The Journal of Supercomputing
https://doi.org/10.1007/s11227-019-02876-9

A write-friendly approach to manage namespace of Hadoop distributed file system by utilizing nonvolatile memory

Won Gi Choi[1] · **Sanghyun Park**[1]

Abstract
With the emergence of the big data era, various technologies have been proposed to cope with the exascale of data. For a considerably large volume of data, a single machine does not comprise enough resources to store the complete data. Hadoop distributed file system (HDFS) enables large datasets to be stored across the big data environment consisting of several machines. Although Hadoop has become a crucial part of the big data industry, because of its simple architecture which composed of master and slaves several problems such as scalability and performance bottleneck has been remained to solve. New storage technologies offer an opportunity to solve the problems and improve HDFS. We propose a novel management scheme for namespace metadata of HDFS by utilizing nonvolatile memory which has been mentioned as the next-generation device since flash memory devices. Nonvolatile memory, which can guarantee data persistence and high performance with byte-address access, alleviates Namenode bottlenecks resulting from journaling processes performed to preserve the file system's metadata. Our proposed methods show significant improvement compared with block devices such as hard disk drive, solid-state drive in terms of NameNode performance.

Keywords Hadoop filesystem · Nonvolatile memory · NameNode · Distributed computing

✉ Sanghyun Park
sanghyun@yonsei.ac.kr

Won Gi Choi
cwk1412@yonsei.ac.kr

[1] Yonsei University, Yonsei-ro 50, Seodaemungu, Seoul, Republic of Korea

Published online: 10 May 2019

1 Introduction

The emergence of extremely large-scale data has led researchers and data engineers to focus on the importance of big data technologies. Conventional database management systems are inappropriate for processing big data in terms of performance and scalability. Therefore, Apache Hadoop [2, 27], in its early years, introduced a distributed file system called Hadoop Distributed File System (HDFS), which is a leading system enabling a large volume of data to be stored in a cluster environment. HDFS forms an intuitive structure in the form of a master–slave architecture introduced in other distributed systems. Furthermore, HDFS provides interfaces and shell commands for users similar to a well-known UNIX-based file system. As HDFS provides a base environment for storing files in a distributed environment, various ecosystems, which operate based on HDFS, such as databases, query engines, and other applications, have been emerged to meet various needs of the big data industry. As a result, HDFS has become a core of big data era.

However, owing to its underlying architecture, HDFS could still face several problems including performance degradation. HDFS consists of two types of components, called NameNode and DataNode. NameNode plays the role of managing the namespace of the distributed file system. The namespace indicates system hierarchy with necessary information about inodes and data block location, which represents the information of the machine storing the specific data block. DataNode stores data blocks that make up the files and the size of the block is approximately 64 to 128 megabytes. In previous studies, researchers mainly focused on storage technologies related with DataNode in HDFS, for example, support of archival storage [12], which provides a hybrid block selection scheme by storage type in heterogeneous environment, because supporting high-storage I/O request is significant issue to improve HDFS performance and storage I/O influences overall HDFS performance in fact.

However, as the amount of data grows and the number of servers making up HDFS grows, a load of NameNode, which causes a bottleneck in the overall system, have been noticed in recent years. Recent HDFS NameNode structure supports horizontal scalability by providing an option for high availability (HA) architecture that scales from a single NameNode to multiple NameNodes [11]. HA structure also distributes an excessive number of clients requests to multiple NameNodes. Beyond horizontal scaling, however, NameNode also requires to improve the performance of the server itself. NameNode stores data in the memory for quick responses to the client's requests. However, the additional disk I/O in the journaling operation performed to ensure persistence of in-memory data reduces NameNode throughput. The primary issue with improving NameNode is to ensure that its processing performance of NameNode is not bound to the disk I/O.

The advent of new storage technologies has provided the opportunity to solve the problems as mentioned earlier. Nonvolatile memory (NVRAM) has been considered as a promising device for replacing the conventional system architecture.

As NVRAM promises byte-addressable access and high performance comparable to dynamic random access memory (DRAM) with data persistence, NVRAM has the potential to overcome the limitations of NameNode in HDFS. As NVRAM devices, e.g., phase change memory (PCM), resistive random access memory and spin-transfer torque RAM, have not been commercialized yet like flash storage, related technologies for applying these devices to the existing systems are immature; various approaches to utilize features of such devices have been recently researched in academia.

In this study, we utilized a real NVRAM device, developed by Flashtec Corporation, to improve performance in NameNode. The device consists of internal memory, and flash chips that permanently store data and an external battery that allows the flash chips to store data in memory just before the system abnormally terminates. The device has characteristics similar to other NVRAM devices and is highly valued because it can also be used in various ways if an application makes good use of its properties.

In the case of big data platform, quite a lot of studies applying NVRAM in big data in general and in HDFS have been discussed recently [1, 15, 22, 28, 29, 31]. The emergence of NVRAM requires a transformation in the existing system architecture to take advantage of the characteristics of the device. In this study, we propose methods exploiting NVRAM in NameNode to place metadata in memory into the NVRAM to alleviate journaling and recovery overhead. We proposed an NVRAM-aware NameNode in HDFS in order to draw full potential of NVRAM and provide effective data communication methods between a memory space in NVRAM and a Java virtual machine (JVM).

In summary, the contributions of this paper are as follows:

- We confirm that disk I/O generated from NameNode process can affect entire system performance.
- We propose new methods to utilize a real NVRAM device in the NameNode process of HDFS. Not many technologies have been proposed to leverage an NVRAM device in NameNode of Hadoop distributed file system.
- We determined the best suit for our proposal, and our experimental results present improvement in the benchmark target for NameNode and streaming data. Besides, our proposed model shows significant improvement in a recovery process in NameNode and provides an opportunity for vertical scalability to reduce the usage of Java heap memory.
- As the bottleneck of NameNode in the exascale big data platform cannot be ignored, our methods exploiting NVRAM to alleviate the bottleneck of NameNode can be widely applied to any HDFS-based ecosystem.

The remainder of the paper is organized as follows: Sect. 2 provides a brief overview of the necessary background knowledge, including the NameNode structure, working of NameNode process, and type of NVRAM used. In Sect. 3, we describe the basic idea behind NVRAM-aware NameNode, including the application-level memory management, Log-structured format with byte-addressability, In-memory index, and Inode read cache. In Sect. 4, we present the experimental results for the

proposed method and compare it with the results of previous HDFS with respect to the NameNode overhead benchmark, TestDFSIO which is a well-known benchmark in Hadoop experiments and realistic stream benchmark provided by Intel HiBench. In Sect. 5, we present studies related to our work. Finally, Sect. 6 concludes our paper.

2 Background

2.1 HDFS and its NameNode

An HDFS cluster consists of two types of nodes: NameNode and DataNode. In general, HDFS follows the master–slave architecture, where NameNode plays a role of managing the slave machines by storing the metadata of the entire system including HDFS namespace. When accessing a file in HDFS, the client requests NameNode to return information about which DataNode and which storage device the data block is stored in. Clients access DataNode directly based on the information. NameNode maintains all the data in memory to increase the throughput for client requests. As the memory device has volatile characteristics, HDFS requires additional procedures to store log files recording operations that modify the namespace to allow recovery when an abnormal system crash occurs.

Figure 1 describes the HDFS structure, especially the NameNode structure. NameNode manages the following three types of data structures in the memory. (1)

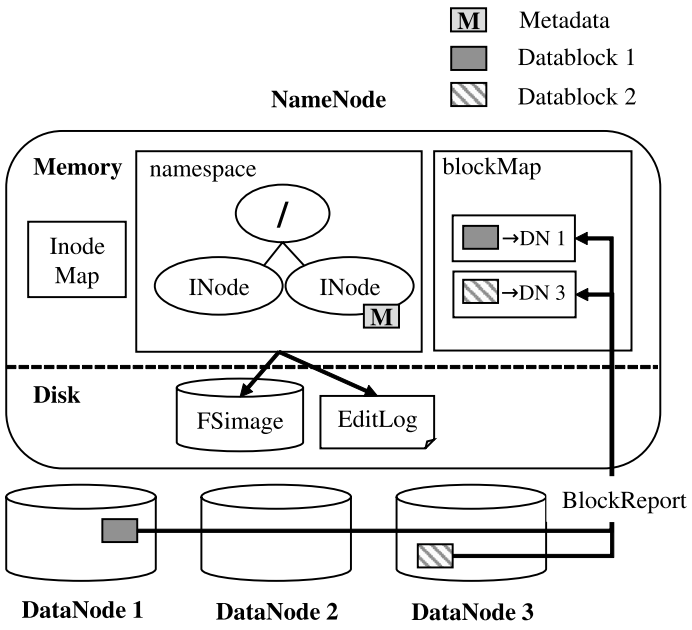


Fig. 1 HDFS structure

A namespace structure with metadata, including file or directory name, permission, and access time, which is described as a tree structure whose node is an inode in Fig. 1. (2) Block location, which represents DataNode and storage information of a data block that makes up a particular file. The block location information is stored in the blockMap structure, and the information in the blockMap is returned to the client whenever the client requests for the data block. (3) The last component is InodeMap, which is an index structure that rapidly searches for the inode.

