J Supercomput (2017) 73:2657–2681
DOI 10.1007/s11227-016-1949-7

Moving metadata from ad hoc files to database tables for robust, highly available, and scalable HDFS

Heesun Won[1,2] · Minh Chau Nguyen[2] · Myeong-Seon Gil[3] · Yang-Sae Moon[3] · Kyu-Young Whang[1]

Published online: 27 March 2017

Abstract As a representative large-scale data management technology, Apache Hadoop is an open-source framework for processing a variety of data such as SNS, medical, weather, and IoT data. Hadoop largely consists of HDFS, MapReduce, and YARN. Among them, we focus on improving the HDFS metadata management scheme responsible for storing and managing big data. We note that the current HDFS incurs many problems in system utilization due to its file-based metadata management. To solve these problems, we propose a novel metadata management scheme based on RDBMS for improving the functional aspects of HDFS. Through analysis of the latest HDFS, we first present five problems caused by its metadata management and derive three requirements of *robustness, availability, and scalability* for resolving these problems. We then design an overall architecture of the advanced HDFS, *A-HDFS*, which satisfies these requirements. In particular, we define functional modules according to HDFS operations and also present the detailed design strategy for adding or modifying

✉ Kyu-Young Whang
kywhang@cs.kaist.ac.kr; kywhang@mozart.kaist.ac.kr

Heesun Won
hswon@etri.re.kr

Minh Chau Nguyen
chau@etri.re.kr

Myeong-Seon Gil
gils@kangwon.ac.kr

Yang-Sae Moon
ysmoon@kangwon.ac.kr

1 School of Computing, KAIST, Daejeon, Korea

2 BigData Intelligence Research Department, ETRI, Daejeon, Korea

3 Department of Computer Science, Kangwon National University, Chuncheon, Korea

the individual components in the corresponding modules. Finally, through implementation of the proposed A-HDFS, we validate its correctness by experimental evaluation and also show that A-HDFS satisfies all the requirements. The proposed A-HDFS significantly enhances the HDFS metadata management scheme and, as a result, ensures that the entire system improves its stability, availability, and scalability. Thus, we can exploit the improved distributed file system based on A-HDFS for various fields and, in addition, we can expect more applications to be actively developed.

Keywords Hadoop · HDFS · Advanced HDFS · Distributed file systems · Metadata management

1 Introduction

Apache Hadoop [1] is a representative low-cost, high-efficiency distributed system proposed for big data management, gaining a lot of attention since it was first introduced. Even though Hadoop is an open-source software with its functions continuously evolving, it has become a de facto base technology for storing and analyzing a large volume of data. We can exploit Hadoop in a variety of applications such as SNS, medical, image, and weather services. Recently, a number of studies on IoT (Internet of Things), an increasingly growing topic, have been conducted based on Hadoop [20–23]. The latest version of Hadoop consists of HDFS(Hadoop Distributed File System) [3] responsible for data storing and management, YARN (Yet Another Resource Negotiator) [2] responsible for resource management and job execution management, and MapReduce [4] working as a programming model for data processing and analysis. In this paper, we focus on HDFS to improve its functional aspects.

HDFS, a framework inside Hadoop, can substitute legacy storage systems, which are not suitable for management of a large volume of data. It is a low-cost and high-efficiency distributed file system that consists of a large-scale cluster using commodity computer nodes. Since HDFS can store large-scale files by splitting big data into multiple blocks and easily extending its nodes, it is gaining growing attention as a representative technique for big data storage. Even recently, many companies such as IBM [11], Amazon [12], Cloudera [13], Hortonworks [14], and MapR [15] have developed HDFS-based storage systems and have provided them as their own products or new technologies to users. Furthermore, since the advent of HDFS, there have been many successful studies and products on newly distributed file systems such as Cassandra [16], Ceph [17], Lustre [18], and OneFS [19].

However, the existing HDFS incurs many problems caused by storing and managing metadata into ordinary files. In this paper, we propose an advanced HDFS, called *A-HDFS*, for solving those problems. The problems of the existing HDFS become much severe in system scalability and stability as the number of system objects such as users, files, and directories increases. According to our analysis, most of these problems result from inefficient file-based metadata management. In this paper, we identify five problems through analysis of the existing HDFS.

Problem 1 (*Metadata Limitation Problem*) Since the existing HDFS loads all the metadata into the main memory of NameNode, the number of files (or directories) might be limited. □

Problem 2 (*Slow Bootstrapping Problem*) The existing HDFS manages the metadata in complex image and log files, and this incurs a long bootstrapping time since complicated integration of these complex configuration files should be done at the time of system (re)boot. □

Problem 3 (*Complex Management Problem*) The existing system can search or modify the metadata only through commands, and this command-based scheme is highly inefficient for a large-scale distributed environment, which should manage a variety of metadata of different objects. □

Problem 4 (*Location-aware Accesses Problem*) Since Hadoop manages the metadata in a single NameNode (or one or more separate NameNodes in a federation mode), all the access requests to HDFS need to pass through the corresponding NameNode. This access structure severely restricts the Hadoop scalability. □

Problem 5 (*Access Limitation Problem*) Since Hadoop does not permit user applications to access the HDFS metadata, these applications cannot exploit HDFS for various purposes such as HDFS status monitoring or usage statistics. □

To solve these five problems, this paper derives three requirements for improving HDFS functions. First, *robustness* related to Problems 2 and 3, means that an integrated metadata management system is required for resolving the instability of the original HDFS. Second, *availability* related to Problems 2, 4 and 5, means that an *open* metadata management system is required for resolving the application limitations caused by the current *closed* metadata management system. Third, *scalability* related to Problems 1 and 4 means that, since the existing NameNode-centered metadata management has difficulty in constructing a large-scale cluster that is geographically distributed, a new management scheme for solving this difficulty is required.

To satisfy these three requirements, we propose a new HDFS, A-HDFS, which adopts RDBMS for metadata management. By exploiting RDBMS, which has already proven its efficiency and stability, Hadoop can ensure more efficient system operations, stable metadata management, and high scalability. For the design and implementation of A-HDFS, we first analyze the latest HDFS source codes and present a two-step design for confirming that A-HDFS works correctly. In Step 1, we design and implement the *Writing Process*, which stores the HDFS metadata into RDBMS without affecting the existing HDFS operations. In Step 2, we discard all the original HDFS metadata files, but we instead develop the *Reading Process* for A-HDFS to work correctly just through the integrated MetaDatabase newly constructed. Following this two-step design, we develop the entire system of the proposed A-HDFS by modifying the actual source codes, and we also prove that the developed system satisfies each of the three requirements by experimental evaluation. The experimental results confirm that A-HDFS makes the metadata management more convenient and reflects the metadata information changes to the running system immediately. Moreover, the results

show that the system utilization is improved by allowing external applications to easily access the integrated MetaDatabase and that the cluster size is largely extendible.

The contribution of this paper is as follows. First, while previous efforts have focused on only the performance improvement of Hadoop [25–28,33], we instead focus on the functional improvement of Hadoop, and this improvement can be a basis for maximizing the actual system utilization. Second, we derive three major requirements, *robustness*, *availability*, and *scalability*, as the solution of the five problems in the legacy HDFS and present how to satisfy these requirements by proposing a novel architecture of A-HDFS and its detailed working mechanism. More precisely, we achieve these three requirements by replacing an old file-based metadata management with a new RDBMS-based metadata management, which will be explained and evaluated in detail in Sects. 3 and 5. Third, we increase the overall computing resource utilization by assigning the metadata management function to Secondary NameNode, which has no significant function in the current Hadoop. Fourth, through actual implementation and experimental evaluation, we prove that the proposed A-HDFS satisfies all three requirements and its functions work correctly.

The rest of this paper is organized as follows. Section 2 describes related work and previous research efforts. Section 3 analyzes the problems of existing system and derives the requirements for solving these problems. Based on the requirement analysis of Sect. 3, we propose in Sect. 4 A-HDFS, which uses an RDBMS-based metadata management scheme. Through actual implementation, we present in Sect. 5 the experimental results, which validate that the A-HDFS functions fully satisfy all the requirements of Sect. 3. Finally, Sect. 6 concludes the paper with future work.

