefit from the memory-based intermediate data storage.

The result on multinode cluster also indicates the overall better performance of MHDFS with the increasing of data size from 500 MB to 4 GB. Interestingly, due to the inferior capacities of data nodes, both MHDFS performance and HDFS performance are worse than that on single-node cluster from 500 MB to 1.5 GB. Since data processing is dispatched to various machines in a multinode cluster, the capability of machine is the major factor for smaller size of data, whereas, for larger size of data, it is the ability of parallel processing that counts, and thus HDFS and MHDFS achieve better performance than single-node situation, between which MHDFS costs less executing time owing to the employment of memory space.

6. Related Work

Despite its popularity and enormous applications, Hadoop has been the object of severe criticism, mainly due to its disk-based storage model [13]. The input and output data is required to be stored into the disk. Meanwhile, the inner mechanism for fault tolerance in Hadoop requires the intermediate results to be flushed into the hardware disk, which undoubtedly deteriorates the overall performance of processing large amount of data. In the literature, the existing improvement for Hadoop framework can be concluded as two categories: one aims at developing a wholly in-memory system that stores all the intermediate results into the memory, while the other focuses on employing a number of optimizations to make each usage of data more efficient [14, 15].

Spark [9, 10] and HANA [8] are typical examples belonging to the first category. They are totally in-memory systems that utilize the high throughput of memory media. Spark is a fast and general-purpose cluster computing system which provides high-level APIs and an optimized engine that supports general job executions. Resilient distributed datasets (RDD) are the key component for Spark to effectively manage the data transformation and storage in memory. HANA is the platform developed by SAP Inc. and is widely used in fast data processing by employing a specifically designed framework of memory-based distributed storing. The main shortcoming is that they are built upon another model completely different from native Hadoop which makes business migration inconvenient for companies. Meanwhile, the absence of disk utilization, to some extent, causes the systems' intensive dependence on limited memory spaces and lack of robust and fault tolerance. However, for those Hadoop-enhanced systems, these problems do not exist.

The second category, however, handles the problem from a different perspective. Hu et al. [15] analyzed the multiple-job parallelization problem and proposed the multiple-job optimization scheduler to improve the hardware utilization by paralleling different kinds of jobs. Condie et al. [16] added pipelined job interconnections to reduce the communication cost and lower the reading overhead from disk hardware. Since these researches mostly focus on designing probable scheduling strategies or intermediate result storing strategies, it is not necessary to enumerate them all, and although they can improve the performance to some extent, these works do not touch upon the idea of deploying Hadoop based on memory spaces.

Our research is distinct from the existing literatures mainly in the following aspects. First, we build MHDFS solely upon native Hadoop system without any modification to the crucial interfaces. This makes MHDFS compatible with currently existing systems and applications and makes migration much easier. Second, we design the strategy of memory data replacement to handle the situation when memory is fully occupied. Disk storages are still effectively utilized to improve the overall performance of our system.

7. Conclusion

As the basic foundation of Hadoop ecosystem, disk-dependent HDFS has been widely employed in various applications. Considering its shortcomings, mainly in low performance of the disk-based storage, we designed and implemented a memory-based Hadoop distributed file system named MHDFS. We proposed the framework of allocating reasonable memory space for our model and employed *Ramfs* tool to develop the system. Besides, the strategy of memory data replacement is studied for handling the insufficiency of memory space. Experiment results on different size of data indicate the effectiveness of our model. Our future study will focus on the further improvement of our proposed model and study the performance in more complicated situations.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grants nos. 61370207, 61572128,

and 61320106007, China National High Technology Research and Development Program (2013AA013503), SGCC Science and Technology Program "Research on the Key Technologies of the Distributed Data Management in the Dispatching and Control System Platform of Physical Distribution and Logical Integration," Collaborative Innovation Center of Wireless Communications Technology, Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu Provincial Key Laboratory of Network and Information Security (BM2003201), and Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant no. 93K-9.

References

[1] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI, San Francisco, Calif, USA, 2004.

[2] D. Howe, M. Costanzo, P. Fey et al., "Big data: the future of biocuration," *Nature*, vol. 455, no. 7209, pp. 47–50, 2008.

[3] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *Proceedings of the IEEE Grid Computing Environments Workshop (GCE '08)*, pp. 1–10, Austin, Tex, USA, November 2008.

[4] Apache Hadoop [EB/OL], http://hadoop.apache.org.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pp. 59–72, Lisbon, Portugal, March 2007.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[8] SAP, HANA[EB/OL], http://www.saphana.com/.

[9] SPARK, [EB/OL], http://www.sparkada.com/.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*, USENIX Association, 2010.

[11] B. Hindman, A. Konwinski, M. Zaharia et al., "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Boston, Mass, USA, April 2011.

[12] M. Zhao and R. J. Figueiredo, "Experimental study of virtual machine migration in support of reservation of cluster resources," in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC '07)*, ACM, November 2007.

[13] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in MapReduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[14] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: increased performance for in-memory Hadoop jobs," *Proceedings of the VLDB Endowment* , vol. 5, no. 12, pp. 1736–1747, 2012.

[15] W. Hu, C. Tian, X. Liu et al., "Multiple-job optimization in mapreduce for heterogeneous workloads," in *Proceedings of the 6th International Conference on Semantics Knowledge and Grid (SKG '10)*, pp. 135–140, IEEE, Beijing, China, November 2010.

[16] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, vol. 10, p. 20, 2010.