Block location can be recovered through a block report, which is periodically received by DataNode. However, the namespace cannot be recovered because Datanode does not send any information to NameNode for the reconstruction of the namespace. Instead, NameNode records data related with the namespace and instructions to replay at the restart time to the log files in the disk. Log files are composed of two files types: FsImage and EditLog. FsImage is a binary file that involves the number of inodes, the generation time of the FsImage, and actual inode information, such as permission of inode and block information that make up the file. EditLog traces the execution operations after FsImage is created. By referencing both the log files, the namespace structure can be reconstructed when HDFS restarts. The performance of the HDFS NameNode depends on being able to flush the log data to disk relatively quickly. Figure 2 describes that delay in writing the log files holds up valuable RPC handler threads and can have cascading effects on the performance of the entire Hadoop cluster in the worst case.

Moreover, checkpoint operations associated with the log files generate a heavy load. The Namenode process executes the checkpoint operation periodically, which applies operations in the Editlog file to a new FsImage file to shrink the EditLog file

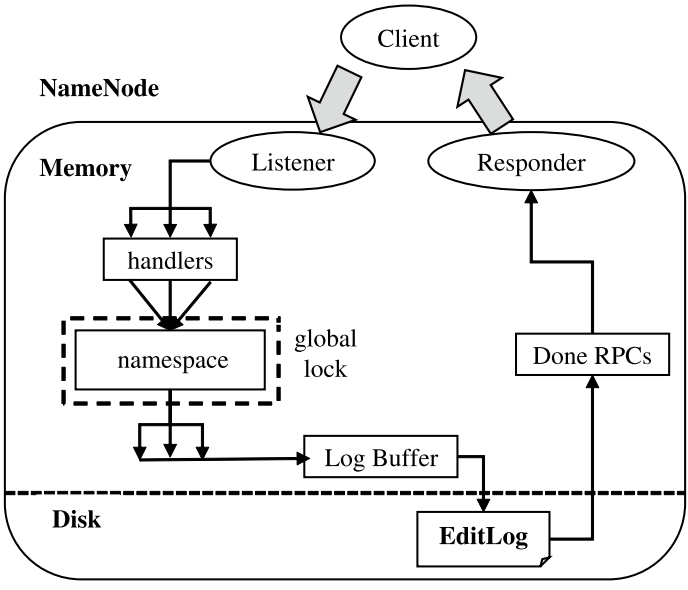


Fig. 2 NameNode procedure to process client requests

size. Without the checkpoint operation, the file size of EditLog can grow significantly and restart time to reconstruct the file system namespace is increased. As the checkpoint operation requires enormous CPU and disk overhead to affect the performance of the client, HDFS can be configured to load balance by setting a secondary NameNode to delegate a task of merging FsImage and EditLog.

In general, previous researches have noted that most performance issues in HDFS were mainly related to managing storage I/O from accessing data blocks and network performance. However, the journaling operations of NameNode can become a major performance bottleneck as the number of client requests and the number of related operations increases significantly.

2.2 NVRAM (battery-backed flash)

A promising device that generally provides low read and write latency comparable to DRAM and prevents data loss is commonly referred to as NVRAM. NV1616 we use is one of the NVRAM devices which is slightly different from conventional NVRAM devices, such as the phase change memory, which is made of new materials, while being compatible with DIMM interface. The NV1616 NVRAM drive provided by Flashtec Corporation is a PCI Express card consisting of 16 GB of DDR RAM and two flash memory banks that can each store one copy of RAM. A supercapacitor module supplies power for NVRAM backup, saving the data in RAM to the persistent storage when unexpected power-down events occur. On power-up, the RAM contents are recovered from the flash memory unit, allowing the host application to continue performing tasks.

The NV1616 NVRAM device supports two modes: block and direct memory interface modes. The block mode allows an application to access internal memory in the NVRAM through the NVMe Driver. It also serves high throughput than a standard NVMe drive which has more massive throughput than solid-state drives (SSDs). The direct memory interface can avoid the overhead of the I/O stack and directly access the memory. Applications access memory space similar

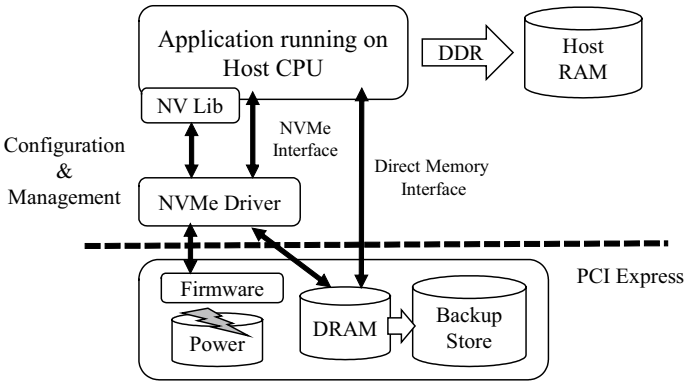


Fig. 3 NV1616 structure

Table 1 NV1616 device's latency comparison with PCM

Latency	PCM	NV1616
Read	408 ns (4B)	41 µs (512B)
Write	7.5 µs (4B)	0.23 µs (512B)

to how the CPU accesses to host memory (Fig. 3). Table 1 compares the latencies of NV1616 and PCM. [26] PCM and NV1616 have a common property: the devices are byte-accessible, and similar in performance to DRAM but have a slightly slower read and write latency than DRAM, and an asymmetric read/write performance. However, PCM has a faster read performance than NV1616, while it has a faster write performance. Although we verify our proposal through the NV1616, it is not limited to NV1616 because most types of NVRAM have a common feature of byte-addressability and high performance with non-volatility.

3 NVRAM-aware NameNode

Figure 4 describes the overview of the proposed NVRAM-aware NameNode architecture. As the inode object is small in size in bytes and the device capacity is in gigabytes, the device can accommodate multiple inode objects required in a big data environment. We exploited the NVRAM device to preserve data that cannot be recovered without log files. The metadata of HDFS NameNode is stored in a proposed data structure, not the log format, to utilize in the HDFS application. This approach improves NameNode performance by reducing data redundancy that exists between data and log files. The NVRAM-aware NameNode can drive full potential from NVRAM because the size of the data to be considered is in bytes, and the device supports byte-addressable and has low latency. Besides, data extensions to NVRAM provide the opportunity to support the vertical scalability of NameNode, which can be fatal because of the lack of Java heap space in the big data environment.

Serializing of the entire inode object to a byte array and passing of the array to a device through the native IO byte buffer function provided by Java library is extremely heavy to be used in the NameNode process that must provide the result to the client in real time. In addition, the serializing of all member variables in the inode object must be avoided because Java object in JVM refers and contains other objects that do not need to be stored in the NVRAM. For example, while DataNode and storage information is stored in blockMap, inode Java object also contains the information. To reduce the overhead from the serialization of the objects and eliminate objects that do not need to be stored in NVRAM because of the redundancy with existing objects in memory, we distinguished and extracted unrecoverable variables of NameNode process in JVM without using a separate block report. The variables in NVRAM can be used to make up a new inode object so that the existing NameNode process can utilize. NVRAM-aware NameNode also allows fine-granularity access to the variable in NVRAM,

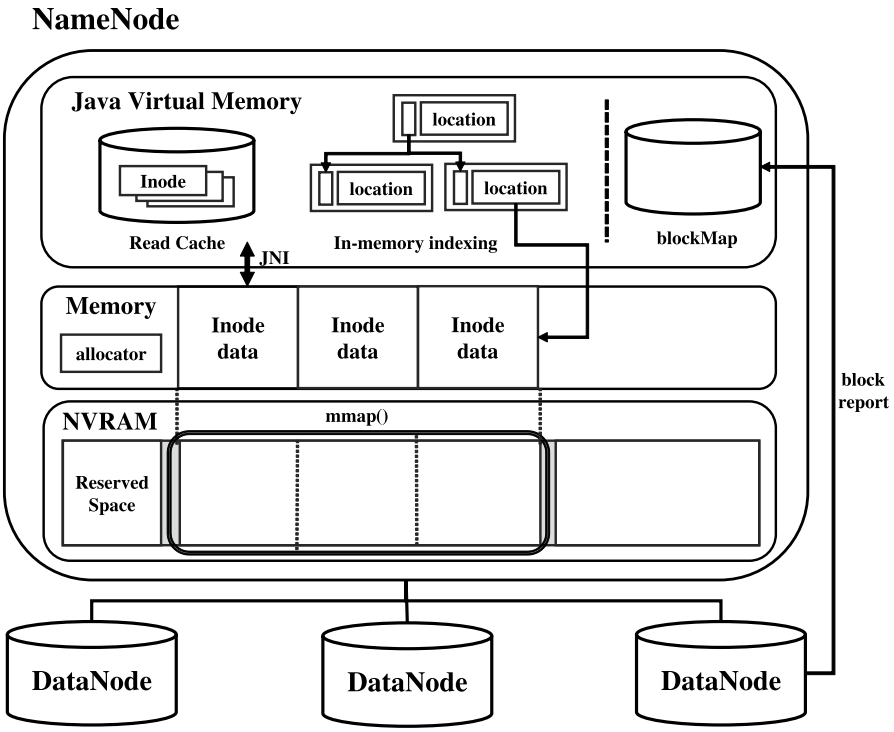


Fig. 4 Overall structure of our NVRAM-aware NameNode architecture

because HDFS often requires metadata in units smaller than the contents of the entire inode.

While proposing NVRAM-aware NameNode, we confirmed that several performance degradation issues could exist. First, the allocation of memory space in NVRAM by utilizing system calls whenever HDFS requires space for storing an object is inefficient because of the heavy load of system calls. Therefore, we suggest the allocation of more NVRAM memory than is necessary and memory management at the application layer.