2 Related work

2.1 Apache Hadoop

Apache Hadoop [1] is an open-source framework for storing, managing, and analyzing a large volume of data. Hadoop consists of three large components: HDFS (Hadoop Distributed File System) [3], MapReduce [4], and YARN (Yet Another Resource Negotiator) [2]. First, MapReduce is a programming model proposed by Google. At the same time, it is a software framework for performing job execution management in Hadoop. In the MapReduce model, we write a distributed program using two primitive functions: Map and Reduce. In Hadoop 1.x, JobTracker and TaskTracker are responsible for job processing and resource management, respectively. This architecture, however, incurs a severe functional overhead in MapReduce, especially in JobTracker. To solve this problem, Hadoop 2.0 introduces the new resource management framework, YARN, to remove the bottleneck caused by inefficient distribution functions in the initial Hadoop. YARN efficiently manages and controls various resources and computing tasks inside the system while providing the extension with a variety of applications, as well as MapReduce. YARN mainly consists of ResourceManager and NodeManager for cluster resource management, and ApplicationManager and ApplicationMaster for job execution management. As we can see in these components, YARN splits the role

of the existing JobTracker into ResourceManager and ApplicationManager, ensuring the stability and scalability of the entire cluster [2].

Finally, HDFS is a distributed file system proposed by Google as a variant implementation of GFS (Google File System) [5]. HDFS is basically designed for handling large-scale files rather than small ones. HDFS mainly consists of NameNode as a master role and DataNode as a slave role. NameNode is responsible for the entire cluster management and uses the metadata to manage all the data distributed in DataNodes. In addition, it might include Secondary NameNode for the purpose of failure recovery and backup of NameNode.

2.2 Metadata management in HDFS

HDFS stores a single data file into multiple blocks and, accordingly, the metadata relating to each data file are highly complex compared to the legacy file system, with the internal structure for managing data files becoming much more complicated. Each data file contains a variety of metadata information such as the directory address for the file location, permissions to file owners and groups, and storage locations for blocks split from the file. The current Hadoop manages the metadata of an entire cluster in a system backup file, called *FSImage*, and a log file, called *EditLog*. Also, it maintains a few separate XML files for the purpose of the configuration and policy management of the Hadoop system [24].

As the abbreviation of a file system image, FSImage stores the metadata information of files including locations of files and directories, ACL (access control list), and block information on each file. We note that FSImage resembles the iNode information in a Linux system [29] and can be considered a main metadata file for managing the iNode information with the system status information in HDFS. Thus, many console commands are processed by referring to the metadata of FSImage, and Secondary NameNode can take over the failed NameNode by using its FSImage periodically copied from NameNode. EditLog is a log file that accumulates every event occurring in HDFS after the most recent FSImage and is used for system monitoring and backup together with FSImage. By using the EditLog file, we can trace all the events occurring in Hadoop and obtain up-to-date metadata information, which reflects the most recent file and system status. Based on these two files, the metadata for files and directories are managed in a similar way as iNode in Linux, and they are loaded in the main memory of the NameNode server. Thus, using these memory-resident metadata, the Hadoop system can retrieve files, directories, and blocks being accessed by current users [3].

Besides FSImage and EditLog files, a few XML files are used for various system configurations. These XML files are mainly required for configuring HDFS, YARN, MapReduce, and KMS (Key Management System). The XML configuration files are as follows.

- Cluster configuration: core-site.xml, hadoop-policy.xml
- HDFS configuration: hdfs-site.xml, https-site.xml
- YARN configuration : yarn-site.xml, capacity-scheduler.xml
- MapReduce configuration: mapred-site.xml
- KMS configuration: kms-acls.xml, kms-site.xml

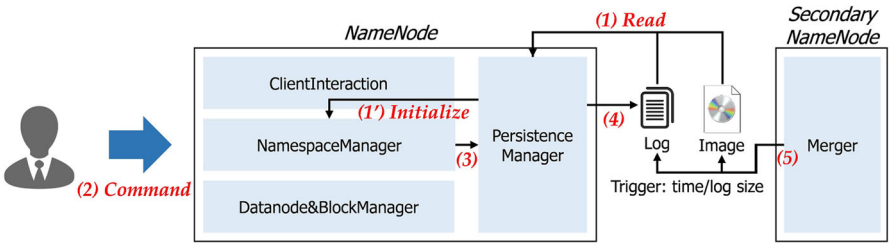

Fig. 1 Read/write operation procedures of the metadata in the existing HDFS

Five site-related XML files, i.e., core-site.xml, hdfs-site.xml, yarn-site.xml, mapred-site.xml, and kms-site.xml, manage the network-related metadata such as the network addresses of nodes, port numbers, and server names. The site information usually contains mandatory configuration parameters required for the initial Hadoop installation. If the information is changed, the Hadoop system often needs to be rebooted [30]. Next, hadoop-policy.xml in cluster configuration manages the ACL information between Hadoop and other services. As a key management server for secure communication between a specified server and Hadoop, KMS stores the metadata for the distribution and management of security keys.

As shown in Fig. 1, FSImage and EditLog, which are frequently changed in many Hadoop operations, are accessed and managed by PersistenceManager, NamespaceManager, Datanode&BlockManager, and ClientInteraction components in the NameNode. The procedures of reading and writing the HDFS metadata are related to these four components and work as in the following steps. Steps (1) and (1′) are executed only once at the Hadoop start time.

(1) NameNode reads the metadata information from FSImage and EditLog files to start up the Hadoop system.
(1′) NameNode loads and maintains the metadata into the main memory.
(2) A client (user) inputs a command to use HDFS.
(3) NameNode retrieves the metadata related to the command and updates the metadata information in the main memory if necessary.
(4) NameNode writes the client's command into the EditLog file.
(5) Secondary NameNode integrates the EditLog to the FSImage periodically and refreshes EditLog to keep the log file to a specified size.

3 Problem and requirement analysis

In this section, we first analyze five problems of metadata management in the current HDFS and then derive three requirements to solve these problems. In particular, we note that the current HDFS manages the metadata based on files, and we explain the restriction and limitation of this file-based metadata management in detail.

Problem 1, called *Metadata Limitation Problem*, is caused by the size restriction of metadata for a file (or a directory) in HDFS. HDFS loads all the metadata of

files and directories into the main memory of NameNode; thus, the metadata size should be smaller than the main memory size. For a large Hadoop cluster, a NameNode server manages hundreds to thousands of DataNodes in general. For example, in 2012 the Hadoop system of Ebay consisted of 532 nodes [39]. In this case, there can be a large number of files and directories, but the actual number of files (or directories) might be limited due to the memory limitation of NameNode [28]. In particular, the amount of ACL information is also limited by this reason, and this ACL limitation makes it impossible to construct a large Hadoop cluster for supporting a large number of users, which will be the base of Multitenant Hadoop [10]. Recently, the number of applications running in Hadoop has been increasing fast. Problem 1, which restricts the extension of HDFS, should thus be resolved in the proposed A-HDFS.

Problem 2, called *Slow Bootstrapping Problem*, is caused by file-based metadata management. It occurs because all the metadata are maintained by file formats such as images, logs, and XML files. If a Hadoop user changes the information of files or directories, the current HDFS reflects it on the main memory only upon logging the change history to EditLog (without changing the FSImage file itself). For example, when user A requests the ACL modification of his/her file for user B, the change cannot occur immediately; instead, user B needs to wait for the process of main memory and log history reflection to access user A's file. Thus, there might be a discrepancy in the metadata between NameNode and its Secondary NameNode until the Secondary NameNode integrates EditLog to FSImage. Moreover, if certain parameters of the cluster configuration need to be changed, the administrator should modify the related XML files manually and reboot the entire Hadoop system to apply that modification. However, rebooting the whole system, which serves a large number of users and applications, creates a big difficulty in practice [32]; thus, altering important parameters related to the cluster configuration causes a very troublesome problem in a running Hadoop system. If we can solve Problem 2, the metadata changes can be applied to the system on the fly, and a more efficient operation and management are possible in the Hadoop system.

Problem 3, called *Complex Management Problem*, is caused by the same reason as Problem 2; i.e., it is caused by the file-based metadata management. In the current HDFS, searching and updating the metadata can be done by system commands only. Also, all important metadata such as file owners, ACL of files and directories, and file modification times are stored in a single file, FSImage, so the metadata management becomes much more complex as the number of files or directories increases, or as the number of users increases. Moreover, the system security becomes vulnerable since all the important metadata are maintained in only a few files [31]. To solve this *Complex Management Problem*, we need to construct an integrated metadata management environment and provide easy maintenance functions with appropriate system authorities.

