PAPER

NDCouplingHDFS: A Coupling Architecture for a Power-proportional Hadoop Distributed File System*

Hieu Hanh LE[†], **Satoshi HIKIDA**[†], ***Nonmembers***, *and* **Haruo YOKOTA**[†], ***Member***

SUMMARY Energy-aware distributed file systems are increasingly moving toward power-proportional designs. However, current works have not considered the cost of updating data sets that were modified in a low-power mode, where a subset of nodes were powered off. In detail, when the system moves to a high-power mode, it must internally replicate the updated data to the reactivated nodes. Effectively reflecting the updated data is vital in making a distributed file system, such as the Hadoop Distributed File System (HDFS), power proportional. In the current HDFS design, when the system changes power mode, the block replication process is ineffectively restrained by a single NameNode because of access congestion of the metadata information of blocks. This paper presents a novel architecture, a NameNode and DataNode Coupling Hadoop Distributed File System (NDCouplingHDFS), which effectively reflects the updated blocks when the system goes into high-power mode. This is achieved by coupling metadata management and data management at each node to efficiently localize the range of blocks maintained by the metadata. Experiments using actual machines show that NDCouplingHDFS is able to significantly reduce the execution time required to move updated blocks by 46% relative to the normal HDFS. Moreover, NDCouplingHDFS is capable of increasing the throughput of the system supporting MapReduce by applying an index in metadata management.

key words: *power-proportionality, HDFS, metadata management*

1. Introduction

Energy-aware commercial off-the-shelf (COTS)-based distributed file systems for cloud applications are increasingly moving toward power-proportional designs, as system configuration can be changed on demand. In this design, the distributed file systems are able to provide IO performance that is proportional to the number of active nodes used in the system. Specifically, the system is designed to operate in multiple gears and each gear contains a different number of active nodes. The higher gears have more active nodes and hence consume more energy. The lowest gear has a covering set group of nodes that are always active with a sufficient amount of data to serve the requests.

Multi-gear operation is made possible through a number of recent works that focus on power-proportional data placement layouts [2–4]. In general, the data from the powered-off nodes are replicated to the activate nodes such that at each operation mode, the amount of processing data is equally shared among activated nodes. As a result, in each gear, the file system is capable of providing power proportionality in IO performance. However, those works have not yet dealt with the reflecting of a modified data set that is changed in low-gear mode when several nodes are powered off. Here, the modification of the data set includes the behaviors of appending, updating or newly creating of files that belong to the data set. In a low gear, instead of the inactive nodes, the currently active nodes that maintain their data's replicas should update the modified data. When the system moves to a high gear, to share the load equally among all active nodes, it is necessary to let the reactivated nodes catch up with the data set modifications. Specifically, the system must transparently replicate the updated data to the reactivated nodes.

In addition to normal operations, the process of reflecting the updated data set increases the costs of metadata management and data transference inside the system. In a low gear, the system has to keep a log record specifying the location information of updated data. And when changing to a higher gear, it must identify the replicated data from log records, access its metadata, transfer the data to original nodes and update the metadata of these data for later reference. Carrying out this process effectively is vital in realizing power proportionality for a distributed file system, such as the Hadoop Distributed File System (HDFS) [5], which is already widely used as a distributed file system for effective big data processing in the cloud. In the current HDFS architecture, reflecting updated files is ineffectively restrained at the NameNode because of access congestion in the metadata information of blocks.

This paper presents a novel architecture called the NameNode and DataNode Coupling HDFS (NDCouplingHDFS), which is designed to effectively reflect updated data in the power-proportional HDFS. NDCouplingHDFS couples metadata management and data management at all the nodes of the system to localize the range of blocks maintained by the metadata. Through this idea, the process is effectively distributed to multiple nodes as the load is shared among the nodes and each node can focus on its own work because all the necessary information is located locally.

[†]The authors are with the Department of Computer Science, Tokyo Institute of Technology, Ookayama, Meguro-ku, Tokyo 152-8552 Japan

[*]This paper is an extended version based on our short paper proposed in DASFAA 2013 [1]

DOI: 10.1587/transinf.E0.D.1

Additionally, taking advantage of coupling architecture, we further optimize the protocol of replicating blocks to improve the performance of the updated block reflection process. In the current HDFS, the block transferring process is performed in sequential manner in which the system has to open a new network connection for every single block. Hence, the performance of the process is degraded when a number of blocks becomes large as the cost relating to the network part increases. In order to decrease this cost, we introduce several batch methods, which are able to process multiple blocks per network connection. The suggested batch methods are effectively suited with NDCouplingHDFS, in which the processed blocks are efficiently localized at multiple nodes of the system.

Moreover, to raise the efficiency of reflecting updated data, it is preferable to eliminate the bottleneck of metadata management at the single NameNode in a normal HDFS by using distributed metadata management. Taking the locality of the file system into consideration, we suggest two approaches of distributed metadata management based on a tree structure, namely static directory partitioning and the parallel B-tree-based index method. In the first approach, we divide the namespace of the system among all the nodes, as each node will maintain a subpart of the directory hierarchy. In the second approach, we apply the parallel index technique, called Fat-Btree [6], which is used in current database management, to manage the metadata of the file system.

Furthermore, although the normal HDFS is designed to run on commodity hardware, the NameNode that centrally maintains all the metadata of the systems should be well equipped with high-end CPU, large memory and so on. However, consisting such high-end machines is not required in NDCouplingHDFS because the metadata is distributed to all nodes and each node only manages the metadata of local data.

In addition, recent works on power-proportional systems have not fully provided power proportionality, as the power consumption of the NameNode was not counted when the power consumption of the system was considered. However, we believe that by coupling the metadata management and data management at all nodes, NDCouplingHDFS can provide perfect power proportionality to the systems.

Our main contributions are the following.