Second, as HDFS is implemented in Java language and runs on JVM, it requires an interface to connect with a system-level language like C to exploit the NVRAM device. Therefore, we require Java Native Interface (JNI) functions to transfer data between JVM and system memory. The NVRAM-aware NameNode transforms from a Java inode object to an innovative structure in NVRAM to utilize the NVRAM device through the JNI functions. However, frequent JNI calls must be avoided because they can degrade the performance owing to transformation cost from Java to C. Thus, we propose an in-memory index to reduce the number of JNI calls required for lookup and propose a read cache for reusing the inode Java object frequently used to avoid unnecessary JNI calls.

Third, the approach of merely mapping a Java object to NVRAM is very naive to draw the full potential of an NVRAM device, which supports byte-addressability. In this study, we drew all the potential of the NVRAM device, providing an interface to access fine-grained data that support modification of member variables, such as file permission, in the inode. For variables that do not support atomic updates, a unique logging scheme is proposed.

In summary, we propose a novel scheme to manage the information of the inode in NVRAM and techniques to resolve the issues generated from unique features in the programming language and NVRAM characteristics.

3.1 Log-structured inode in NVRAM with byte-addressability

We propose a page-sized data structure called inode page structure to store and manage an inode object inside a JVM in the NVRAM which is described in Fig. 5. First, an inode page structure in NVRAM includes a type variable that indicates whether the inode type is a file or directory. The structure is slightly different depending on the inode type, file or directory. Besides, the inode page structure commonly stores necessary information including inode id, modification time, access time, and permission and NVRAM offset of a parent inode in the namespace hierarchy used for the recovery process. In case of the file, the inode page

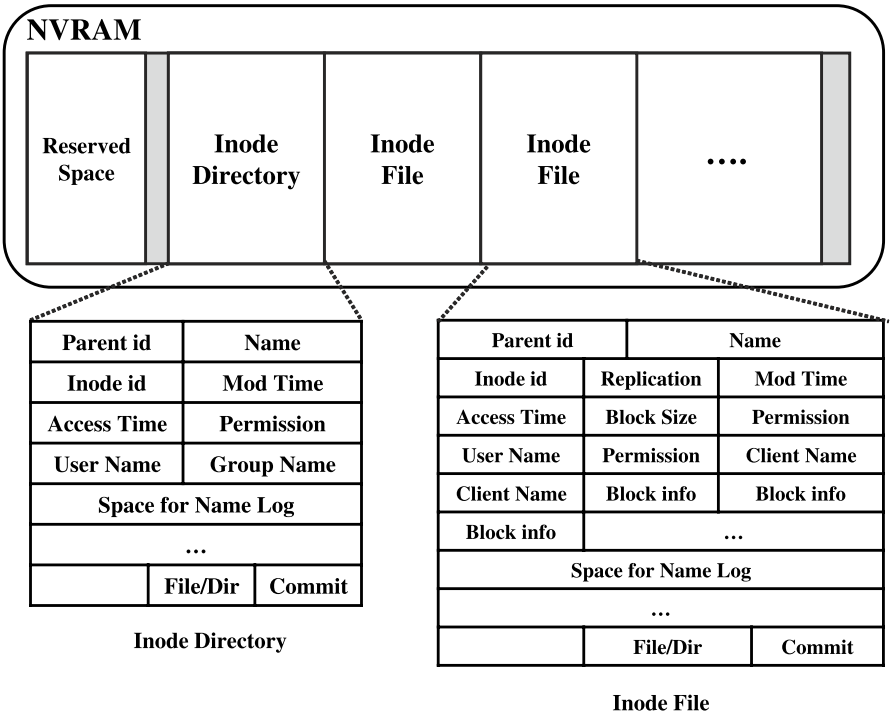


Fig. 5 Log-structured format in NVRAM

structure also stores the size of data blocks, replication factor, machine name and IP address of the client as well as block information comprising id, byte length of block name, and generation timestamp. Also, a commit flag is placed in the page structure to determine the validity of data in the structure.

These inode page structures are arranged so that appending is performed similarly to a log-structured file system. HDFS can create a new Java object by referencing the data in a valid inode page structure based on the commit flag. Commit flag is also used to invalidate deleted inodes. In the recovery process, HDFS namespace can be reconstructed based on valid inode pages which commit flag is set. In the case of an operation that creates a complete inode, an out-place update is applied through the commit log, but fine-grained and in-place update operations are also supported for modifying an individual member variable of the inode. As a modern processor supports 8 bytes atomic writes natively and most variables, except for the string type, in the data structure are under 8 bytes, member variables can be updated atomically.

As HDFS often updates and inquires single metadata, we implemented a fine-grained update interface to leverage the byte-addressability of NVRAM. Byte-addressability is a unique feature of NVRAM compared with other block-based devices. The satisfactory usage of byte-addressability can improve HDFS performance because this allows the reduction in the amount of data written to the storage and preventing redundancy of data. As inode page structures are written to NVRAM in an out-place update manner, no duplicated copies are required for data persistence except for commit log. If a valid commit log is not set, HDFS determines that the inode page structure is not valid for constructing an inode Java object to use in JVM.

However, NVRAM-aware NameNode supports atomic in-place update to modify member variables in an inode page structure because HDFS demands a considerable amount of requests to modify only a small data and not the entire inode structure. An appropriate example could be the modification of the access time of a file or directory permissions. The entire inode structure does not need to be written newly to resolve the requests for small-sized updates. The reflection of the changes of small-sized modifications can be processed by replacing the old value with the new value as the size of most data managed by NameNode is relatively small, that is, 8 bytes or less, thus ensuring atomicity in modern processors.

We propose a partial logging scheme because an additional physical log is required for data size of more than 8 bytes. For instance, in the case of the renaming

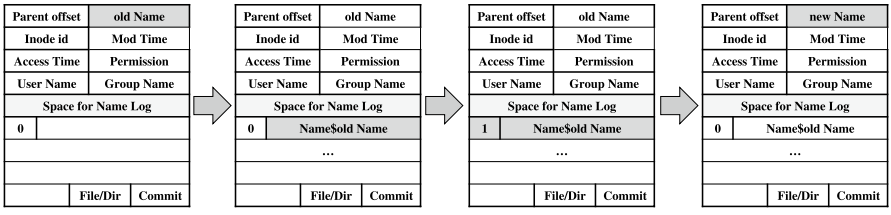


Fig. 6 Partial Logging Method in inode page structure

operation, the preservation of the old image is necessary for recovery in the event of system failure because the size of file or directory name in HDFS is generally over 8 bytes. Figure 6 describes how to perform the partial logging process when a client renames the inode. The inode page structure has a separate space to store the undo log for the string type such as inode name, user name or group name. Firstly, the inode page structure saves a record which includes old string and variable name before updating the actual inode name. Then the structure set a valid flag for the record, which guarantees commit status of the record. After setting the flag, the inode name is updated to a new name. When the data is corrupted because of a crash during the writing operation, the name is recovered to the previous name applying the undo logs.

For data with size greater than 8 bytes, additional writes are necessary to prevent data loss. However, as the size of most data is under 8 bytes and log size is still significantly smaller than that required for writing the entire object, our proposed partial logging scheme that we proposed is efficient. In summary, the use of partial logging scheme also reduces both JNI calls and amount of data written to improve entire performance.

3.2 Application-level memory management

If the direct memory interface mode of the NVRAM drive is selected, the application can receive the address at which the NVRAM drive map to application memory by using mmap function. When Hadoop attempts to store data in NVRAM, it requires the memory address obtained from the NVRAM driver by opening NVRAM and allocating memory space. As a system call is costly, frequent calls will be a significant factor in the overall performance degradation whenever HDFS requires space for data allocation. Therefore, HDFS must avoid unnecessary calls to open/close a device and allocate memory space.

Kim et al. developed a user-level NVRAM management scheme on SQLite, widely used in a mobile environment, to alleviate the overhead from system call invocations. [18] We also implement a simple NVRAM management module for HDFS. The NVRAM-aware NameNode allocates as much memory space as possible in advance, stores address values in JVM, and manages the memory space at the application level. In addition, the pre-allocation and management of space at the application level could alleviate the overhead from JNI calls. Otherwise, HDFS must transform the language from Java to C through JNI call to access the driver's function and request memory allocation.

However, mapping an entire device to virtual memory by using mmap function is impossible and can retain an unstable state at the system level. The NVRAM-aware NameNode manages memory space by storing the initial address sets of memory space allocated to the maximum extent possible by using the mmap function in the form of an array.

3.3 In-memory index implementation

In an NVRAM-aware NameNode, the primary factor involved in boosting the performance is to reduce the number of JNI calls, and NVRAM write operations.

Although a native code is generally much faster than the code executed in JVM, JNI calls which transform of Java code to the native code takes more CPU cycles, and thus frequent calls could affect the entire system. As the objects in HDFS are created in JVM, JNI call is necessary to save the objects in the system memory area which maps NVRAM device. However, unnecessary JNI calls induce NameNode performance degradation.

If all our data structures are implemented in NVRAM in the form of tree-like index, additional JNI call is required to search the data in the NVRAM. HDFS causes lots of traverse operations in the file system namespace such as retrieving a list of files in specific directories. In order to retrieve the target information, the data required for the retrieval procedure of the NVRAM data structure must be fetched through the JNI call, which degrades performance.