Problem 4, called *Location-aware Access Problem*, is also caused by file-based metadata management. More precisely, this problem arises because HDFS manages all metadata files in a single node, and all data accesses should be done through the

NameNode regardless of geographically distributed users and DataNodes.[1] Because it is difficult to construct a large-scale Hadoop cluster in a single region, it is more efficient to operate small-scale data centers distributed in as many regions as possible [8]. For example, a CDN (content delivery network) service system consists of a big Hadoop cluster and multiple distributed *edge servers* [40], each of which corresponds to a small data center. However, such a system configuration cannot be possible in the current metadata management scheme because only a NameNode stores the metadata of DataNodes, and all data accesses should be made through the NameNode even for different locational users and distributed DataNodes. If the metadata can be shared among multiple NameNodes, many independent Hadoop systems are able to work together as a single big system. This multi-NameNodes approach makes it possible to adopt the location-unaware access strategy through the distributed operations of multiple data centers and to reduce the operational overhead of NameNode by operating multiple NameNodes.

Problem 5, called *Access Limitation Problem*, is a big constraint in developing applications running on top of HDFS. Since the current applications must run only on the Hadoop system, they need to do complex and unnecessary installation and configuration processes first. Also, they can use only simple interoperations or a few functions through APIs provided by Hadoop. Due to these unnecessary initialization processes and access limitations, the applications are not sufficient to utilize the whole system efficiently. To solve these problems, we construct an integrated meta-database by separating the metadata from HDFS and open the meta-database to the applications so that they can exploit the HDFS metadata easily with appropriate access rights. This openness feature also makes the existing applications utilize the Hadoop system more efficiently.

Based on the five problems of HDFS mentioned so far, we define three requirements for A-HDFS newly proposed in this paper. In Table 1, we summarize these requirements and explain them in detail.

First, as the requirement for Problems 2 and 3, robustness indicates that the Hadoop system needs to provide a stable and integrated metadata management environment. To satisfy this requirement, we need to consider an easier way of managing a large volume of metadata generated from HDFS. Also, we need to consider a stable way of reflecting metadata changes to the Hadoop system quickly. In particular, as we mentioned in Problem 3, the current HDFS is vulnerable to security compromise due to the file-based metadata management. Therefore, to satisfy the robustness requirement, we need to introduce a new metadata storing and managing scheme, which is strongly secure, easily manageable, and highly stable. If we can ensure robustness, we can alleviate the system operation instability caused by the metadata changes of Problem 2 and the complicated metadata management of Problem 3.

Second, as the requirement for Problems 2, 4, and 5, availability indicates increasing HDFS utilization. We can satisfy this requirement if we resolve the problems caused by the closed metadata management of the current HDFS. For this, we need to separate the

[1] In a federation mode of multiple NameNodes [37], each NameNode is still responsible for a large number of DataNodes. Thus, all data accesses in a set of DataNodes are also done through a NameNode even in such a federation mode.

Table 1 Definitions of A-HDFS requirements and related problems

Requirements	Explanation	Related problems
Robustness	We need to manage the metadata in an integrated manner rather than through the current file-based management for stability, simplicity, and efficiency	Problems 2 and 3
Availability	By the openness feature of the meta-database, we can apply the Hadoop system to a variety of applications and purposes	Problems 2, 4, and 5
Scalability	We need to make the Hadoop system to be scalable to support a large-scale cluster with a large number of files and a large number of users	Problems 1 and 4

metadata stored inside Hadoop into an external storage system and manage the storage stably and independently. In this paper, we first analyze the operational procedures, metadata files, and the corresponding source codes of the current Hadoop system and then present a novel method to bring all the metadata into RDBMS, an external storage space. If we manage the metadata in a separate RDBMS like above, we can resolve the current HDFS problems of searching and modifying configuration information from multiple files. Furthermore, other Hadoop systems or applications can access the metadata easily; thus, we are expected to construct geographically distributed data centers and develop various HDFS-based applications more easily.

Third, as the requirement for Problems 1 and 4, scalability extends the current HDFS to be more scalable for a large-scale distributed system. In general, the current Hadoop system constructs a cluster by connecting many nodes in an internal network and operates a single NameNode to manage those nodes. Against the NameNode failure, the Secondary NameNode is additionally used for failure recovery and high availability, but it is merely an auxiliary server to wait and back up NameNode periodically without any particular function. As the number of nodes, the number of files, and the number of users increase, the metadata size will increase and, accordingly, NameNode will suffer from the heavy load of managing a large size of metadata. In other words, the hardware environment of NameNode restricts the scalability of a whole cluster, and this can be a critical problem of the Hadoop system targeting big data. To satisfy scalability, we need to modify the current Hadoop operating structure from the memory-based structure to a separate and independent storage system structure, which is free from memory limitation. As a storage system, we adopt RDBMS since it is the most general-purpose and concrete system. By using RDBMS, we can solve two problems: first, we can support unlimited size of metadata with an unlimited number of files, and this solves Problem 2; second, we can extend the Hadoop cluster from a small region of a large data center to multiple distributed regions of many small data centers, and this solves Problem 4.

The problems and requirements described so far can be representative criteria to improve the functionality of HDFS. In Sect. 4, we first determine a new operating structure and its components to satisfy the requirements upon analyzing the current HDFS, and we then present an overall architecture of A-HDFS and its detailed functions.

4 RDBMS-based metadata management for A-HDFS framework

4.1 Overall architecture

In this section, we propose a new HDFS framework, A-HDFS, which exploits the RDBMS-based scheme to satisfy the requirements introduced in Sect. 3. This framework solves current HDFS problems of the file-based metadata management and plays an important role of integrating all the metadata scattered in many places of the Hadoop system into a separate storage system. As the storage system for the integrated metadata management, we adopt RDBMS, which has already proven its efficiency, stability, and security in a number of places over a long period. The existing metadata files, especially FSImage and EditLog, have complicated structures, which store all the information of complex metadata objects having different characteristics from each other in a single file. Therefore, it is not easy to even recognize which information is contained in the file, and it is also inefficient to search and manage specific information from the file. Moreover, it is difficult for other applications to access and use such complicated metadata structures. In contrast, RDBMS of this paper is highly convenient in many aspects of installation, operation, and utilization compared with the file-based management, and it can support various applications efficiently and effectively. Figure 2 shows the overall architecture of the proposed A-HDFS exploiting RDBMS for Multitenant Hadoop.

As shown in Fig. 2, we construct an integrated *MetaDatabase* in RDBMS instead of metadata files such as FSImage and EditLog. First, A-HDFS processes HDFS commands given by users by reading MetaDatabase rather than the original HDFS metadata. Also, it reflects the metadata information changes to MetaDatabase. In this way, we reorganize all of the FSImage, EditLog, and XML files of the current Hadoop system into the MetaDatabse of RDBMS and modify HDFS components to manage the

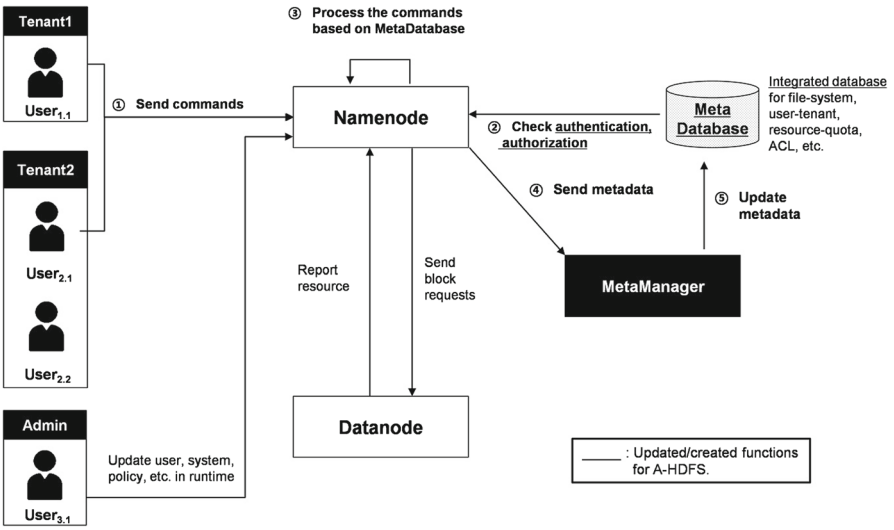

Fig. 2 The overall architecture of A-HDFS

stored objects of MetaDatabase. Moreover, to ensure correct operations of A-HDFS, we present a two-step design scheme by focusing on read and write operations on the metadata. The detailed two-step procedures of Fig. 2 can be found in Figs. 5 and 7, and their detailed working mechanisms are explained in Sects. 4.3.1 and 4.3.2, respectively.

4.2 Relational schema for HDFS metadata

As we mentioned earlier, we use RDBMS or, more specifically, RDBMS tables, for the efficient metadata management of A-HDFS. For this, we analyze all the metadata files of the current HDFS, categorize the information contained in each file according to its property and purpose, and finally define new relational schema that holds all the metadata. Figure 3 shows the metadata information used for the existing file-based management. As shown in the figure, the information on files (or directories) is mainly stored in FSImage and EditLog. For a single file (or a single directory), we note that the HDFS metadata contain various information such as nodes, blocks, and permissions. Thus, we group the information fields generated from a file (or a directory) according to their characteristics and define those groups as RDBMS tables. In case of system information data, such as cluster configuration and network setup data, we design them separate tables since they are not directly related with files or directories.