- NDCouplingHDFS is proposed to solve the problem of reflecting updated data sets when the power-proportional file system shifts from a low gear to a higher gear.
- NDCouplingHDFS improves the IO throughput of the metadata operation of the HDFS by implementing distributed metadata management with an index technique.
- NDCouplingHDFS is compatible with MapReduce frameworks, which are currently supported by the HDFS.
- NDCouplingHDFS eliminates the single point of failure existing in the normal HDFS by utilizing metadata replication at multiple nodes.
- An empirical experiment to comprehensively evaluate the idea of coupling metadata management and data management, and batch block processing in NDCouplingHDFS is performed on actual machines. The empirical experimental results show that NDCouplingHDFS is able to significantly reduce the execution time to transfer updated blocks by 46% relative to a normal HDFS.

This paper is an extended version of [1] as it includes the important parts of reporting the experiment to provide the capability of NDCouplingHDFS to support the MapReduce framework. Furthermore, all other explanations of NDCouplingHDFS and the discussion of the experimental results are described in more detail.

The remainder of this paper is organized as follows. Related work is introduced in Sect. 2 and the background of the HDFS architecture and the assumptions made in this paper are described in Sect. 3. Section 4 describes the architecture of NDCouplingHDFS, the optimized techniques for effective updated data reflecting process. Section 5 presents a performance evaluation of our proposals. Conclusions and future work are discussed in Sect. 6.

2. Related Work

Rabbit [2] is the first work that aims to provide power proportionality to an HDFS by focusing on read performance. Rabbit uses the equal-work data layout policy using data replication. The primary replica is stored evenly on primary nodes. The remaining replicas are stored on additional and increasingly large subsets of nodes. A node in the subsets is fixedly ordered and stores the number of blocks that inversely relates to its order, to guarantee that the system can distribute the workload equally to all activated nodes. However, Rabbit still has not supported write workload, hence not yet considered the cost of reflecting updated data in a low gear.

Designed as a power-proportional distributed storage system for data centers, Sierra [3] is a replicated object store that is able to support both the write and read workload in multiple-gear operation. This method guarantees the write availability in a low gear by taking advantage of the idea of Write Off-loading [7], which is originally motivated by the goal of saving power through spinning down unnecessary disks. It allows write requests on spun-down disks to be temporally redirected to other active disks in the file system. As a result, this technique lengthens the spin-down durations, thereby achieving additional power saving. Although not aiming to provide power proportionality, the idea could be considered as a solution for multi-gear file systems dealing with updated data when the system operates in a low gear. Nonetheless, despite the fact that Sierra is able to deal with write requests in a low gear, it is still not optimized for the efficiency of reflecting updated data when the system moves to higher gear. Similar to the current HDFS, the current architecture of Sierra exploits the centralized metadata management which is widely considered as a bottleneck of the system [8].

In previous work, we have taken into consideration the cost of updated data reflection relating to the size of moving data in a power-proportional HDFS [9]. As the size of moving data is small, the reflection process could be shortened.

There are also other efforts on reducing the total power consumption of storage systems.

Weddle et al. [10] have proposed PARAID, a skewed pattern for replicating and striping data blocks to the disks inside the system. This enable adaption to the system load by varying the number of activating disks, where the disks are organized into a number of groups and identified subsets of groups are the gears. PARAID only focuses on RAID unit and lack the reliability when adapts to distributed environment.

Kim et al. [4] suggest a fractional replication method to achieve a balance between the power consumption and performance of a system. In their proposal, the data-placement layout is inspired by PARAID [10] in performing fractional replication. Their work considers the problem of identifying a suitable time to gear down and save power.

Kaushik et al. [11] proposed an energy-conserving multiple-zone approach for HDFS that utilized life-cycle information for the data. They divided storage clusters of the HDFS into a Hot Zone and a Cold Zone. The frequently accessed data were placed in the Hot Zone, where all datanodes are active and consuming power. Less frequently accessed data were placed in the Cold Zone, allowing datanodes to be inactive. The power consumption in the Cold Zone is therefore lower than in the Hot Zone. Via this mechanism, some power saving for the HDFS is achieved. However, this does not provide full power proportionality to the HDFS.

3. Background

This section describes the concepts of the HDFS architecture and the assumptions made in this paper.

3.1 HDFS Architecture

An HDFS cluster has two types of nodes, a single NameNode and a number of DataNodes; the major modules are shown in Fig. 1. The centralized metadata management (Centralized MDM) on the NameNode maintains the directory tree of the file system and all the metadata from all files and directories in the system. In an HDFS, internally, a file is split into one or more blocks and these blocks are stored on DataNodes. The process of Data Placement decides which DataNodes will store the file's blocks by considering the locality and availability of the system. The process of Block Mapping keeps the mapping information between the blocks and the DataNodes on which the blocks are located.

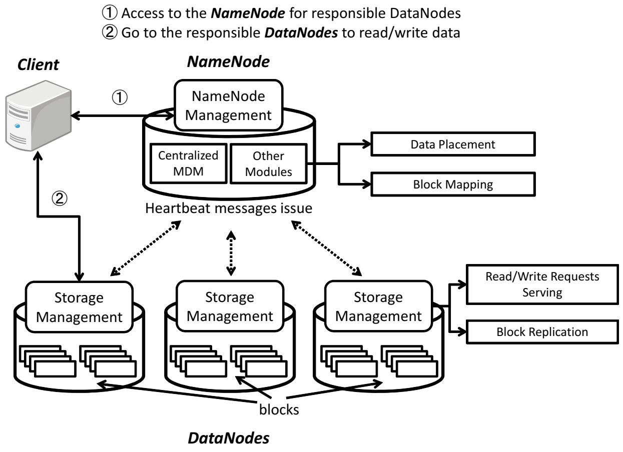


Fig. 1 HDFS architecture and data flow

The Storage Management at each DataNode stores and retrieves blocks to serve read/write requests from clients or block replication requests from other DataNodes. Additionally, the contacts between NameNode and DataNodes are kept by heartbeat messages that are periodically issued after each *heartbeat_interval*. Using this mechanism, the NameNode can maintain the status of DataNodes and instruct the block replication process if needed. In detail, the transferred blocks coupled with the destination node information are sent to the block transfer queue of the responsible Storage Management at DataNodes. This queue is then drained by the Storage Management after each *heartbeat_interval*.