In addition, as the write latency in NVRAM is slightly slower than DRAM latency, storing data in a complex data structure that requires maintenance cost have to be avoided. If the data related to Hadoop variables are stored in a B+tree node or hash structure, additional writes in the NVRAM can be generated to satisfy structural constraints. For example, data in a B+tree node have to be sorted by keys. Furthermore, if the stored data amount increases and node capacity is exceeded, the tree index would have to be rebuilt by using the split operation, which generates extra writes, to balance the overall structure. Even in the case of a hash structure, extra writes are required to resolve hash collision and perform rehashing if the number of data increases. In addition, extra native interface calls for language transformation to access device is required in order to perform additional write requests to NVRAM.

In contrast, a log-structured scheme, which follows the append-only update strategy, is simple and does not require additional maintenance cost for the entire structure. However, this scheme results in unsatisfactory read performance because an inefficient process, such as sequential pattern searching without index structure, will be performed to find the desired data. Therefore, we propose a log-structured structure in NVRAM with in-memory index implementation to avoid additional access to the NVRAM.

As NameNode of HDFS requires fast response against client requests for block location, read performance in NameNode is critical in the big data environment. The original HDFS manages the inode information by developing an object tree, which maintains namespace hierarchy and includes data related to the inode. HDFS finds specific inodes by traversing the tree based on the name. In the case of NVRAM-aware Namenode, we implemented in-memory index with the variation of existing inode object tree structure in HDFS. A node in the inode object tree structure has a name in bytes and location information as the key and value. The name of the inode

is utilized in search and traverse operation in the namespace hierarchy. Location information refers to the address in the log-structured scheme on NVRAM which stores the entire data of the inode. In summary, search operations are executed in the in-memory index in JVM and scan operations on the actual data are executed from the log-structured scheme on NVRAM.

As the in-memory index is implemented in JVM, the information related to address mapping is not guaranteed to be persistent during the occurrence of the system failure. However, the in-memory can be reconstructed by referencing data in NVRAM during the recovery process.

3.4 Inode read cache implementation

The HDFS frequently accesses namespace data to obtain the information about file or state indicating whether write process is complete even though HDFS performs write-intensive workloads. Moreover, clients of HDFS tends to frequently access a specific inode, such as directories which have higher hierarchy level in the namespace. HDFS requires additional cost because the stored data must be fetched through JNI to create a new inode object from data in the NVRAM. For the write operation in HDFS, the cost of JNI can be offset based on performance gained by eliminating the cost of logging. However, in the case of HDFS read operations, a method is needed to offset the overhead from JNI.

We proposed a read cache of inode to boost read performance. We implemented a read cache that utilizes the linked hash map structure to store frequently accessed inodes into JVM. When a client sends a read request to receive the specific inode information, the data is retrieved from the NVRAM if read cache does not exist, even if the last accessed inode was required. To avoid inefficient access to NVRAM, HDFS returns the inode object for the read request and stores the object in the cache for reusing at a future request.

A linked hash map is a complex structure based on a hash structure with linked components. As a hash structure is constructed by using a name-based key, inode can be found in the read cache by using the name in constant time. The inodes are also linked for performing cache feature of victim eviction. The cache requires replacement policy to acquire space for allocating a new inode component because the cache has limited capacity. Thus, the NVRAM-aware NameNode replaces an inode in the read cache based on the least recently used (LRU) policy utilizing the linked list structure. LRU policy is used because the HDFS repeatedly tends to request information about the accessed files and directories. The use of the read cache improves the read performance of the NVRAM-aware NameNode by reducing the number of JNI calls and relieving a load of garbage collection (GC) due to the frequent creation of Java object.

3.5 Operations in NVRAM-aware NameNode

This section describes the implementation of the primary operations of the NVRAM-aware NameNode architecture. The primary operations include writing the inode to the namespace, updating the block information, retrieving the block information, removing the inode from the namespace.

Algorithm 1:
How to create INode in NVRAM-aware NameNode

Result: Write INode child in NVRAM and Update in-memory index

```
1  Flow in addFile function in HDFS NameSystem;
   /* save path to target's parent INode                              */
2  INodeInPath iip = resolve(rootDir);
3  INode parent = iip.getLastINode();
   /* initialize with name and NVRAM location                         */
4   INode target;
5  target.setParent(parent);
6  parent.children.add(target);                    // children is INode list
7  if target.isFile() then
8  |  addChildToNVRAM(FileMeta);
9  else
10 |  addChildToNVRAM(DirectoryMeta);
11 end
```

Algorithm 1 shows how to create the inode objects and store the objects to the NVRAM device. In HDFS startup, NameNode creates a root inode which indicates root directory and maintains the inode in the Java heap memory. All inodes in the proposed NameNode utilized as index entries have only their own inode name, a list of child inodes and an address in NVRAM device storing inode's metadata and log data. The inodes in the proposal are simplified compared to the existing inode model. An in-memory index is constructed by inserting inodes from the root into the target location.

Above algorithm is implemented in addFile function of NameNode which is executed by file creation instruction. The procedure for creating a directory is also similar to the algorithm. Line 2 shows a traverse method which accesses the target's parent inode. Namespace navigation starts at the root directory and is executed by binary search algorithm based on the inode name in byte format. When NameNode completes accessing the parent node of the target, an inode object for the target is initialized with the inode name and NVRAM location (Line 4). Line 6 shows that the target inode is inserted into the parent's child list. Lines 7–11 describe that inode's data is stored in the page structure in NVRAM which is denoted in 3.1 according to the inode's type. In summary, when the new inode is allocated, new index entry which has name and NVRAM address information is inserted into in-memory index, and remained metadata for the inode is stored in NVRAM in the form of the page structure.

Algorithm 2:
How to update Block information in NVRAM-aware NameNode

```
   Result: Update Block information to INode NVRAM and BlockMap
1  Flow in addBlock function in HDFS NameSystem;
   /* save path to target INode                                          */
2  INodeinPath iip = resolve(rootDir);
3  INode target = iip.getLastINode();
   /* commit block with update block size                                */
4  setBlockBytes(target.location,lastBlock.bytes);
5  Block blk = new Block();
   /* write block's meta data to NVRAM                                   */
6  addBlockToNVRAM(target.location, blk.id, blk.gentime);
   /* add block information to BlockMaps, block information contains DataNode
      location                                                           */
7  BlockMaps.addCollection(blk);
```

Algorithm 2 shows how to update the block information of the file. Algorithm 2 is implemented in addBlock function which is executed when the data stream is open and the data is written to HDFS. Algorithm 2 begins by committing the last block in the data stream. The commit process modifies the state of the last block, and the size of the block is stored. The setBlockBytes function takes advantage of byte-accessibility and sets information about the block size directly to the address in NVRAM which is allocated to the target file. A block object is newly allocated and metadata including block id and generation timestamp is stored to the NVRAM structure (lines 5–6). A blockMap that maps blocks to information about DataNodes and storage is updated for the new block (Line 7). The block object is used as the key of the blockMap, and the value for the key contains the information, which is maintained in Java virtual memory. The blockMap remains in the Java virtual memory and keeps the block information through the block report continuously, and returns the necessary information when a request is received from the client.

Algorithm 3:
How to get Block information in NVRAM-aware NameNode

```
   Result: Get Block information including DataNode location
1  Flow in getBlockLocations function in HDFS NameSystem;
   /* save path to target INode                                          */
2  INodeinPath iip = resolve(rootDir);
3  INode target = iip.getLastINode();
   /* get block meta data from NVRAM                                     */
4  long blk_id, blk_gentime, blk_bytes = getBlockInfoFromNVRAM(target.location);
5  Block key_blk = new Block(blk_id, blk_gentime, blk_bytes);
   /* get block information from BlockMaps                               */
6  return(getDataNodeInfo(BlockMaps.getCollection(key_blk)));
```

Algorithm 3 shows how to retrieve the information of the data blocks to view the file contents. Algorithm 3 is implemented in getBlockLocations function which

is executed when clients require the contents of files. Clients directly access DataNodes by referring to the information of the data blocks returned from the function. The contents of the file are composed of the data blocks obtained from DataNodes. When the client request to view the file, NameNode gathers the list of blocks retrieved from NVRAM. NameNode creates a block of metadata stored in NVRAM for use as a key to blockMap (lines 4–5). Finally, NameNode uses the blocks as a key to refer to the value found in the blockMap and returns information of the DataNode including the storage information to the client (Line 6).

Algorithm 4:
How to remove INode in NVRAM-aware NameNode

```
   Result: Remove INode from NVRAM and in-memory index and update BlockMaps
 1 Flow in delete function in HDFS NameSystem;
   /* save path to target INode                                          */
 2 INodeinPath iip = resolve(rootDir);
 3 INode target = iip.getLastINode();
 4 INode parent = target.getParent();
 5 parent.children.remove(target);
   /* set invalid Flag to NVRAM                                          */
 6 setRemoveFlag(target.location);
 7 if target.isFile() then
 8 |   blockList = getBlockListFromNVRAM(target.location);
 9 |   for Block block in blockList do
10 |   |   blockMaps.deleteBlock(block);
11 |   end
12 else
13 |   for INode inode in target.children do
   |   |   /* This function is called recursively                       */
14 |   end
15 end
16 target = null;
```

Algorithm 4 shows how to remove a particular inode from the namespace. Algorithm 4 is implemented in delete function which is executed when clients delete files or directories. The delete operation is divided into two parts: Deletion of the inode from the in-memory index and NVRAM structure, deletion of data blocks in blockMap. When the NameNode receives requests for the delete operation, NameNode traverses to the parent of target inode and delete the node from parent child list (lines 2–5). Then NameNode sets an invalid flag to the page structure of the node in NVRAM. (Line 6) If the type of the target inode is the file, blocks included in the file will be deleted in blockMap. If the type of the target inode is the directory, the above sequence will be performed for the child nodes in the same way until the child is a leaf node in the index structure (lines 7–15).