Figure 4 shows the entire table schema of A-HDFS newly designed for the integrated metadata management. The mapping relationship between Figs. 3 and 4 is shown in Table 2. Table 2 shows how we map the metadata of Fig. 3 to the relational tables of Fig. 4. We extract information fields such as File ID, Path, and Permission from FSImage and EditLog files and reflect those fields in the table structures as their attributes. We also make system-related tables by referring configuration XML files. Based on Fig. 4 and Table 2, we summarize the major metadata information of A-HDFS as follows.

- System information: inter-connection information of clusters and nodes, security configuration among clusters, service ACL, etc.
- File and directory information: ID, storage location, owner and owner group, access right by user and group, the number of data blocks, block locations, block size, creation/update times, etc.

Kerberos [6,34] has been introduced to complement a simple user authentication process of the initial Hadoop, and it has been an essential security mechanism [27] that should be used to securely manage all system components such as clusters, nodes, data, and users. Our Hadoop also uses the security mechanism of Kerberos, but note that we retrieve the user and group information from the integrated MetaDatabase rather than XML files.

4.3 A-HDFS with RDBMS-based MetaDatabase

Hadoop consists of a huge amount of source codes, and its configuration and operational procedures are very complex. As a result, even if we add or modify a function cautiously, there might be a risk that the system will not work correctly. Therefore,

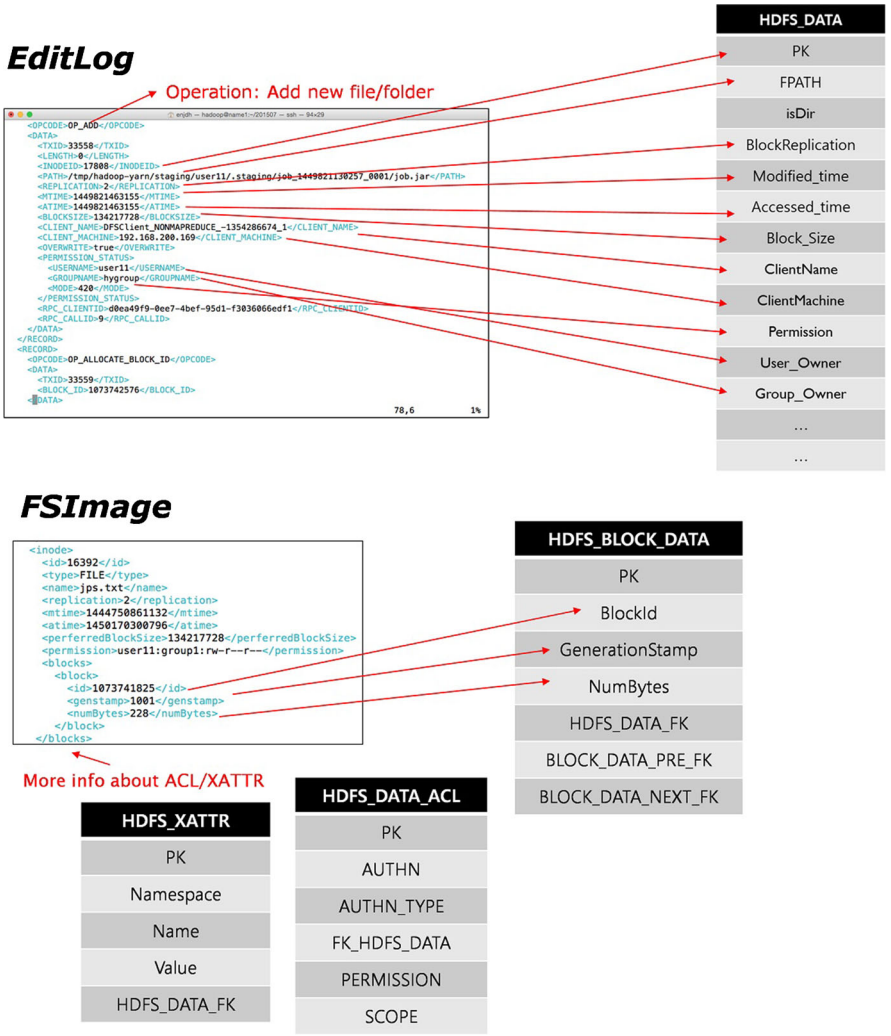

Fig. 3 The representative metadata of the current file-based HDFS

in this paper we adopt a stepwise improvement approach to confirm that the system works properly in each step. More specifically, we develop A-HDFS by two steps: the first step is to store the original metadata into MetaDatabase tables without reading or using the tables, and the second one is to read and use the MetaDatabase tables, as well as the first storing function.

4.3.1 Step 1: RDBMS-based metadata writing process

In Step 1, we design A-HDFS focusing on the writing process that stores the current HDFS metadata into RDBMS. In this step, we duplicate the metadata both in the

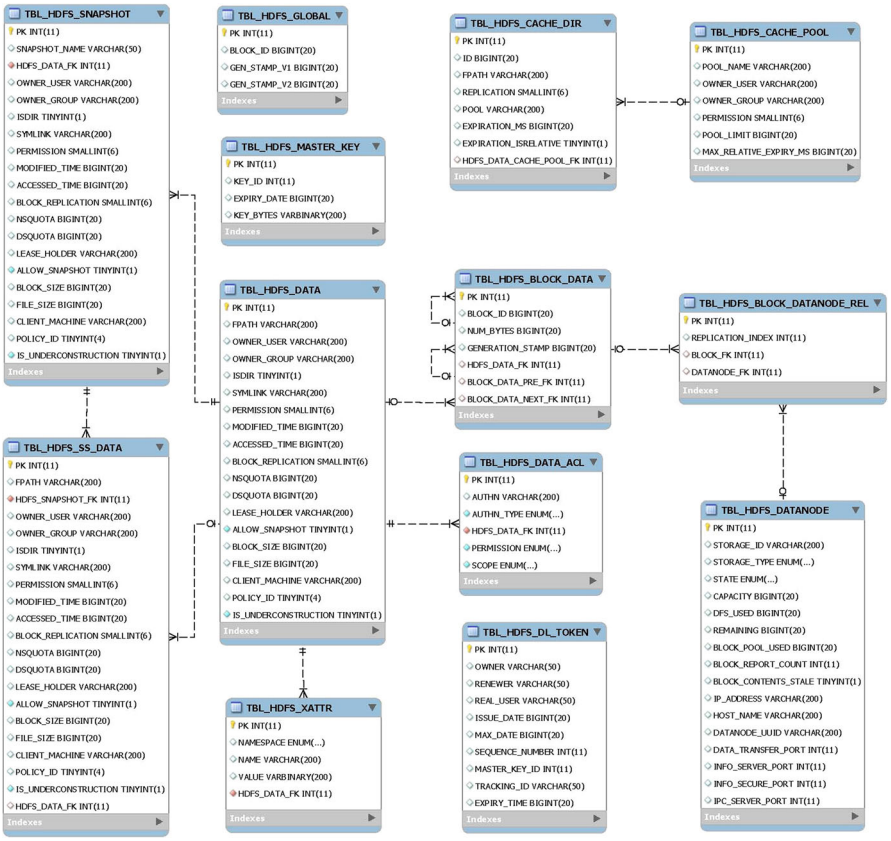

Fig. 4 The metadata table schema of A-HDFS

Table 2 Mapping from the file-based metadata to relational tables

Files	Original metadata	Related tables
FSImage	File information (ID, path, blocks list, permission, owner, etc.), Block information (ID, location, creation time, etc), ACL information, XATTR information, Snapshot information, etc.	HDFS_DATA, HDFS_GLOBAL, HDFS_XATTR, HDFS_BLOCK_DATA, HDFS_DATA_ACL, HDFS_SNAPSHOT, HDFS_CACHE_DIR, HDFS_CACHE_POOL, HDFS_SS_DATA, HDFS_DATANODE, HDFS_DL_TOKEN, HDFS_MASTER_KEY, HDFS_BLOCK_DATANODE_REL
EditLog	Command information including: Command Name (e.g. OP_MKDIR, OP_DELETE) and corresponding information of each command (e.g., file path, access time)	(*) MetaManager reflects metadata changes into the corresponding MetaDatabase tables
System XMLs	Configuration information (HDFS, YARN, MapReduce, etc.)	SYSTEM_SERVICE_ACL