When the client wants to access certain files, it first needs to connect to the NameNode to obtain metadata information and then directly opens connections to responsible DataNodes to read or write the blocks of that file.

3.2 Assumptions and Conditions

In our proposal, we employed the following assumptions and conditions.

1. Data layout policy: The scope of this paper is limited to the metadata management and the cost of reflecting updated data at power-proportional file systems. As we focus on the locality to decrease

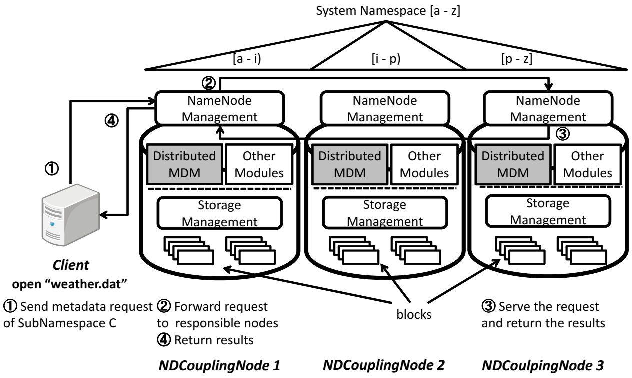


Fig. 2 A NameNode and DataNode Coupling HDFS architecture and data flow

the cost, we assume that every file's blocks are initially stored at one node and then are replicated to other nodes according to an appropriate power-proportional method.

2. Replication: This paper suggests coupling metadata management and data management methods. When data are replicated at other nodes, their metadata are also replicated at the same node. In a low gear, both the metadata and the blocks from inactive nodes are replicated at other, active nodes.
3. Failure: We suppose that all nodes in the system operate without failure. However, because all the metadata and data are replicated, when a node fails, all its requests can be redirected to other nodes that maintain its replicas. Consequently, it is derived that the single point of failure does not exist in our proposal. This important characteristic differentiates our proposal from the normal HDFS for which the single NameNode that maintains whole namespace of the systems becomes the single point of failure. When the NameNode fails, the files in the systems are unaccessible. The Byzantine failures are not considered in this paper's scope.

4. NDCouplingHDFS

This section at first depicts the architecture and a data flow of our system. And then, two methods for distributed metadata management are described. Finally, we present the system's behavior in reflecting updated data.

4.1 Architecture and Data Flow of NDCouplingHDFS

The architecture of NameNode and DataNode Coupling HDFS (NDCouplingHDFS) are shown in Figure 2. In this paper, because we focus on the locality of metadata management to improve the efficiency of reflecting the updated data when the system shifts to a higher gear, we decide to apply the equivalent coupling as all the NDCouplingNodes contain both NameNode Management and Storage Management. In future work, the looser or tighter coupling in which several nodes manage the metadata of one node's data or one node maintains the metadata of several nodes' data could be considerable.

In NDCouplingHDFS, the NameNode Management includes the distributed metadata management (Distributed MDM) and other modules such as the Data Placement and Block Mapping, as in a normal HDFS. The important difference from a default HDFS is that the namespace of the file system is divided among all the nodes in the cluster while take locality into consideration. The local Distributed MDM and the Block Mapping only manage the metadata for files and blocks that are locally located. For example, if a file *weather.dat* is located on NDCouplingNode 3, then both of its metadata and blocks will also be managed by NDCouplingNode 3. The Storage Management at NDCouplingNode is the Storage Management at DataNode in a normal HDFS.

Because NDCouplingHDFS is aimed to be applied to the system which provides power-proportionality through data replication, the mechanisms to manage the number of replicas of blocks are achieved at the following two circumstances. Firstly, like in the normal HDFS, whenever receiving a new block from client for a writing request or from other nodes for a block replication request, the node's Storage Management sends a notification message to the local NameNode Management to inform the local Block Mapping about the new location of the block. After the local Block Mapping finish updating such location information for the block, the local NameNode Management further broadcasts the notification messages to all other nodes in the system. The NameNode Managements at other nodes in which the blocks are replicating on are subjected to inform the corresponding Block Mappings to update this new location information. The NameNode Managements at nodes that are not storing any replicas of the block then simply drop the message. Through this mechanism, the block's location information is always kept updated. Secondly, the number of replicas of blocks in the system is monitored through the periodical heartbeat messages between the NameNode Management and the Storage Management at each node and among the NameNode Managements at all nodes of the system. The communication protocols of the NameNode Managements among all nodes of the system are newly implemented into NDCouplingHDFS. Further techniques to optimize the network communication protocols in this processes could be considered in future work.

Here, relating to the power consumption of the system, it is well recognized that at each configuration which requires the similar number of nodes containing the physical data, NDCouplingHDFS generally consumes less total energy than the normal HDFS. At each configuration, NDCouplingHDFS uses fewer active nodes than the normal HDFS because the separate NameNode to maintain the metadata is not required.

Next, the data flow for the client interacting with NDCouplingHDFS is explained using Fig. 2. At first, when the client wants to access the file system (open *weather.dat*), as opposed to the case of the normal HDFS, the system randomly connects to a node of the cluster, in this case NDCouplingNode 1. Subsequently, at this connected node, according to the new Distributed MDM, the system forwards the request to the corresponding NDCouplingNode (NDCouplingNode 3) that contains the metadata of this file. Then, at NDCouplingNode 3, the system looks for the file's metadata and sends back to the client the result through NDCouplingNode 1. Finally, based on this result, the Client opens connections to the responsible NDCouplingNodes to retrieve or store all the blocks of the file.