4 Experimental results

4.1 Experimental settings

We analyzed our proposed model in the Hadoop environment. For the first time, we presented a benchmark related to NameNode directly affecting NVRAM. We also experimented by using TestDFSIO provided by Hadoop to examine the application of the NVRAM-aware NameNode in the real big data environment. Furthermore, we performed a streaming benchmark with Apache Storm to verify the full potential of the NVRAM-aware NameNode.

We constructed an HDFS with nine clusters: one for NameNode and eight for DataNode. As described in Table 2, a cluster has a quad-core CPU and a 64 GB RAM. Clusters are connected with 1 Gbit Ethernet, and each cluster consists of three HDDs and one SSD. Flashtec manufactures the NVRAM device used in NameNode with a capacity of 16 GB for implementing NVRAM-aware NameNode.

To verify the effectiveness of NVRAM-aware NameNode in this section, we performed experiments by mounting a repository that records logging data for NameNode, such as FsImage and EditLog, on the HDD, SSD and NVMe which is block device mode of the NVRAM device. We compared the experimental results with our proposal denoted as NVM-NameNode. We confirmed that the logging load affects the performance of the NameNode, and it is a major factor affecting the overall system.

4.2 NameNode benchmark

Hadoop provides two types of benchmark targets for measuring the impact of various operations to NameNode. The first is the NNBenchWithoutMR, which is a single-thread Java program using HDFS client libraries to perform operations of the file system. All the operations are blocking operations, and no calls are issued asynchronously. In other words, the next operation cannot proceed until the previous operation is completed.

The other is NNBench, which is a more sophisticated version enabling the running of MapReduce jobs. The MapReduce framework allows the benchmark to launch multiple instances operating the workload specified by the user. For example, in the case of file creation workload, the user can set the number of map tasks, and

Table 2 Experiment environment

CPU	Intel i7 6700K
RAM	64 GB
OS kernel	Linux Kernel 3.10.0 CentOS 7
Hard disk drive	Seagate 3TB Barracuda
Solid-state drive	Samsung 850 PRO SSD 256 GB
Network	1 Gbit Ethernet
NVRAM	Flashtec NV1616
Hadoop	2.7.3 (with 9 clusters)

each map task performs file creation in the working directory of HDFS. Further, a reduce task collects the results from each map task and computes overall statistics.

Both benchmarks commonly set a small value close to zero as the file size for storing to measure the performance of a pure NameNode.

Figure 7a, b describes the result of NNBenchWithoutMR. We conducted experiments by varying the number of files to be created from one thousand to five hundred thousand. We noticed that NVM-NameNode performs 2.5 to 3.5 times faster than a conventional HDFS with namespace repository mounted on the HDD. When log files related to namespace data of HDFS are stored in SSDs, the NVM-NameNode completes file creation workload 1.6 to 1.9 times faster. As operations of NNBenchmarkWithoutMR are conducted synchronously and file size is almost zero, a significant factor affecting the elapsed time is disk I/O required for storing the log file in the forms of FsImage and EditLog associated with namespace data. When log files are stored in NVMe, the NVM-NameNode completes file creation workload 1.2 times faster.

In the big data environment, since the basic unit of execution time is enormous, even if it is improved to a small ratio, actual performance improvement is considerable. As the number of files to create increases, the performance gap also increases. Delay in a request due to disk I/O introduces a cascading effect. The

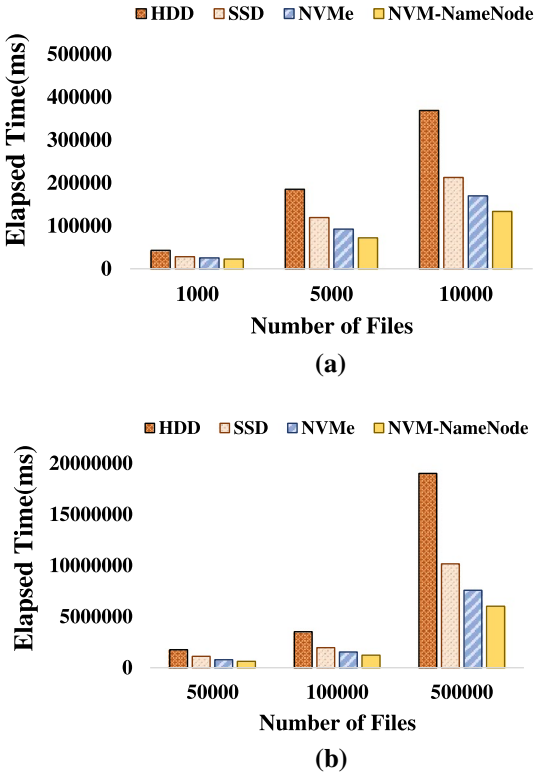


Fig. 7 **a** NNBenchmarkWithoutMR Result for cases that the number of files is from 1000 to 10,000 **b** NNBenchmarkWithoutMR Result for cases that the number of files is from 50,000 to 500,000

result from NNBenchmarkWithoutMR clearly shows that the journaling overhead affects the throughput of NameNode and our proposed structure is more effective than existing Hadoop structure.

Figure 8a, b describes the result of NNBench, which differs from NNBench-WithoutMR in that it uses the MapReduce framework to create multiple instances to load the NameNodes. NNBench first creates control files on HDFS for recording information about files to create. The number of these control files to be created is as much as the number of map tasks, and each map task generates a specified number of files referring to the control file. We conducted experiments by retaining the total number of files created and increasing the number of instances. For example, for the creation of 10000 files with 100 map tasks, as described in Fig. 8a, each map task creates 100 files to the namespace in HDFS.

In this experiment, the size of the file to be stored was close to 0 to measure the load of the pure NameNode; however, this requires an additional cost of using the MapReduce framework, including the cost of the reducer necessary for analysis.

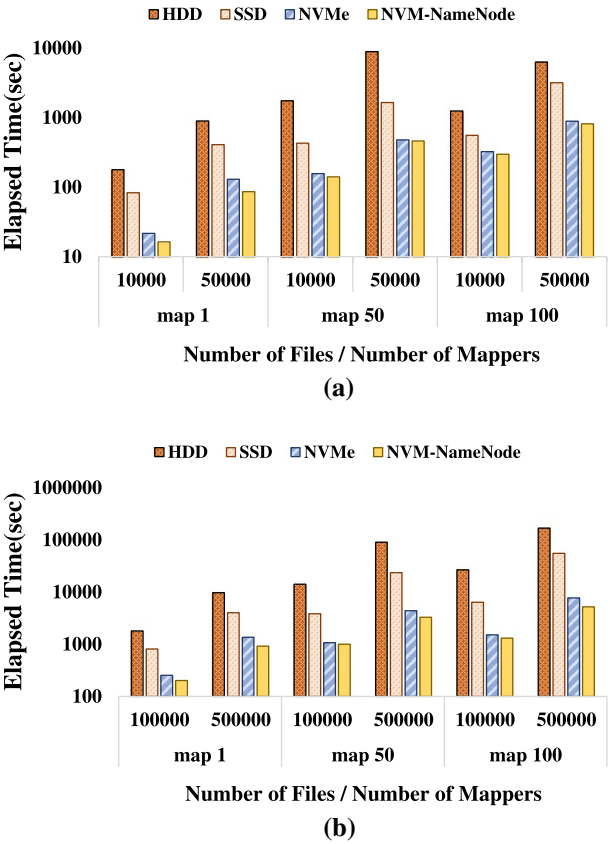


Fig. 8 **a** NNBench result for cases that the number of files is from 10,000 to 50,000 **b** NNBench result for cases that the number of files is from 100,000 to 500,000

As the number of map tasks increases, the latency increases slightly owing to contention. As the namespace of HDFS allows only one write transaction, the namespace is returned to the next write transaction after the updates in the previous transaction are guaranteed to be stored; this is to preserve data consistency. Therefore, the latency of synchronizing the log files necessary to maintain consistency affects the overall time of completing a workload.

The NVM-NameNode shows a significant performance improvement about ten times to 19 times compared to the case where the log file storage is mounted on the HDD, and it is improved from 1.7 times to 4.8 times when mounted on the SSDs. When log files are stored in NVMe, the NVM-NameNode shows 1.05 times to 1.64 times improvement in NNBench test case.

In the NNBenchmark, the number of clients to NameNode is more than a single-thread-based benchmark and the number of files to create is increased because of additional file creation for an experiment like temporary files and directory. NameNode performance degradation due to journaling is more prominent than NNbench, NVM-NameNode which does not require to flush the log to the disk shows better performance than the conventional NameNode.

4.3 TestDFSIO

TestDFSIO provided by Hadoop is a well-known benchmark for evaluating the performance of MapReduce applications. TestDFSIO produces map tasks as many as the number of files, and each map task runs an instance of the tests. A map task records the elapsed time and the size of the data that the task transfers. A reduce task collects the information from files and calculates total experimental results. As TestDFSIO generates numerous network overheads and disk I/O, the impact of NameNode bottleneck is relatively weakened. In addition, the reduce task induces numerous read operations to NameNode; this requires costs for transforming variables from system memory to the Java virtual machine.