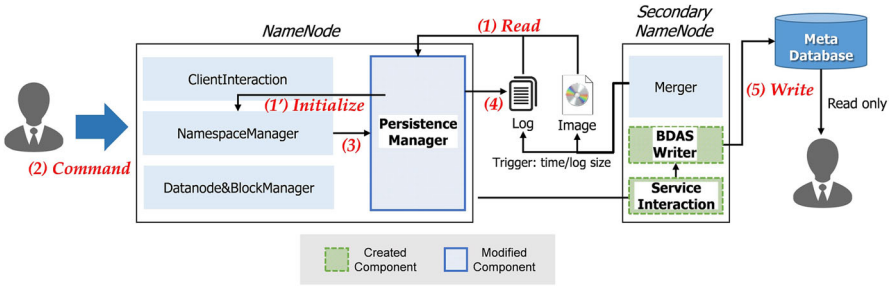

Fig. 5 Step 1: RDBMS-based metadata writing process

original metadata files and RDBMS tables. That is, we still maintain the metadata in FSImage, EditLog, and XML files, and at the same time we store the corresponding metadata into MetaDatabase. Through this duplication approach, we first confirm if the original metadata are stored in RDBMS correctly and also check if there are data losses or corruptions by comparing the original metadata with the MetaDatabase. Figure 5 shows an operation flow of the proposed Step 1 with the new or improved components, and the explanation of each step is given by (1)–(5) as follows. As in Fig. 1, steps (1) and (1′) are executed only once at the Hadoop start time.

(1) NameNode reads the metadata information from FSImage and EditLog files to start up the Hadoop system.
(1′) NameNode loads and maintains the metadata into the main memory.
(2) A client (user) inputs a command to use A-HDFS.
(3) NameNode retrieves the metadata related with the command and updates the metadata information in the main memory if necessary.
(4) NameNode writes the client's command into EditLog and transfers the change to the Secondary NameNode.
(5) Secondary NameNode backs up the modified metadata by integrating EditLog to FSImage, and simultaneously reflects the modified information to the RDBMS tables.

Prior to explaining new or improved components of Step 1, we review the original function of each module in further detail. This is for the easier understanding of each module changes that will occur in Step 1. Please refer to the major modules in Fig. 1. Their explanation is as follows.

- *NamespaceManager* manages the metadata of the current directories and files in the main memory and processes user-given commands. The corresponding major classes are FSDirectory, FSPermissionChecker, FSDirMkdirOp, FSDirRenameOp, FSEditLog, etc.
- *PersistenceManager* makes the memory-resident metadata consistent with the metadata of EditLog and FSImage files by reflecting the changes of the main memory into those files. The corresponding major classes are HdfsData, HdfsDataBlock, etc.

Fig. 6 The class hierarchy of A-HDFS modules modified or added in Step 1

- *Datanode&BlockManager* manages the DataNode and block information among the memory-resident metadata. The corresponding major classes are BlockInfo, BlockManager, DatanodeStorageInfo, etc.
- *Merger* working at Secondary NameNode checks the metadata changes of EditLog and FSImage in (Primary) NameNode and backs up those changes.

Based on the major modules above, we analyze the original HDFS process for storing metadata into files and design a new mechanism of storing data in both files and RDBMS tables together. We design the proposed system to store the metadata into RDBMS through Secondary NameNode, which is for minimizing the functional overhead of NameNode by reducing its workload. Also, we can achieve another purpose of increasing the utilization of Secondary NameNode, which is mainly in waiting state without any specific operation, by assigning table management roles to its new major functions.

We create two new modules, BDASWriter and ServiceInteraction in Secondary NameNode, and change the original PersistenceManager to store the metadata into RDBMS tables. Figure 6 shows the class hierarchy of these modules. First, BDASWriter is a new module for storing NameNode metadata into RDBMS through Secondary NameNode. This module maintains a connection to RDBMS, and it consists of two classes: (1) DbAdmin of creating tables using DDL (Data Definition Language) and (2) DbAdapter of executing SQL to store, modify, and delete the metadata to/from RDBMS. Next, ServiceInteraction is a new module for transferring command objects delivered from the NameNode to the Secondary NameNode. Here, a command object includes a command given by a client and the corresponding metadata. This module serializes command objects using Protocol Buffer [7] for transferring them between

two different processes of NameNode and Secondary NameNode, and it transfers the metadata inside the delivered command objects back to BDASWriter. Finally, PersistenceManager is a modified module that stores the changed metadata information delivered from the main memory into EditLog in NameNode and, at the same time transfers the information to the ServiceInteraction module in Secondary NameNode by modifying the FSEditLog class.

4.3.2 Step 2: RDBMS-based metadata reading process

In Step 2, we use the RDBMS-based metadata management only; that is, we delete all the original metadata files and use the RDBMS tables only. More specifically, we first store the HDFS metadata into RDBMS successfully through Step 1, and we next remove all the unnecessary metadata files from the Hadoop system. We then design HDFS operation procedures by reading the metadata information from the integrated MetaDatabase. Figure 7 shows how the client commands are processed using the integrated MetaDatabase. Compared with Fig. 5, it removes metadata files including FSImage and EditLog, and it instead manages the information using MetaDatabase in an integrated manner. The detailed explanation of each step is given through (1)–(5) as follows.

(1) A client inputs a command to use A-HDFS.
(2) Hadoop reads the related metadata information from MetaDatabase to process the command.
(2') Hadoop loads the retrieved metadata into the main memory.
(3) Hadoop delivers the metadata modified by the client command to PersistenceManager.
(4) PersistenceManager transfers the received metadata to MetaManager.
(5) MetaManager stores the modified metadata information into MetaDatabase.

As shown in Fig. 7, Step 2 is the final step for integrating metadata to RDBMS following Step 1. In Sect. 4.3.1, we see that the Step 1 procedure of A-HDFS is similar to that of the original HDFS that uses the metadata files. In contrast, Step 2 reads and writes the metadata only through RDBMS; thus, it is not necessary to read the metadata from FSImage and EditLog on starting up the Hadoop system. Also, we note that even if the metadata are modified, the following two processes can be omitted:

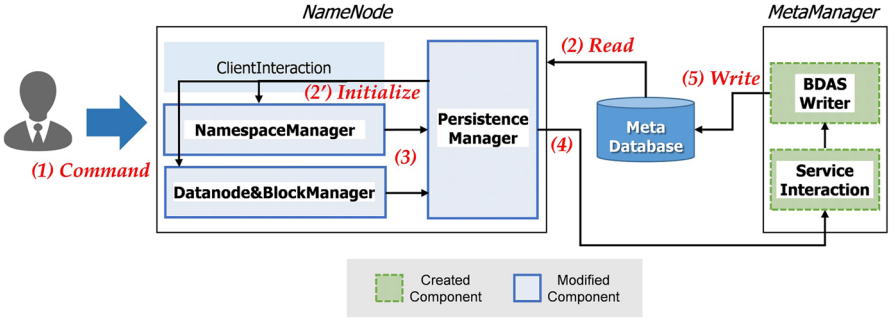

Fig. 7 Step 2: RDBMS-based metadata reading process

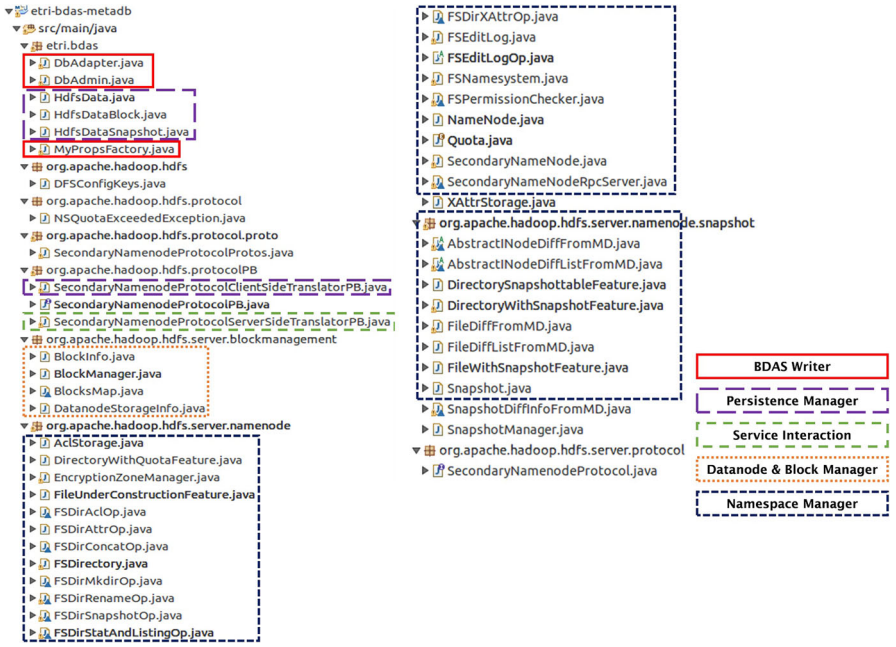

Fig. 8 The class hierarchy of A-HDFS modules added, modified, and removed in Step 2

(1) the recording process of metadata changes into EditLog and (2) the duplication checking/backup process of them in MetaManager.