From the above description, the important point of NDCouplingHDFS is how to identify the responsible NDCouplingNode that contains the metadata for the accessed files under Distributed MDM. We explain the two approaches that we use in the following.

4.2 Distributed Metadata Management

Here, we describe two distributed metadata management (Distributed MDM) methods that take the locality of the file system into consideration.

4.2.1 Static Directory Partitioning Method

In this paper, we first try the static directory partitioning (SDP) method in distributing the namespace to multiple NDCouplingNodes in the system. Here, this method requires a system administrator to decide in advance how the file system should be distributed and to manually assign a subpart of the directory hierarchy to individual NDCouplingNodes. All the NDCouplingNodes in the system have the mapping information about which node is responsible for what subpart of the file system directory. The system can process the request at most one hop to determine the appropriate NDCouplingNodes because the subparts of the hierarchy are treated as independent structures.

Under Distributed MDM, for a power-proportional file system, each node is able to locally acquire metadata for the transfer of data of certain files to other nodes. As a result, in general, the process of transferring data during a system configuration change is distributed among multiple nodes in the system. Consequently, the cost of this process could be less than that for a normal HDFS that includes only one NameNode to perform metadata management.

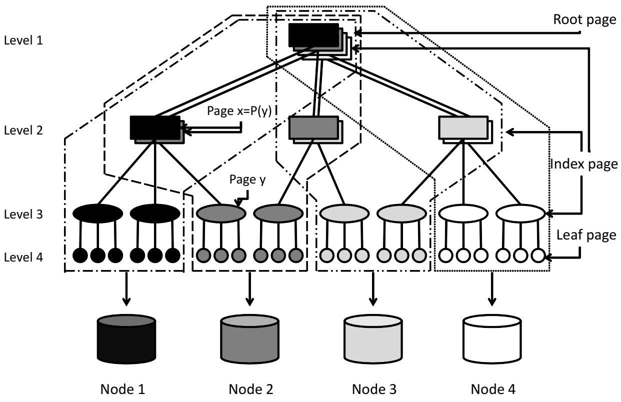


Fig. 3 An example of Fat-Btree

4.2.2 Fat-Btree-based Method

The second approach of Distributed MDM which is based on a parallel B-tree called Fat-Btree is described. Fat-Btree is an update-conscious parallel B-tree structure that was originally proposed in database management as an indexing technique for efficient data management [6,12]. Because of the tree structure, compared with the SDP method, the Distributed MDM based on Fat-Btree achieves high performance for range search query processing while maintaining good locality tracking of the file system. The formal definition of a Fat-Btree is given using the following notation. Each page of a tree is distinguished by its identifier i, and the level of page i is denoted by $L(i)$. When page i is the parent of page j, $P(j) = i$ and $L(j) = L(i) + 1$. Therefore, let S_i be a set of nodes storing page i. Then, a parallel B-tree structure satisfying the condition $S_i \supseteq S_j$ if $i = P(j)$ is called a Fat-Btree structure. For example, in Fig. 3, $x = P(y)$, $S_x = \{\text{Node 1}, \text{Node 2}\}$, $S_y = \{\text{Node 2}\}$. From this figure, it can be observed that multiple copies of index pages close to the root page with relatively low update frequency are replicated on several nodes, while leaf pages with relatively high update frequency are distributed across the nodes. Thus, the maintenance cost of the Fat-Btree is much lower than that of other parallel Btree structures. In addition, the Fat-Btree has a more efficient concurrency control protocol than other parallel B-tree methods [12].

4.2.3 Alternative Techniques

To realize good performance with Distributed MDM, many recent systems distribute the metadata across multiple nodes utilizing distributed hash table. Vesta [13], zFS [14] and Lustre [15,16] all hashed the file pathname and/or some other unique identifier to determine the location of metadata. However, distributing metadata by hashing eliminates all hierarchical locali-

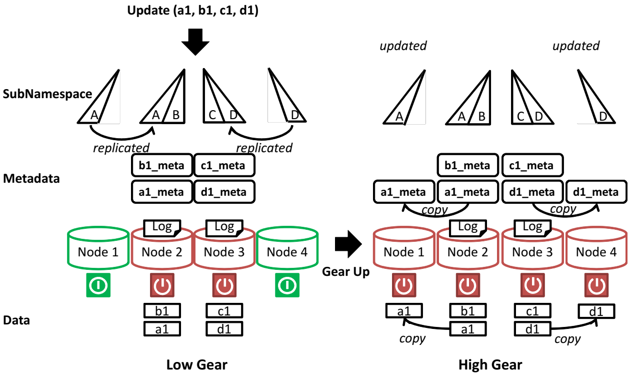


Fig. 4 Operations at updated data reflection processes of NDCouplingHDFS

ties, and with them, many of the typical locality benefits of local file systems, for instance, the POSIX directory access semantics. In such operations, the metadata management must traverse prefix directories containing a requested piece of metadata to ensure that the directory permissions allow the current user access to the metadata and data in question. Because the files and directories are scattered throughout the directory hierarchy, a hashed metadata distribution results in a traversal of metadata scattered over multiple nodes.

4.3 Updated Data Reflection

In this section, we describe the behavior of NDCouplingHDFS in serving the updated-data requests in a low gear and reflecting the updated data when the system changes to a high gear by reactivating a subset of nodes. In the normal HDFS, basically all the operations are similiar however because there is only a single NameNode that is in charge of metadata management, all the metadata operations are proccessed at the NameNode. Figure 4 shows a simple example of a four-node system in which each node maintains a subNamespace of the system. In the low gear, Node 1 and Node 4 are inactive, and their maintenance data are consequently replicated at Node 2 and Node 3. When the system changes to the high gear, Node 1 and Node 4 will be reactivated .