Figure 9 shows the results of TestDFSIO. We compared the proposed NVRAM-aware NameNode with NameNode whose repository is mounted on the block mode NVRAM and SSDs. NVM-NameNode shows a slight improvement than a conventional NameNode with block mode NVRAM and SSDs. However, as TestDFSIO is susceptible to be influenced by network status and disk throughput, the experimental results are not consistent with those of other test cases. Table 3 shows the experimental results of TestDFSIO with various file sizes running in the NVM-NameNode environment.

Table 3 shows that the performance in the TestDFSIO benchmark is proportional to the file size. In other words, the total amount of data to write can be calculated by multiplying the number of files by the size of the file and the total amount of data is a crucial factor in determining the TestDFSIO result. Comparing the cases where the total amount of data is the same but the number of files is different, the results are almost identical. The experimental results demonstrate that the latency that data is transferred to the DataNodes and stored in their storage is a significant factor for the overall result. Therefore, in the TestDFSIO, the overhead of the storage or network

Fig. 9 TestDFSIO result

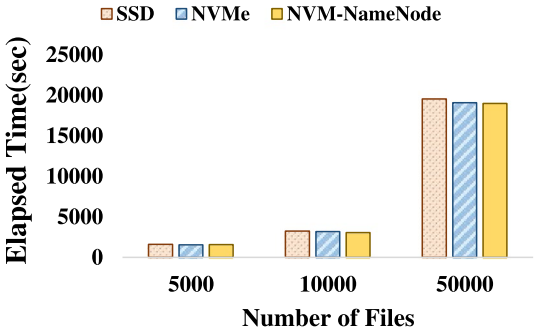


Table 3 TestDFSIO results according to the file size

Number of files	File size (MB)	Elapsed time (s)
5000	64	839
	128	1626
	256	3202
10,000	64	1659
	128	3223
	256	6576

is a more dominant factor than the number of files associated with the NameNode in performance. In general, network bandwidth is much lower than memory bandwidth or NVRAM bandwidth, so overall performance is bound to server-to-server network performance. Since Hadoop sets replicas, network costs can occur between DataNodes, and performance may be lower. Also, in a 1-gigabit network environment, network performance cannot be maintained at the same level of storage bandwidth as an SSD, which can be a factor in performance degradation.

Therefore, our proposal demonstrate efficiency when there are many clients requesting NameNode and when the size of data to be written is small.

4.4 Apache Storm environment benchmark: applicable scenario

Apache Storm [4] is a distributed real-time computation system which processes streams of data. Storm is used in many areas such as real-time analysis, continuous computation, and online machine learning. In the real-time computation environment, the efficient management of a state to maintain the idempotent of the data updates when failure occurs is a crucial issue.

Storm Trident, which is a high-level abstraction to compute real-time data provides methods for data consistency called exactly-once allowing to the storage of the Storm state in an external data store including HDFS. We constructed a Storm environment for verifying streaming benchmark provided by Intel HiBench [13, 14].

Table 4 describes the components of the environment. Apache Kafka [3] is a distributed messaging system used for producing data stream. Apache Zookeeper [5] is

Table 4 Apache Storm setting

Apache Storm	1.0.5
Apache Kafka	2.10-0.8.2.0
Apache Zookeeper	3.4.10
HiBench	7.0—word-count workload

a distributed coordination server used to manage Storm clusters. Under the supervision of the zookeepers, the Storm servers process the input data stream from the Kafka and preserve the state of the stream in HDFS. Although data processing of Storm application is conducted in memory, the state of the application is stored in HDFS, and the overall performance of Storm is inevitably affected by the HDFS performance. Since the characteristics of above workload is that small-sized data is continuously requested to be written by a large number of clients, the results from the experiment with Storm are suitable to show the efficiency of the proposed model.

Storm performs wordcount workload in HiBench by counting the number of words in the data stream. In addition, it performs operations related to counting the words in memory and stored the state of stream in HDFS to be idempotent during the occurrence of failure. From the perspective of HDFS, the storing of state workload is write-intensive. As the NVRAM-aware NameNode is a write-friendly structure utilizing NVRAM, it shows improvement compared with the conventional HDFS used in Storm benchmark.

Figures 10a, b and 11a, b show results from the Storm benchmark in HiBench. In each case, Kafka sends the data stream with a data size of 256 and 1024 bytes, for which Kafka data generator produces 5000 and 50,000 records, respectively, in one microsecond. HiBench periodically measures throughput which presents processed messages per second.

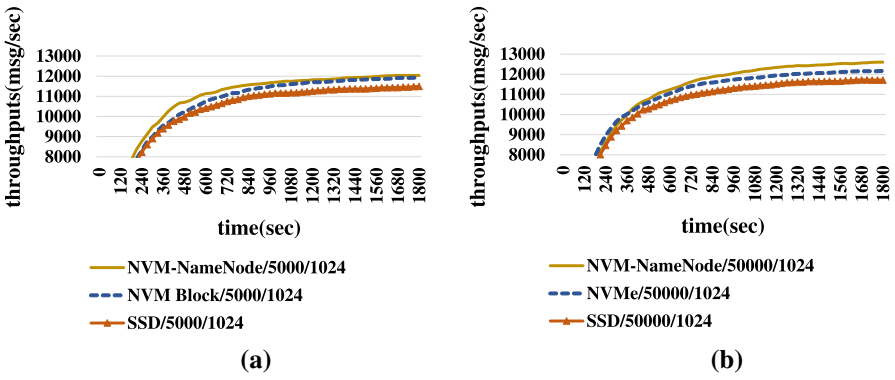


Fig. 10 **a** Storm throughput when Kafka generates 5000 records per 1 ms with 1024 bytes **b** Storm throughput when Kafka generates 50,000 records per 1 ms with 1024 bytes

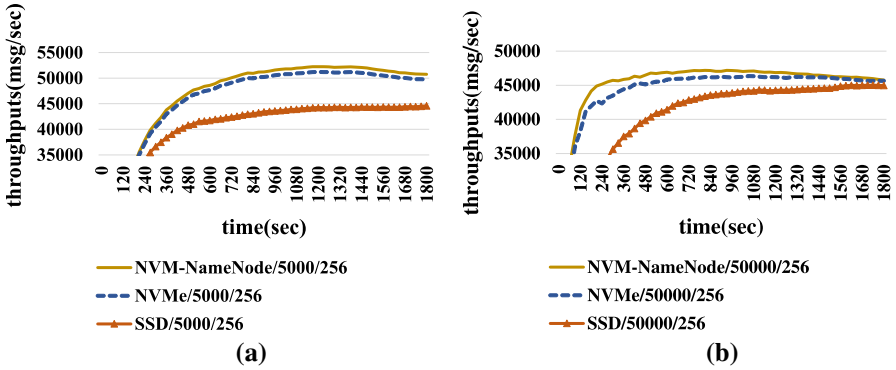


Fig. 11 **a** Storm throughput when Kafka generates 5000 records per 1 ms with 256 bytes **b** Storm throughput when Kafka generates 50,000 records per 1 ms with 256 bytes

In terms of throughput, NVM-NameNode shows an improved performance by up to 1.15 times compared with SSD mounted HDFS. In addition, NVM-NameNode shows slightly better performance than the NVMe. The smaller the data size, the greater is performance difference. As the size of data becomes small, Kafka can generate a more significant number of records simultaneously leading to increase in the number of files in HDFS.

We also measure elapsed time for NameNode operations in same test cases to show performance difference in NameNode aspects.

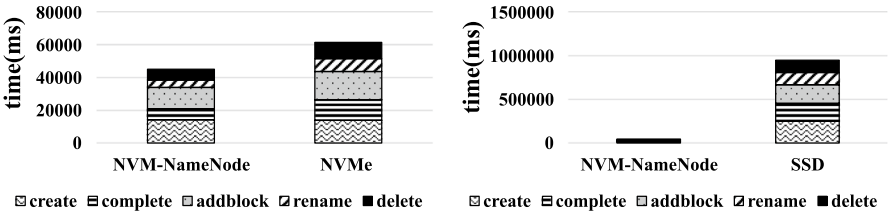
Figure 12a–d shows elapsed time for representative NameNode operations necessary to perform the benchmarks. In the case of the block devices, the time for logging procedure and sync time is included.

Create is an operation for creating an inode for a file to write. Addblock is an operation for adding block information to the specific inode. Complete is an operation related to closing the file and the write stream. Rename is an operation for renaming the specific inode, and delete is an operation related with deletion of the designated inodes.

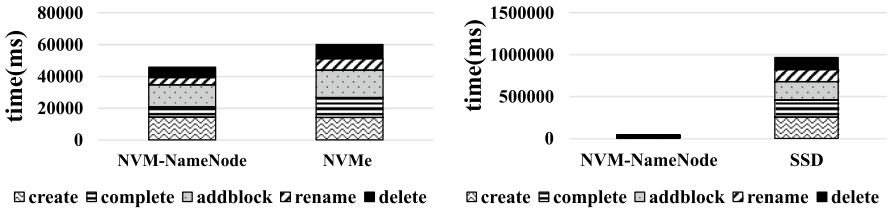
NVM-NameNode takes similar or slightly faster than NVMe in elapsed time for creation because NVM-NameNode takes times to store data for inodes to NVRAM. On the other hand, in other operations, NVM-NameNode only stores the small size of data to NVRAM with byte-addressability to perform the operations, compared with HDFS with NVMe which generates disk I/O to record modifications of file data and states. In terms of total time to process the operations, NVM-NameNode shows 1.3 to 1.6 times faster than NVMe mounted HDFS, and 15 to 26 times faster than SSD mounted HDFS.