To develop these operation procedures in Step 2, we modify all modules except for ClientInteraction in NameNode, and we remove Merger merging metadata files in MetaManager. Figure 8 shows the class hierarchy of A-HDFS modules added, removed, or modified in Step 2. In particular, the biggest difference compared with Step 1 is the modification of NamespaceManager and Datanode&BlockManager. The original HDFS brings the metadata information from the memory-resident iNode, which is created from the existing metadata files. On the other hand, in Step 2, A-HDFS reads all the metadata information from MetaDatabse. We do not create any new modules here, but for reading and writing information from MetaDatabase, we need to modify classes in most modules. Thus, it is effortful to reflect these modifications to the actual Hadoop source codes. In this paper, we reflect the created and modified modules described in Steps 1 and 2 to the latest version of Hadoop source code, and we confirm the proposed A-HDFS to work correctly through various experiments. We explain these experimental results in Sect. 5.

4.4 Enhancement analysis of A-HDFS

In this section, we describe how the technical elements embedded in A-HDFS through Steps 1 and 2 satisfy the three requirements. In addition, we also present how these elements resolve the existing system problems.

First, we can surely satisfy robustness by using RDBMS instead of files. RDBMS, as an already proven system of supporting ACID (atomicity, consistency, isolation, and durability) [9] properties, can ensure the stability of a transaction, which is a logical execution unit. Since the current HDFS manages the metadata based on main memory and/or files, if there are some losses of the metadata changes due to system or network errors, it is often difficult to recover the changes to the previous stable status or to redo the executions. For example, if problems or errors occur during the booting process bringing the metadata files to the main memory, or if they occur inside metadata files such as LogEdit and FSImage, the system operation itself may not be possible. Therefore, the adaptation of RDBMS with stable data storing and managing scheme is the most effective method for satisfying robustness. In particular, RDBMS makes the metadata more easily manageable by formalizing the complex metadata files. Thus, using the RDBMS-based management, we can also resolve Problem 2 (Slow Bootstrapping Problem) and Problem 3 (Complex Management Problem) related to robustness.

Second, we can satisfy the availability by managing the metadata information in separate MetaDatabase tables rather than many files. In this paper, we design a stepwise scheme completely moving all the metadata inside the existing system into an external storage space through Steps 1 and 2. According to this stepwise scheme, we show that unnecessary file search and operation processes can be omitted. During the booting process, the Hadoop system requires to load the necessary information from scattered and separated metadata files into the main memory. In this case, if the cluster size becomes bigger or the amount of data increases, the metadata size to be loaded increases; thus, the booting time also becomes longer. The proposed A-HDFS system shortens the system booting time by loading the necessary information only, and it reads and writes the necessary information only required for system operations. By this scheme, we can significantly improve the system availability for users and we can resolve Problem 2 (Slow Bootstrapping Problem). In addition, since information sharing with other environments exploiting the HDFS metadata becomes easier, it is useful to interoperate with the existing systems/applications and to develop new applications. This information sharing scheme can resolve Problem 4 (Location-aware Accesses Problem) and Problem 5 (Access Limitation Problem).

Third, scalability can be satisfied by two technical aspects: one for exploiting RDBMS and another for constructing a separate metadata management system. We can resolve Problem 1 (Metadata Limitation Problem) by the read-on-demand technique, which reads the necessary metadata only for handling a given command like Step 2, instead of using the memory-resident metadata limited by the NameNode memory size. However, this technique is difficult to realize on top of the file-based storing method. In particular, Hadoop stores complicated metadata in a single file. Thus, just searching for some required metadata may cause the degradation of system stability as well as its performance. Therefore, if we exploit RDBMS, which has expandable storage capacity and easy management for search, update, and deletion, we can operate the system without the metadata size limitation for an object. Moreover, if multiple NameNodes of Hadoop clusters in different regions can share such stable MetaDatabase, we can construct a very large Hadoop environment consisting of multiple clusters across distributed regions [8]. This means that we can extend the current

Hadoop environment to a larger scale one without location restriction. In conclusion, the proposed A-HDFS enables us to construct a scale-free Hadoop environment with supporting unlimited numbers or unlimited sizes of files and directories and to operate a number of clusters with location-unaware accesses.

5 Experimental evaluation

To demonstrate the validity and practical use of the proposed A-HDFS, we performed its actual implementation by modifying Hadoop 2.7.0. In this section, we present the experimental results on the implemented A-HDFS. The major goals of the experiments are as follows: (1) validation of the metadata maintenance ability for robustness, (2) examination of the operation of file service and metadata access for availability, and (3) demonstration of the namespace management superiority of A-HDFS for scalability.

5.1 Experimental setup

In the experiment, we configure a 24-node Hadoop cluster as shown in Fig. 9. Each node has two processors with 2.4 GHz CPU, 3 GB RAM, 1GB Ethernet NIC, and 200 GB HDD. All nodes work on CentOS 6, our modified Hadoop, and Java 1.6.0. The first 20 nodes play as Hadoop slaves (DataNode for HDFS and NodeManager for YARN), while Node 21 plays as a Hadoop master (NameNode for HDFS and ResourceManager for YARN). SecondaryNameNode (MetaManager), Key Distribution Center (KDC) for the Kerberos security, and MariaDB for MetaDatabase are installed in Nodes 22, 23, and 24, respectively.

5.2 Robustness evaluation

We first verify the robustness of the A-HDFS prototype in the aspects of corruption and recovery ability. For this, we implement a new administration interface named "hdfs metadatabase check/recover/backup" for calling database engine functions related to metadata corruption checking, metadata recovery, and metadata backup in manual or periodical modes. We think these three features can reflect the robustness of managing metadata in a file system. In this experiment, we compare A-HDFS with the original HDFS (*O-HDFS* in short) by measuring the average time required for checking corruption, backing up, and recovering the metadata. Figure 10 shows the evaluation result on corruption checking, metadata backup, and recovery processes. As shown in the figure, A-HDFS performs the corruption checking and recovering processes much more quickly than O-HDFS with the same amount of file/directory entries. The reason for this superior performance is that O-HDFS always needs to reload FSImage files and perform again all transactions in EditLog files, respectively, whereas A-HDFS just needs to access the metadata tables with the support of the database engine itself. In case of the metadata backup, however, A-HDFS is slightly worse than O-HDFS. This is because A-HDFS performs a more complicated metadata backup process compared with O-HDFS, which simply duplicates the metadata folder as the backup process.