In the low gear, certain parts of the new updated data cannot be reflected at the deactivated nodes. In such cases, according to predefined data placement policy, the system chooses another node from among the active nodes to serve this request. Information about the data, the temporary node, and the intended node is saved into a log file (Log file). This information will be retrieved when the system changes from the low to the high gear. In this example, Node 2, which maintains the replication metadata and data of Node 1, will update the data (here is $a1$) that should be updated by Node 1.

When the system changes to the high gear by reactivating nodes (Node 1 and Node 4), the following four-step operations are carried out.

(1) Step 1: Transfer updated metadata.

First, the active nodes (Node 2, Node 3) check the Log files and the metadata management transfers only the different metadata, here are *a1_meta* and *d1_meta*, to the activated nodes (Node 1 and Node 4).

(2) Step 2: Issue block transfer commands.

Next, in the log records, the metadata management searches for updated file blocks. It then issues the block transfer command by filling the block transfer queue of each Storage Management with the block and destination node paired information. After each *heartbeat_interval*, the Storage Management receives a command and transfers the blocks to the destination nodes. Here, there are two approaches for issuing a command, namely sequential and batch methods. The **sequential issuance method** repeats the above search-and-issue operation for each transferred file, while the **batch issuance method** first looks for all the blocks and their destination nodes and then places them into a queue.

(3) Step 3: Transfer updated blocks.

When the Storage Management receives the command issued by metadata management, it sends the blocks to the destination nodes. However, in the current implementation in this part of the HDFS, for each block, the system has to open a new connection to the destination node. Here, to increase the efficiency of reflecting updated data, it is favored to reduce this cost. In the case of the **batch issuance method**, when the Storage Management knows all the blocks it has to transfer, the cost of opening a new network connection can be reduced by sending all the blocks through just a single network connection. We call this the **batch transfer method**. The current implementation in the HDFS is called the **sequential transfer method**.

(4) Step 4: Reflect updated metadata.

After receiving the updated data, the just-activated nodes notify the newly arrived data information to the responsible metadata management as in the default HDFS.

5. Experimental Evaluation

We carried out an empirical experiment with actual machines and an HDFS to verify the contributions of this paper. In the first part, we validate NDCouplingHDFS architecture with the original HDFS in terms of reducing the cost of updated data reflection when the system shifts to higher gear. Next, we present MapReduce benchmark results for both NDCouplingHDFS and the original HDFS to confirm the capability to support MapReduce. Finally, we examine the effectiveness of

Table 1 Characteristics of the configurations used in experiments

Configuration	**NormalHDFS**	**SSS**	**SBS**	**SBB**	**FBB**
Metadata management	Centralized	**S**DP	**S**DP	**S**DP	**F**at-BTree
Command issuance	Sequential	**S**equential	**B**atch	**B**atch	**B**atch
Block transference	Sequential	**S**equential	**S**equential	**B**atch	**B**atch
Updated metadata transference	-	○	○	○	○

Distributed MDM relating to the scalability of metadata operations.

5.1 Updated-data Reflection

We verified our proposal with a normal HDFS in terms of effective updated-data reflection [1]. To verify the effectiveness of each contribution proposed in Sec. 4, we prepared several configurations. Here, the **NormalHDFS** uses the sequential issuance method and sequential transfer method. The **SSS** configuration, which is the simplest configuration of NDCouplingHDFS, is the combination of SDP Distributed MDM, sequential issuance method and sequential transfer method. The **SBS** configuration is configured to verify the effect of batch issuance method and is integrated from SDP, batch issuance method and sequential transfer method. Moreover, to verify the contribution of the block transfer method, the **SBB** configuration that uses SDP, batch issuance method and batch transfer method is examined. Finally, we prepare the **FBB** configuration that implements Fat-BTree-based metadata management, batch issuance method and batch transfer method. The characteristics of each configuration are summarized in Tab. 1.

5.1.1 Experimental Environment

We compare the proposed NDCouplingHDFS with the normal HDFS in terms of reducing the cost of reflecting updated data when the system shifts to a higher gear by changing the configuration of the system. Here, we use the execution time to finish reflecting updated data to present this cost based on the observation that the current power-proportional systems are mainly achieved through a dedicated data placement method with a good load balancing mechanism. As a result, when the system moves to a higher gear, it must reorganize the data layout through reflecting updated data before serving the request on processing newly updated dataset. Consequently, the system is preferable to finish the reflecting updated data process as fast as possible in order to reach to the higher performance regarding to power-proportionality.

In this experiment, NDCouplingHDFS and HDFS operate in two gears, a low gear and a high gear with different configurations. For **NormalHDFS**, there is one NameNode, and eight active DataNodes in the low gear and 16 active DataNodes in the high gear. For NDCouplingHDFS, there are eight active NDCouplingNodes in the low gear, and 16 active NDCouplingNodes in the high gear (Tab. 2). Here, because we focus on the energy-aware system, we use low-power-consuming ASUS Eeebox EB1007 machines, whose specifications are given in Tab. 3. Furthermore, because we want to evaluate the effectiveness of metadata management in this paper, the number of updated (newly created) files when the system operates at the low gear is fixed at 16000 dividing equally to 16 nodes, and the size of each file is 1MB. The *max.rep-stream*, which specifies the maximum number of blocks that can be replicated by a Storage Management at the same time, is set to 100. In order to efficiently perform the updated data reflection, the communication frequency between NameNode Management and Storage Managements is increased by setting *heartbeat_interval* to one second, the smallest allowable value in the current HDFS (Tab. 4).