4.5 Log rolling performance

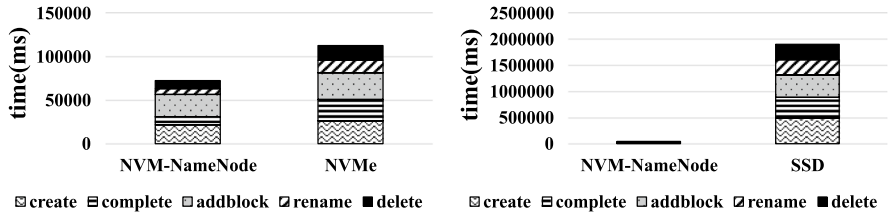
NameNode has metadata for HDFS namespace to manage the data blocks in DataNodes. If the data in NameNode becomes corrupted or loss, the entire file system becomes meaningless. NameNode supports logging mechanism called FsImage



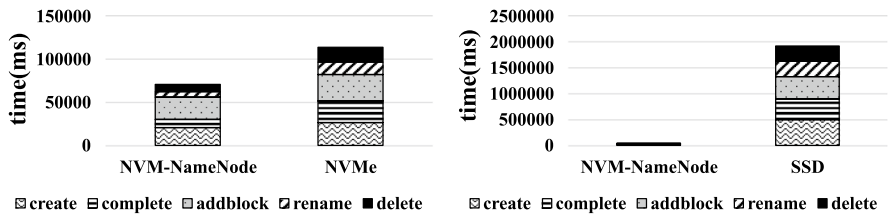
(a)



(b)



(c)



(d)

Fig. 12 **a** Elapsed time of NameNode operations when Kafka generates 5000 records per 1 ms with 1024 bytes **b** elapsed time of NameNode operations when Kafka generates 50,000 records per 1 ms with 1024 bytes **c** elapsed time of NameNode operations when Kafka generates 5000 records per 1 ms with 256 bytes **d** elapsed time of NameNode operations when Kafka generates 50,000 records per 1 ms with 256 bytes

and EditLog, to recover the system from the data corruption or crash. NameNode refers to FsImage and EditLog to reconstruct the namespace so that HDFS can be resumed. First, NameNode initializes the namespace based on the latest version of the FsImage and then perform a task called log rolling, which replays the operations recorded in the EditLog file. Log rolling algorithm is utilized in both recovery and checkpoint operation. Secondary NameNode receives the FsImage and EditLog from active NameNode and performs log rolling to make a new version of FsImage. Log rolling is a CPU-intensive and I/O-intensive operation, and performance is affected by the size of the EditLog file. If the size of the EditLog is too large, recovery and checkpointing can take a long time. On the other hand, our proposal does not require log rolling procedure because NVRAM reflects the modification of inode in real-time maintaining data persistence. When restarting, NVM-NameNode accesses sequentially from the start address of the device, rebuilding the index by referring to the address of each inode page structure, the commit flag, and the ID of the parent node.

To verify validation and performance, we measured the time it took to recover the namespace after verifying that the data that was created previously was restored after restarting HDFS. Figure 13a, b shows the performance of the recovery algorithm with log rolling in existing Hadoop and our proposal's recovery algorithm. We perform two types of experiments: recovery after NNbenchWithoutMR execution and recovery after TestDFSIO execution. As NNbenchWithoutMR benchmark simply makes a directory and creates files in the directory, building time of in-memory index in our recovery procedure is much faster. On the other hand, recovery procedure after the MapReduce jobs which record temporary directories and files for the data processing requires more difficulty in building the index.

Experimental results demonstrate that the proposal outperforms the existing Hadoop to recover using the log files in the block devices. NVM-NameNode approximately takes 3 times to 15 times faster than the SSD and NVMe in the first case. In the second case, NVM-NameNode approximately takes 4.5 times to 9.1 times faster than the SSD and NVMe. Depending on the number of files to be recovered, the existing method takes a longer time to recover, while the recovery time of the proposal is slightly increased. If the size of EditLog file is larger than in the case of the experiment, the performance gap will increase. Significant performance improvement concerning log rolling operation related to recovery and checkpoint demonstrates the efficiency of the proposal.

4.6 Usage of Java heap memory space

NameNode process is executed in Java virtual machine, and Java object which NameNode creates is managed in Java virtual memory. As the files or directories are created, inode objects and block objects are created in the heap memory. The size of the Java object for namespace management is almost 150 bytes. In the early days, because of the object's small size, a considerable number of objects could be stored in the file system.

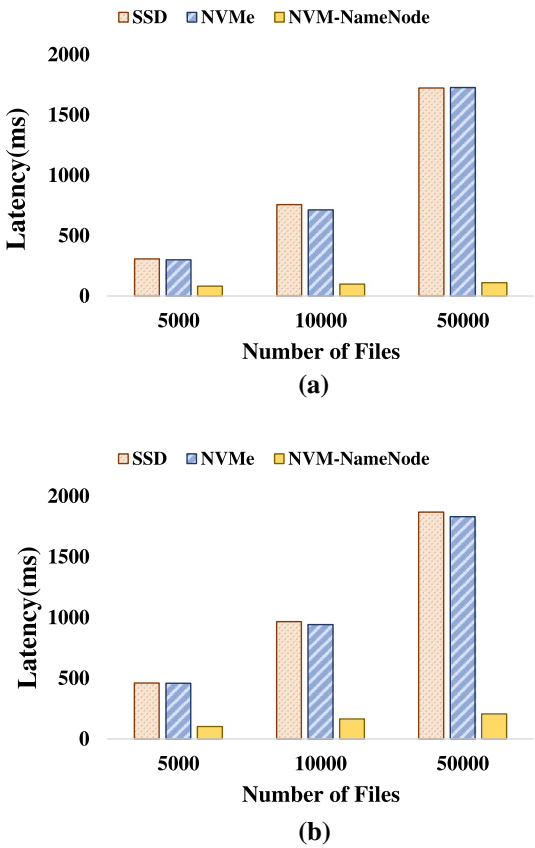


Fig. 13 **a** Recovery experiment result after NNbenchWithoutMR execution **b** recovery experiment result after TestDFSIO execution

However, as the scale of data increases in recent years, the engineers have focused on the scalability issue of NameNode. Single NameNode can shut down in worst case because it cannot bear the explosion of data. In a big data environment where service stability is essential, the system crash from out of memory is fatal.

HDFS Federation [11] supports configuring namespaces with multiple NameNodes to extend the namespace and distribute the burden of NameNode. Our proposal provides another approach to extend the namespace exploiting NVRAM. NameNode with the block devices stores the data and redundantly stores the log files that contain the history of the data updates. However, the proposal eliminates data redundancy by applying updates for data directly to the device. Only block objects which are stored in blockMap and inform the clients about the target DataNode and storage are stored in Java heap memory.

In this section, we measure Java heap memory usage by referring to the logs generated by the metric system of NameNode. Figure 14 describes the usage of Java heap memory in control group and NVM-NameNode. As the number of files to be created increases, the heap memory usage increases in both cases due to block objects. However, because the size of the in-memory index entry used in

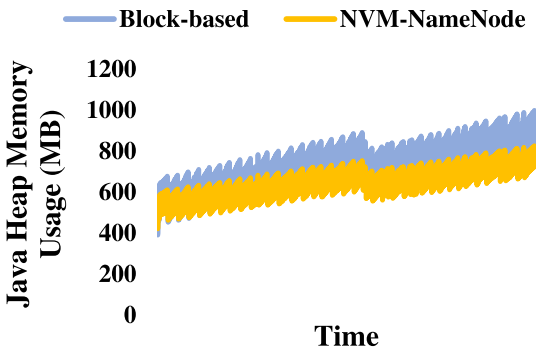


Fig. 14 Usage of Java heap memory space during the creation of 1 million files

the proposal is smaller than the original inode size, the increase ratio in NVM-NameNode heap memory usage is lower than the original method. The result does not show an ideal result of a one-half reduction in heap usage. There is a case where a new inode object needs to be created and initialized with NVRAM data when the entire inode information is required. It is challenging to gauge accurate heap memory usage due to GC, but after the creation of a million files, the difference in usage between the two experiments averaged about 100 megabytes, which is about the capacity to store about six hundred million inode objects. As the number of files to be written increases, the difference in heap memory will be maximized.

The various types of NVRAM are in development. The density of NVRAM is expected to be higher than DRAM and NVRAM are expected to be deployed at much larger scales. In contrast, DRAM faces the scalability issue. It is worth studying the scale-up method of extending namespaces using NVRAM.

5 Related works

5.1 NVRAM technology

Studies exploiting persistent memory device have emerged in various research areas [6, 10, 20, 21, 32]. Kim et al. [18] proposed a novel logging scheme called NVWAL, which is a variation of write-ahead logging (WAL) in SQLite to leverage the characteristics of NVRAM. NVWAL utilizes byte-addressability of NVRAM to store only the modified part, unlike the conventional SQLite WAL based on page granularity. In addition, NVWAL reduces the number of system calls to pre-allocate NVRAM space for logging with user-level heap management. NVWAL also reduces the number of flush cache line operations to reference the SQLite feature, which allows only one write transaction.

Arulraj et al. [7] developed a new logging and recovery protocol called write behind logging (WBL), which tracks not how a database was changed but what parts of the database were changed; this avoids redundant writes from the

conventional logging protocol by allowing DBMS to flush the changes to the database before recording them in the log file. WBL takes advantages of NVRAM allowing high performance in random writes when DBMS flushes the changed data.