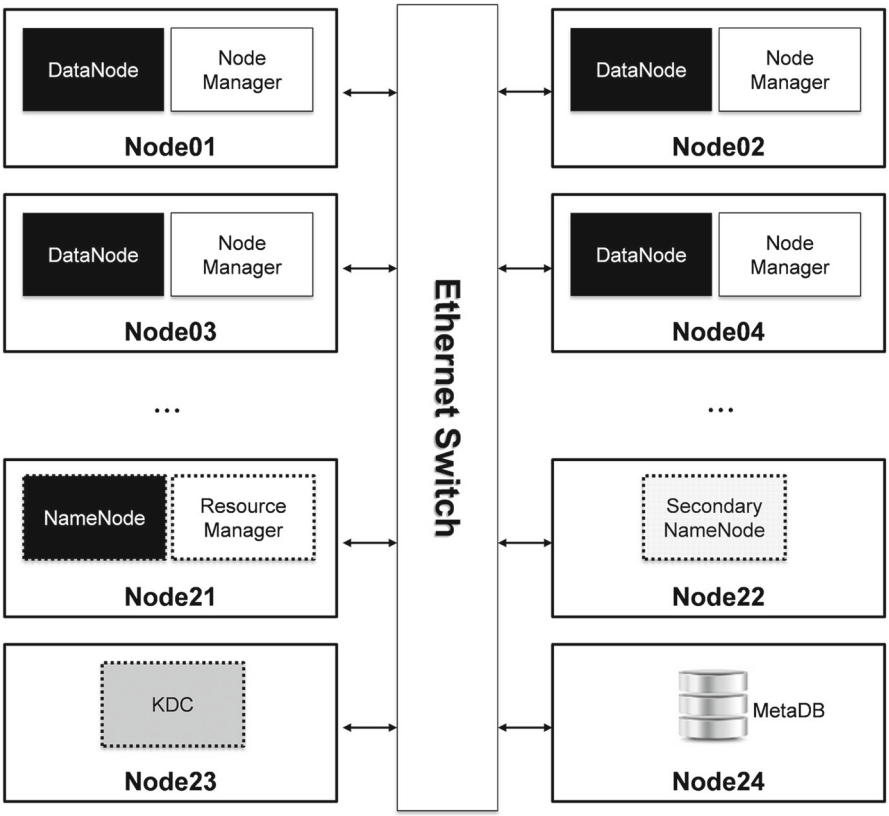

Fig. 9 Configuration of the experimental Hadoop cluster

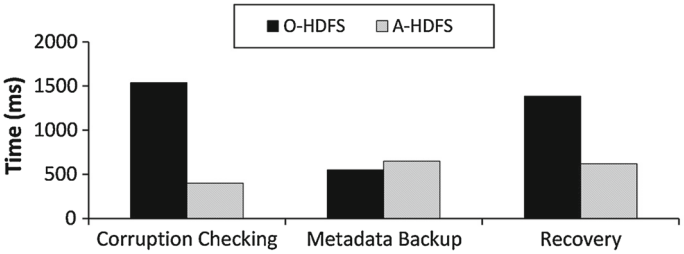

Fig. 10 Robustness evaluation between A-HDFS and O-HDFS

5.3 Availability evaluation

We next examine the availability of A-HDFS in the aspect of file service and metadata access ability. Here, we perform two experiments for showing the superiority of A-HDFS in availability. The first one is the bootstrapping time of a cluster, and the second one is the metadata access time from the third parties.

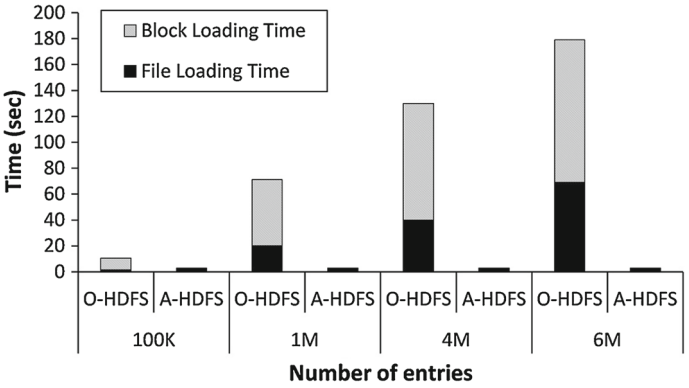

Fig. 11 Comparison of bootstrapping times of A-HDFS and O-HDFS

In the first experiment, we measure the bootstrapping time of a cluster by using A-HDFS and O-HDFS, respectively. The detailed restarting procedure is as follows. First, we upload a fixed number of file entries to the cluster. Second, we randomly change the value of block replication for each file in the file set. Third, we restart the cluster and measure its bootstrapping time. We vary the number of file entries from 100K to 6M, repeat the restarting procedure ten times for each number, and take their average as the result. Figure 11 shows the experimental result of bootstrapping times. As shown in the figure, A-HDFS significantly reduces the bootstrapping time compared with O-HDFS. This is because O-HDFS spends much time in reloading all the metadata into the main memory while A-HDFS does not need this bulk process by accessing MetaDatabase on the fly and by amortizing the bootstrapping time over the long Hadoop operational time. In particular, if the number of file entries is large, the time difference becomes much larger. This is trivial since the amount of metadata uploaded into the main memory also increases as the number of file entries increases.

In the second experiment, we measure the metadata access time from the third parties. For this, we first construct a third party application that searches files or directories by changing the search criterion to type, permission, size, and block location. We execute this third party application in three different cases: (1) O-HDFS of accessing metadata files, (2) O-HDFS of accessing metadata in main memory, and (3) A-HDFS of accessing MetaDatabase. Figure 12 shows the experimental result of these three cases. As shown in the figure, A-HDFS significantly outperforms two O-HDFS cases for all four criteria. This is because A-HDFS can exploit the indexing scheme of RDBMS, while O-HDFS does not. More precisely, O-HDFS spends much time because it has no function of searching files or directories for a given criterion and, thus, it should look at a list of all files or directories first and filter them based on the given criterion. In contrast, A-HDFS is extremely fast since it uses RDBMS indexes, which are built on important criteria. In summary, the third parties can efficiently access the metadata in A-HDFS compared with O-HDFS by exploiting the fruitful indexing features of RDBMS.

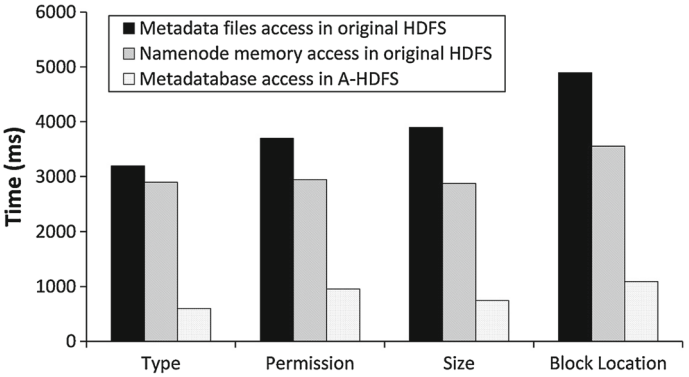

Fig. 12 Comparison of third party metadata access times of A-HDFS and O-HDFS

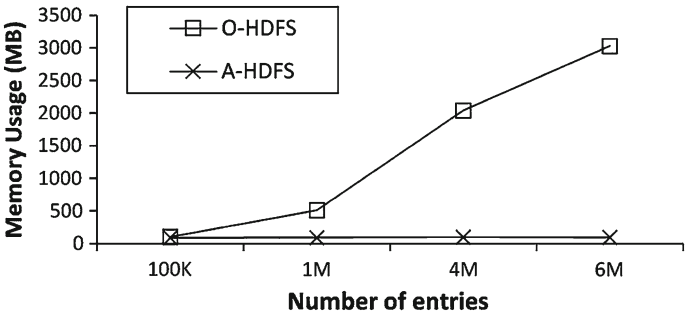

Fig. 13 Memory usage comparison of A-HDFS and O-HDFS

5.4 Scalability evaluation

We finally evaluate the scalability of A-HDFS in the aspects of namespace management. For this, we use a benchmark tool, LoadGenerator [35], which is a stand-alone tool using the DFS client libraries to stress the NameNode. LoadGenerator creates a number of threads, which run in a loop, randomly picking HDFS operations. We use it to generate a large number of file and directory entries in NameNode for memory usage comparison purposes.

Figure 13 shows the scalability test of using LoadGenerator, where we measure the memory usage by varying the number of file (or directory) entries. We note that as the number of entries increases, the amount of memory required in O-HDFS increases rapidly while that of A-HDFS is fixed to a significantly small amount. This is because O-HDFS maintains the metadata of files or directories in the main memory of NameNode, but A-HDFS does them in a separate RDBMS server. That is, O-HDFS uses the main memory of NameNode to maintain all the metadata, and its memory usage increases in proportion to the number of file entries. In contrast, A-HDFS moves the memory overhead from NameNode to a separate system that manages MetaDatabase. We believe that this approach of using MetaDatabase will increase the scalability of an entire Hadoop cluster, as well as NameNode.