Table 2 Experimental environment

Number of Gears	2
Number of active nodes at Low Gear	8
Number of active nodes at High Gear	16
Number of updated files	16000
File size	1MB

Table 3 Specification of a node

CPU	TM8600 1.0GHz
Memory	DRAM 4GB
NIC	1000 Mb/s
OS	Linux 3.0 64bit
Java	JDK-1.7.0

Table 4 HDFS information and parameters

version	0.20.2
max.rep-stream	100
heartbeat_interval	1 second

5.1.2 Experimental Results

Figure 5 shows the execution time for reflecting the updated data to just-activated nodes using different methods employing the combination of this paper's contributions. The left vertical axis shows the execution time from the time that the system begins to change from the low gear to the high gear until all the just-activated nodes catch up with the most current status of the updated data set. The right vertical axis shows the maximum number of transfer block command issuances, which is the number of times that the Storage

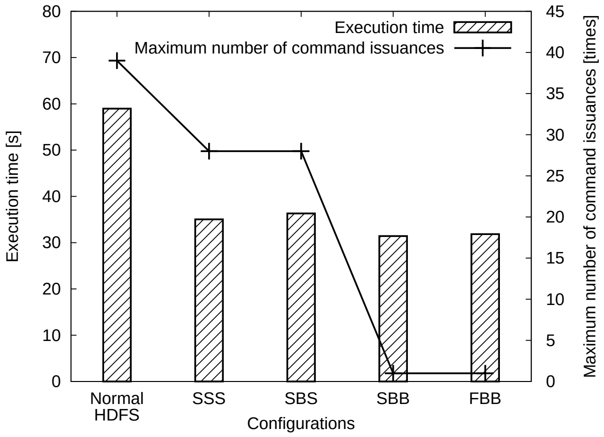


Fig. 5 Experimental results for updated data reflection

Management has to make a connection with the metadata management to drain the block transfer queue.

(1) Performance of NDCouplingHDFS.

To confirm the NDCouplingHDFS's performance, we focus on the experimental results of **NormalHDFS** and **SSS**, the simplest configuration of NDCouplingHDFS, in Fig. 5. We see that NDCouplingHDFS has significantly reduced cost in reflecting updated data; the reduction is nearly 41% (from 58 seconds to 34 seconds). In the HDFS, because of the large load at the NameNode with the processing of 8000 files that should be replicated to eight nodes, it requires about 40 connections (about 40 seconds since *heartbeat_interval* equals one second) between the NameNode Management and Storage Management to drain the block transfer queue of the Storage Management. Meanwhile, the process is distributed to eight NDCouplingNodes in NDCouplingHDFS, and the load at each NDCouplingNode is thus divided by eight (eight active nodes in the high gear) and overall it takes about 34 seconds to complete. It internally includes the execution times of four steps, i.e. transferring updated metadata, issuing block transfer command, transferring updated blocks and reflecting updated metadata as described in Sect. 4.3. Specifically, the time for transferring updated metadata was about 1.8 seconds. In HDFS, it is estimated that the data size of metadata of a single file or a single block is 200 bytes [8]. As a result, in our experiment, the amount of transfered updated metadata of 1000 one-block files was about 390KB. The remaining time (32.2 seconds) was for the last three steps in which the transferring updated blocks step was the major. As these three steps were executed in parallel at several nodes in almost non-sequential manner, it was difficult to clearly classify the results of each step. However, because the number of blocks that can be transfered between two nodes at the same time is fixed through parameter *max.rep-stream* (= 100), it can be derived that the execution time required in these three steps is the sum of three kinds of time, i.e. the time for issuing the block transfer commands for the first 100 blocks, the time for transfer all updated blocks (1000MB) and the time for reflecting updated metadata of the last group of blocks. The number of blocks in the last group generally was smaller than 100. Because issuing command and reflecting the metadata are processed on memory, their execution times were significantly small (several milliseconds) and could be ignored.

As indicated by the maximum number of command issuances, in the normal HDFS, the retrieval of metadata information is restrained at NameNode, and the possibility of increasing the free time at several DataNodes is higher than in NDCouplingHDFS.

(2) Performance of the command issuance.

From the results of **SSS** and **SBS**, we see that the batch command issuance provided a slightly worse result than did sequential command issuance. The reason is that the Storage Managements in **SBS** wasted several first connections to the NameNode Management before it had finished retrieving all 1000 updated files' data. On the other hand, the Storage Management in **SSS** can perform the block replication process immediately from the very first communication.

(3) Performance of the block transfer method.

Figure 5 shows that **SBB** reduces costs by approximately 10% compared with **SBS**, as reduce the execution time of the process to 31 seconds. This means that batch block transfer was able to reduce the cost of opening a new network connection for sending blocks. In total, SDP-based NDCouplingHDFS was able to reduce the execution time required for reflecting the updated data by 46% relative to **NormalHDFS**.

(4) Performance of Fat-Btree-based distributed metadata management method.

We also confirmed the validity of introducing Fat-Btree-based Distributed MDM methods. It is easier for Fat-Btree to implement the batch transfer block as it has the advantage of a range search. There was little difference between the performance of **FBB** and **SBB**. The cost of the latter is slightly less by 0.5 seconds owing to the lower cost of metadata management operations. This is due to the process of transferring incremental metadata, as the Fat-Btree-based method has to transfer more information than SDP because of the complex structure. However, the difference (0.5 seconds) is small compared with the overall execution time.

(5) Remarks.

The different contributions of this paper are verified through empirical experiments on actual machines. From the experiment results, our proposed system is able to reduce the cost of reflecting updated data compared with the case for the default HDFS. The simplest

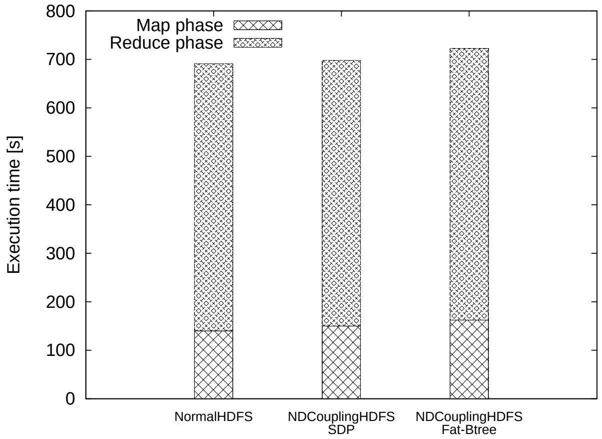


Fig. 6 Execution time of the wordcount benchmark

configuration of NDCouplingHDFS achieved up to 41% better performance, and the batch block transfer in the data transfer implementation further reduced computational costs by more than 5%.