Xia et al. [30] implemented a new hybrid index scheme based on both DRAM and NVRAM for key-value storage. They provided a fast point search and range search by maintaining both tree and hash indices. As the size of the hash index key is small enough to guarantee atomic update and NVRAM supports byte-addressability and low latency, a hash index is constructed on NVRAM. In contrast, a tree index is constructed on DRAM through the referencing of data by the background threads in the hash index because tree index requires additional writes for maintaining the structure, and the writes are critical to the NVRAM in terms of performance and life endurance.

5.2 Big data platform with storage technology including NVRAM

Researchers have attempted to boost the performance of a big data system as big data environment is utilized in many research and enterprise areas. Kambatla et al. [16] evaluated MapReduce performance on SSDs and HDDs in terms of cost and performance, which are significant factors for designing the data center server architecture. A hybrid block selection algorithm [17], which allows big data query engines to select block location among replications, resolves the issue by bounding the entire performance to slower disk performance. Moon et al. [23] suggested cost-effective SSD utilization methods in a Hadoop environment to leverage the potential of SSDs. Neshatpour et al. [24] proposed selective offload method that improves energy efficiency in Hadoop MapReduce environment utilizing heterogenous CPU+FPGA platform. Since satisfying the high-storage I/O is a major issue for obtaining high performance of distributed systems, early researchers [8, 9, 19] mainly focus on improving the performance of components related to storing data.

However, as the size of clusters used in enterprises has expanded in recent years, researchers have begun to pay attention to the limited scalability of NameNode. Niazi et al. proposed HopsFS [25] which is a production-level distributed hierarchical file system that stores its metadata in an external NewSQL database. Hopefs replaced the role of NameNode with NewSQL to improve the throughput of HDFS and increase the amount of metadata it can accommodate.

In recently, studies applying new storage technologies to big data platform and HDFS has emerged to reflect the needs of a variety of big data industries. Islam et al. [15] proposed a novel design called NVFS to redesign the data block management scheme of HDFS DataNodes to leverage the characteristics of nonvolatile memory and RDMA (Remote Direct Memory Access)-based communication. NVFS minimized the memory contention of I/O and computation by allocating memory from nonvolatile memory for RDMA-based communication. Yang et al. implemented Orion [31] which is a novel distributed file system for distributed nonvolatile main memory storage and RDMA networks by combining file system functions and network operations into a single layer.

This paper focuses on the journaling overhead of in-memory metadata server called NameNode, one of the bottleneck factors. With the NVRAM technology, we show improvement in HDFS performance in both general cases and recovery procedures. In addition, NVRAM-aware NameNode provides the possibility of scale-up approaches to resolve the limited capacity issue of NameNode.

6 Conclusion

Since the emergence of NVRAM as a promising device to enhance existing system architectures that utilize block devices, taking advantage of excellent performance and nonvolatile features, techniques related with the utilization of NVRAM in various areas have gained considerable attention. However, research on a Big data system with NVRAM is relatively immature because the device is not commercialized and has difficulty in applying the devices to the system architecture designed based on block devices. To leverage NVRAM requires modification of the existing system.

NameNode is suited to apply the NVRAM device with byte-addressability because the size of data units stored and maintained in NameNode is small. By using the real NVRAM device in battery-backed flash form, which guarantees data persistence, we proposed a new structure to maintain NameNode data in the NVRAM by reducing disk overhead from the operation of writing log files.

Our proposed structure is best suited for write-intensive big data workload such as playing roles of storage for saving state data of stream benchmark. Our proposal improves performance not only NameNode but also the overall system in various experiments, including the streaming benchmark.

Furthermore, our proposal doesn't require the log rolling operation utilized in recovery and checkpoint procedure, which consumes considerable amounts of CPU and disk I/O resources. NVRAM-aware NameNode shows a significant improvement in the recovery process of NameNode. Our proposal reduces the usage of NameNode's Java heap memory by eliminating data redundancy by writing data directly to the device, unlike other block devices used as log devices.

Acknowledgements This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2015M3C4A7065522).

References

1. Andrei M, Lemke C, Radestock G, Schulze R, Thiel C, Blanco R, Meghlan A, Sharique M, Seifert S, Vishnoi S et al (2017) Sap hana adoption of non-volatile memory. Proc VLDB Endow 10(12):1754–1765
2. Apache Hadoop Home Page. http://hadoop.apache.org
3. Apache Kafka Home Page. https://kafka.apache.org
4. Apache Storm Home Page. http://storm.apache.org
5. Apache Zookeeper Home Page. https://zookeeper.apache.org

6. Arulraj J, Pavlo A (2017) How to build a non-volatile memory database management system. In: Proceedings of the 2017 ACM International Conference on Management of Data. ACM, pp 1753–1758
7. Arulraj J, Perron M, Pavlo A (2016) Write-behind logging. Proc VLDB Endow 10(4):337–348
8. Bakratsas M, Basaras P, Katsaros D, Tassiulas L (2016) Hadoop mapreduce performance on ssds: the case of complex network analysis tasks. In: INNS Conference on Big Data. Springer, Berlin, pp 111–119
9. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
10. Gao S, Xu J, Härder T, He B, Choi B, Hu H (2015) Pcmlogging: optimizing transaction logging and recovery performance with PCM. IEEE Trans Knowl Data Eng 27(12):3332–3346
11. Hadoop Distribted Filesystem Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html
12. Hadoop Archival Stroage, SSD & Memory Document. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html
13. HiBench Home Page. https://github.com/intel-hadoop
14. Huang S, Huang J, Dai J, Xie T, Huang B (2010) The hibench benchmark suite: characterization of the mapreduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010). IEEE, pp 41–51
15. Islam NS, Wasi-ur Rahman M, Lu X, Panda DK (2016) High performance design for HDFS with byte-addressability of NVM and RDMA. In: Proceedings of the 2016 International Conference on Supercomputing. ACM, p 8
16. Kambatla K, Chen Y (2014) The truth about mapreduce performance on SSDS. In: 28th Large Installation System Administration Conference (LISA14), pp 118–126
17. Kim M, Shin M, Park S (2016) Take me to SSD: a hybrid block-selection method on HDFS based on storage type. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, pp 965–971
18. Kim WH, Kim J, Baek W, Nam B, Won Y (2016) Nvwal: exploiting NVRAM in write-ahead logging. ACM SIGOPS Oper Syst Rev 50(2):385–398
19. Krish K, Iqbal MS, Butt AR (2014) Venu: Orchestrating SSDS in Hadoop storage. In: 2014 IEEE International Conference on Big Data (Big Data). IEEE, pp 207–212
20. Lee BC, Ipek E, Mutlu O, Burger D (2009) Architecting phase change memory as a scalable dram alternative. ACM SIGARCH Comput Archit News 37(3):2–13
21. Lee SK, Lim KH, Song H, Nam B, Noh SH (2017) WORT: write optimal radix tree for persistent memory storage systems. In: 15th USENIX Conference on File and Storage Technologies (FAST 17), pp 257–270
22. Lu Y, Shu J, Chen Y, Li T (2017) Octopus: an RDMA-enabled distributed persistent memory file system. In: 2017 USENIX Annual Technical Conference (USENIXATC 17), pp 773–785
23. Moon S, Lee J, Kee YS (2014) Introducing SSDS to the Hadoop mapreduce framework. In: 2014 IEEE 7th International Conference on Cloud Computing. IEEE, pp 272–279
24. Neshatpour K, Malik M, Ghodrat MA, Sasan A, Homayoun H (2015) Energy-efficient acceleration of big data analytics applications using fpgas. In: 2015 IEEE International Conference on Big Data (Big Data). IEEE, pp 115–123
25. Niazi S, Ismail M, Haridi S, Dowling J, Grohsschmiedt S, Ronström M (2017) Hopsfs: scaling hierarchical file system metadata using newsql databases. In: 15th USENIX Conference on File and Storage Technologies (FAST 17), pp 89–104
26. Oh G, Kim S, Lee SW, Moon B (2015) Sqlite optimization with phase change memory for mobile applications. Proc VLDB Endow 8(12):1454–1465
27. Shvachko K, Kuang H, Radia S, Chansler R et al (2010) The hadoop distributed file system. MSST 10:1–10
28. Wasi-ur Rahman M, Islam NS, Lu X, Panda DK (2016) Can non-volatile memory benefit mapreduce applications on hpc clusters? In: 2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS). IEEE, pp 19–24
29. Wasi-ur Rahman M, Islam NS, Lu X, Panda DKD (2017) Nvmd: non-volatile memory assisted design for accelerating mapreduce and dag execution frameworks on HPC systems. In: 2017 IEEE International Conference on Big Data (Big Data). IEEE, pp 369–374
30. Xia F, Jiang D, Xiong J, Sun N (2017) Hikv: a hybrid index key-value store for dram-NVM memory systems. In: 2017 USENIX Annual Technical Conference (USENIXATC 17), pp 349–362

31. Yang J, Izraelevitz J, Swanson S (2019) Orion: a distributed file system for non-volatile main memory and RDMA-capable networks. In: 17th USENIX Conference on File and Storage Technologies (FAST 19), pp 221–234
32. Yang J, Wei Q, Wang C, Chen C, Yong KL, He B (2016) Nv-tree: a consistent and workload-adaptive tree structure for non-volatile memory. IEEE Trans Comput 65(7):2169–2183