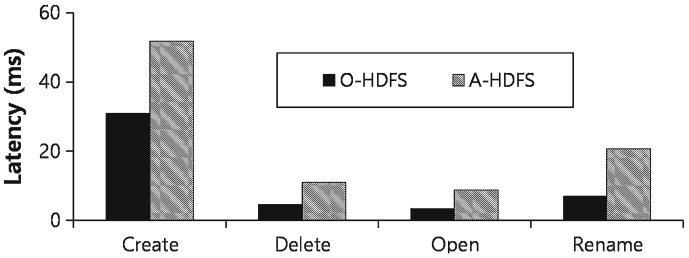

Fig. 14 Operational performance comparison of A-HDFS and O-HDFS

We also perform a metadata access test using NNBench [36], which generates metadata access requests for create, delete, open, and rename operations. For a fair comparison, we install MetaDatabase inside NameNode for A-HDFS since the original metadata of O-HDFS are resident in the main memory of NameNode. The test results show that the average metadata access time of A-HDFS is 2.42 times longer than that of O-HDFS, since it maintains the metadata in RDBMS rather than in the main memory. Figure 14 shows the detailed experimental result of all four operations. As shown in the figure, A-HDFS is 1.67 times longer than O-HDFS in create files, 2.44 times longer in delete files, 2.62 times longer in open files, and 2.95 times longer in rename files. This is an obvious result due to the use of MetaDatabase. Note that, for each operation, we need to access the metadata, which are in the main memory in O-HDFS but in MetaDatabase tables in A-HDFS. Accordingly, the memory access time in O-HDFS is evidently shorter than the RDBMS access time in A-HDFS. However, this performance degradation can be significantly reduced by optimization techniques such as (1) adopting a high-performance main memory RDBMS, (2) optimizing the database operations based on cluster configuration, and (3) caching/prefetching mechanism. However, this optimization is a separate research issue, and we leave it as a further study. Moreover, in practice, the metadata access does not incur a performance problem since a metadata access request leads to a large number of subsequent data access requests due to the write-once-read-many principle [38]. For example, to process a 1 TB data file of block size 128 MB, a client accesses NameNode just once for retrieving the metadata, but it subsequently accesses DataNodes more than 8192 times for retrieving the actual data. By this difference in metadata and data access requests, we believe that the overhead of handling metadata access requests in A-HDFS can be negligible in a practical big data cluster.

6 Conclusions and future work

In this paper, we presented a new metadata management scheme based on RDBMS for ensuring that the existing HDFS improves its stability, availability, and scalability. Using this scheme, we proposed the A-HDFS framework. The major contributions of this paper can be summarized as follows. First, we clearly defined five problems in the current HDFS and presented their concrete solutions using three requirements of robustness, availability, and scalability. Second, to satisfy the three requirements, we

introduced an overall architecture of A-HDFS and re-constructed the internal components of (primary) NameNode and MetaManager based on the real source codes. Third, by the implementation of A-HDFS and its experimental evaluation, we confirmed that the proposed system worked correctly with the enhanced functions. To our best knowledge, this is the first attempt to lay the groundwork for extending HDFS application fields by improving its functions instead of its performance. We think that this is a promising result ensuring HDFS provides its stability, availability, and scalability.

As a future work, we will investigate the performance improvement for the integrated MetaDatabase of A-HDFS. In particular, we will design a new metadata management system exploiting the in-memory technology for a more stable and fast MetaDatabase. Furthermore, we plan to develop a functionally improved Hadoop system by integrating A-HDFS and Multitenant YARN [10].

Acknowledgements This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government (MSIP) (No. 2016R1A2B4015929). This work was also partly supported by ICT R&D program of MSIP/IITP [B0101-16-0233, Smart Networking Core Technology Development] and [R7117-16-0214, Development of an Intelligent Sampling and Filtering Techniques for Purifying Data Streams].

References

1. Apache Hadoop. http://hadoop.apache.org. Accessed 26 Mar 2017
2. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache Hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, Article No. 5
3. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies, Lake Tahoe, pp 1–10
4. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, pp 137–149
5. Ghemawat S, Gobioff H, Leung S (2003) The Google file system. In: Proceedings of the 9th ACM Symposium on Operating Systems Principles, Lake George, pp 29–43
6. Kohi J, Neuman C (1993) The Kerberos Network Authentication Service (V5), RFC1510
7. Dean J, Ghemawat S (2010) MapReduce: a flexible data processing tool. Commun ACM 53(1):72–77
8. Won HS, Nguyen MC (2015) Multitenant Hadoop across geographically distributed data centers. Strata + Hadoop World, Singapore, Oral Presentation
9. Elmasri R, Navathe SB (2015) Fundamentals of database systems, 6th edn. Pearson
10. Won HS, Nguyen MC, Gil MS, Moon YS (2015) Advanced resource management with access control for multitenant Hadoop. J Commun Netw 17(6):592–601
11. IBM Open Platform with Apache Hadoop. http://www-03.ibm.com/software/products/en/ibm-open-platform-with-apache-hadoop. Accessed 26 Mar 2017
12. Apache Hadoop on Amazon EMR. https://aws.amazon.com/elasticmapreduce/details/hadoop. Accessed 26 Mar 2017
13. Cloudera Enterprise with Apache Hadoop. http://www.cloudera.com/products/apache-hadoop.html. Accessed 26 Mar 2017
14. Hortonworks Data Platform with Apache Hadoop. http://hortonworks.com/hdp. Accessed 26 Mar 2017
15. Apache Hadoop for the MapR Converged Data Platform. https://www.mapr.com/products/mapr-distribution-including-apache-hadoop. Accessed 26 Mar 2017
16. Cassandra. http://cassandra.apache.org. Accessed 26 Mar 2017
17. Ceph. http://ceph.com. Accessed 26 Mar 2017
18. Lustre. http://lustre.org. Accessed 26 Mar 2017

19. OneFS. http://www.emc.com/en-us/storage/isilon/onefs-operating-system.htm. Accessed 26 Mar 2017
20. Tracey D, Sreenan C (2013) A holistic architecture for the internet of things, sensing services and big data. In: Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Delft, pp 546–553
21. Anderson JW, Kennedy KE, Ngo LB, Luckow A, Apon AW (2014) Synthetic data generation for the internet of things. In: Proceedings of 2014 IEEE International Conference on Big Data, Washington, DC, pp 171–176
22. Hromic H, Phuoc DL, Serrano M, Antonic A, Zarko IP, Hayes C, Decker S (2015) Real time analysis of sensor data for the internet of things by means of clustering and event processing. In: Proceedings of 2015 IEEE International Conference on Communications, London, pp 685–691
23. Rathore MM, Ahmad A, Paul A (2015) The internet of things based medical emergency management using Hadoop ecosystem. In: Proceedings of IEEE Sensors, Busan, pp 1–4
24. White T (2015) Hadoop: The Definitive Guide, 4th edn. OReilly Media
25. Liu X, Han J, Zhong Y, Han C, He X (2009) Implementing WebGIS on Hadoop: a case study of improving small file I/O performance on HDFS. In: Proceedings of 2009 IEEE International Conference on Cluster Computing and Workshops, New Orleans, pp 1–8
26. Zhang J, Wu G, Hu X, Wu X (2012) A distributed cache for Hadoop distributed file system in real-time cloud services. In: Proceedings of the 13th ACM/IEEE International Conference on Grid Computing, Beijing, pp 12–21
27. Lu X, Islam NS, Wasi-ur-Rahman M, Jose J, Subramoni H, Wang H, Panda DK (2013) High performance design of Hadoop RPC with RDMA over InfiniBand. In: Proceedings of the 42nd International Conference on Parallel Processing, Lyon, pp 641–650
28. He H, Du Z, Zhang W, Chen A (2016) Optimization strategy of Hadoop small file storage for big data in healthcare. J Supercomput 72(10):3696–3707
29. Tanenbaum AS (1992) Modern Operating Systems. Prentice-Hall, Upper Saddle River
30. Rabkin A, Katz RH (2013) How Hadoop Clusters Break. IEEE Softw 30(4):88–94
31. Cohen JC, Acharya S (2014) Towards a trusted HDFS storage platform: mitigating threats to Hadoop infrastructures using hardware-accelerated encryption with TPM-rooted key protection. J Inf Secur Appl 19(3):224–244
32. Borthakur D, Gray J, Sarma JS, Muthukkaruppan K, Spiegelberg N, Kuang H, Ranganathan K, Molkov D, Menon A, Rash S, Schmidt R (2011) Apache Hadoop goes realtime at Facebook. In: Proceedings of International Conference on Management of Data, ACM SIGMOD, Athens, pp 1071–1080
33. Hua X, Wu H, Li Z, Ren S (2014) Enhancing throughput of the Hadoop distributed file system for interaction-intensive tasks. J Parallel Distrib Comput 74(8):2770–2779
34. Neuman BC, Tso T (1994) Kerberos: an authentication service for computer network. IEEE Commun Mag 32(19):33–38
35. Hairong Kuang synthetic load generator for NameNode testing. https://issues.apache.org/jira/browse/HADOOP-3992. Accessed 26 Mar 2017
36. Mukund Madhugiri NNBench for NameNode testing. https://issues.apache.org/jira/browse/HADOOP-2000. Accessed 26 Mar 2017
37. HDFS Federation. https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/Federation.html. Accessed 26 Mar 2017
38. HDFS Architecture. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed 26 Mar 2017
39. Organizations Powered by Apache Hadoop. https://wiki.apache.org/hadoop/PoweredBy. Accessed 26 Mar 2017
40. Held G (2010) A practical guide to content delivery network, 2nd edn. CRC Press, Boca Raton