5.2 MapReduce Benchmark

Here, to confirm that NDCouplingHDFS has the capability to support the MapReduce framework, we ran a *wordcount* benchmark on a 16-node NDCouplingHDFS (SDP-based and Fat-Btree-based). The nodes are similar to those used in previous experiments. Because we also wanted to verify the metadata operation's performance for a large number of files, we tested our system with a 1-GB data set of 1024 files of 1 MB each, which is equivalent to the block size of the HDFS. The replication parameter was set to one. Figure 6 shows the execution time including the running time in map phase and reduce phase.

Figure 6 shows that almost the same result for the benchmark was obtained for all three configurations. The result for NDCouplingHDFS was dominated by the network communication cost, especially at map phase, which required only light read operations. In the MapReduce framework, a *jobtracker* must first open the files' metadata to locate the blocks to allocate the job to *tasktrackers* that are close to the files. Because in NormalHDFS, the *jobtracker* is also located locally in the NameNode, all the work can be performed within the NameNode. However, because NDCouplingHDFS distributes the namespace to multiple nodes, it requires network communications among the nodes in this phase.

To confirm the effect of SDP and Fat-Btree-based methods on MapReduce jobs, the performance evaluation relating to the scalability of simple read/write metadata operations is reported in the next section.

Table 5 Workload

Fat-Btree leaf fanout	16
Data size ($\#files$)	3000
Number of nodes ($\#nodes$)	1, 2, 4, 8
File size	1KB
Number of write accesses per node	$\frac{\#files}{\#nodes}$
Number of read accesses per node	$\#files$

5.3 Distributed metadata management Performance

5.3.1 Experimental Environment

We verified the scalability of NDCouplingHDFS relating to read and write operations by changing the number of nodes and the data size as shown in Table 5. For the write workload, each node generated an equal number of requests as the number of files divided by the number of nodes used in that experiment run. For the read workload, each node performed work that requires the scanning of the whole data set, which means reading all the files in the system without redundancy. The order that files were requested was randomly generated. Furthermore, because we wanted to verify the effectiveness of namespace operations, the size of the physical file was kept to a small 1 MB.

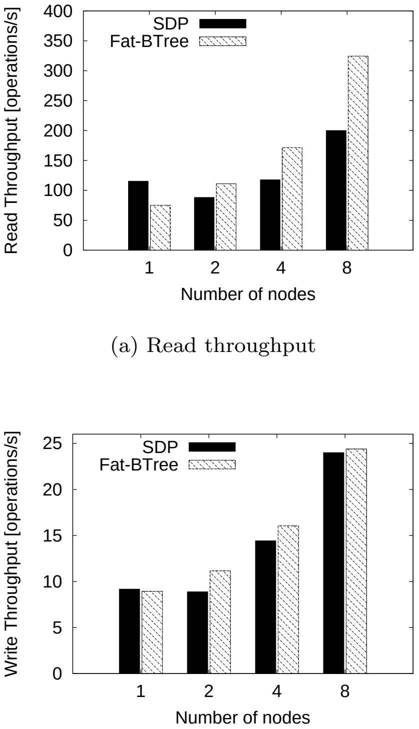
5.3.2 Experimental Results

Figure 7 shows the read and write throughput of SDP and Fat-Btree. Note that we also performed the experimentation for the normal HDFS in which there were only one node to perform both NameNode and DataNode's functionalities under the same environments. It was observed that these results were almost the same as the results for SDP when there was one node. The reason is that the overhead of SDP relating to identify which node to serve the request was extremely small in this case.

From Fig. 7(a) it is seen that the read performance of the Fat-Btree-based method significantly scales out as the number of nodes increases. Here, the transaction includes searching for the block locations of the query file and reading the blocks from the nodes. The good balance of the parallel B-Tree structure means that the read requests are effectively distributed to all the nodes in the system; hence, the overall throughput increased as the number of nodes increased.

In contrast, in the case of the SDP-based method, the throughput slightly decreased as the number of nodes increased from one to two. The reason here is the cost of opening a new connection to other nodes that are responsible for the request is much larger than the cost of searching for the responsible INode of the searched file.

Figure 7(b) describes the overall throughput for write requests. Here, a transaction is determined by



(a) Read throughput

(b) Write throughput

Fig. 7 The scalability of read/write performance of SDP and Fat-Btree-based NDCouplingHDFS

the two processes of creating new metadata for the file, and actually writing the physical data of the file to the node. Overall, it is seen that the Fat-Btree-based method do not provide such a considerable efficiency compare with the SDP-based method because of the high cost of synchronizing of B-tree structures among nodes during an update. Whenever a node splits, leading to some change in the structure, the related nodes of different NDCouplingNodes need to be synchronized. On the other hand, in SDP, each partition of the directory is treated as local metadata and independent of other partitions at other nodes.

Overall, from the above read and write performance results, it is believed that the Fat-Btree is more suitable for the read-mostly workloads in MapReduce applications.

6. Conclusion and Future Work

In this paper, we first described the issue of inefficient updated data reflection and then proposed the NDCouplingHDFS architecture to solve the issue. Coupling the metadata management and data management at every nodes to efficiently localize the range of data, NDCouplingHDFS significantly reduced the execution time required to move updated data relative to the normal HDFS. Moreover, in order to further improve the performance of block transferring process, we suggested and evaluated several batch methods of command issuance and block transfer. Experiments using actual machines verified that our solution was able to shorten the execution time required to reflect updated data by 46% relative to the time required by the default HDFS. Moreover, NDCouplingHDFS was able to increase the throughput of the system supporting MapReduce by applying an index in metadata management. In addition, as utilizing metadata replication, NDCouplingHDFS was able to get rid of the single pointer of failure problem existing in the normal HDFS.

In the future, we would like to carry out more experiments with different workloads and a larger scale of nodes. Moreover, we would like to develop a system that integrates NDCouplingHDFS with suitable data placement to provide power proportionality. We would then like to evaluate this system for several workloads including MapReduce-based applications. Last but not least, as NDCouplingHDFS utilizes the distributed metadata management, it is suggested that optimization techniques of network communication cost reduction at internal metadata processing can be further explored.

Acknowledgements

This work is partly supported by Grants-in-Aid for Scientific Research from Japan Science and Technology Agency (A) (#22240005).

References

[1] H.H. Le, S. Hikida, and H. Yokota, "NameNode and DataNode Coupling for Power-proportional Hadoop Distributed File System," Proc. the 18th International Conference on Database System for Advanced Applications, Part II, DASFAA '13, vol. 7826, pp. 99–107, Springer Verlag, 2013.

[2] A. Hrishikesh, C. James, G. Varun, G. Gregory R., K. Michael A., and S. Karsten, "Robust and Flexible Power-proportional Storage," Proc. the 1st ACM Symposium on Cloud Computing, SoCC '10, pp. 217–228, ACM, 2010.

[3] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: Practical Power-proportionality for Data Center Storage," Proc. the 6th European Conference on Computer Systems, EuroSys '11, pp. 169–182, ACM, 2011.

[4] J. Kim and D. Rotem, "Energy Proportionality for Disk Storage using Replication," Proc. the 14th International Conference on Extending Database Technology, EDBT '11, pp. 81–92, ACM, 2011.

[5] Apache Hadoop, "HDFS Hadoop Wiki." `http://wiki.apache.org/hadoop/HDFS`.

[6] H. Yokota, Y. Kanemasa, and J. Miyazaki, "Fat-Btree: An Update Conscious Parallel Directory Structure," Proc. the 15th International Conference on Data Engineering, ICDE '99, pp. 448–457, IEEE Computer Society, 1999.

[7] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-loading: Practical Power Management for Enterprise Storage," ACM Transaction on Storage, vol. 4, no. 3, pp. 10:1–

10:23, 2008.

[8] K.V. Shvachko, "HDFS Scalability: The Limits to Growth," ;login:, vol. 35, no. 2, 2010.

[9] H.H. Le, S. Hikida, and H. Yokota, "An Evaluation of Power-proportional Data Placement for Hadoop Distributed File Systems," Proc. the 2011 Dependable, Autonomic and Secure Computing, DASC '11, pp. 752–759, IEEE Computer Society, 2011.

[10] W. Charles, O. Mathew, Q. Jin, W.A.I. Andy, R. Peter, and K. Geoff, "PARAID: A Gear-shifting Power-aware RAID," ACM Transaction on Storage, vol. 3, 2007.

[11] R.T. Kaushik and B. Milind, "GreenHDFS: Towards an Energy-conserving, Storage-efficient, Hybrid Hadoop Compute Cluster," Proc. the 2010 International Conference on Power Aware Computing and Systems, HotPower '10, pp. 1–9, USENIX, 2010.

[12] T. Yoshihara, D. Kobayashi, and H. Yokota, "A Concurrency Control Protocol for Parallel B-tree Structures Without Latch-coupling for Explosively Growing Digital Content," Proc. the 11th International Conference on Extending Database Technology, EDBT '08, pp. 133–144, ACM, 2008.

[13] P. Corbett and D. Feitelson, "The Vesta Parallel File System," ACM Transactions on Computer Systems, vol.14, no. 3, pp. 225–264, 1996.

[14] O. Rodeh and A. Teperman, "zFS - A Scalable Distributed File System using Object Disks," Proc. the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, MSST '03, pp. 207–218, IEEE, 2003.

[15] "Lustre-Main Page." `wiki.lustre.org/`.

[16] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," Proc. the 2003 Linux Symposium, 2003.

Hieu Hanh LE received his B.E., and M.E. degree from Tokyo Institute of Technology in 2008, and 2010, respectively. He is currently a Ph.D student at Tokyo Institute of Technology. He is interested in research on data engineering, information storage systems, and network engineering. He is a member of IPSJ.

Satoshi HIKIDA received his M.E. degree from Tokyo Institute of Technology in 2011. He is currently a Ph.D student at Tokyo Institute of Technology. He is engaged in research on data engineering, and information storage systems. He is a student member of IPSJ.

Haruo YOKOTA received his B.E., M.E. and Dr.Eng. degrees from Tokyo Institute of Technology in 1980, 1982, and 1991, respectively. He joined Fujitsu Ltd. in 1982, and was a researcher at ICOT for the Japanese 5th Generation Computer Project from 1982 to 1986, and at Fujitsu Laboratories Ltd. from 1986 to 1992. From 1992 to 1998, he was an Associate Professor at Japan Advanced Institute of Science and Technology (JAIST). He is currently a Professor at the Department of Computer Science in Tokyo Institute of Technology. His research interests include the general research areas of data engineering, information storage systems, and the dependable computing. He was a chair of ACM SIGMOD Japan Chapter and a trustee member of IPSJ. He is a Vice Chair of the Database Society of Japan (DBSJ), the Editor-in-Chief of Journal of Information Processing, an associate editor of the VLDB Journal, a fellow of IEICE and IPSJ, and a member of JSAI, IEEE, IEEE-CS, ACM, and ACM-SIGMOD.