PAPER

Job-Aware File-Storage Optimization for Improved Hadoop I/O Performance

Makoto NAKAGAMI[†a)], **Jose A.B. FORTES**[††], ***Nonmembers***, ***and*** **Saneyasu YAMAGUCHI**[†], ***Member***

SUMMARY Hadoop is a popular data-analytics platform based on Google's MapReduce programming model. Hard-disk drives (HDDs) are generally used in big-data analysis, and the effectiveness of the Hadoop platform can be optimized by enhancing its I/O performance. HDD performance varies depending on whether the data are stored in the inner or outer disk zones. This paper proposes a method that utilizes the knowledge of job characteristics to realize efficient data storage in HDDs, which in turn, helps improve Hadoop performance. Per the proposed method, job files that need to be frequently accessed are stored in outer disk tracks which are capable of facilitating sequential-access speeds that are higher than those provided by inner tracks. Thus, the proposed method stores temporary and permanent files in the outer and inner zones, respectively, thereby facilitating fast access to frequently required data. Results of performance evaluation demonstrate that the proposed method improves Hadoop performance by 15.4% when compared to normal cases when file placement is not used. Additionally, the proposed method outperforms a previously proposed placement approach by 11.1%.

key words: *Hadoop, MapReduce, SWIM, file system*

1. Introduction

Hadoop is a popular MapReduce-based big-data processing platform [1] extensively used in data-analytics applications. Intuitively, performance of the Hadoop platform can be enhanced by considering the nature of applications that require storage resources, and accelerating storage input/output (I/O) access.

In several of its applications, Hadoop is employed to analyze large-scale datasets stored in massive storage devices, such as hard-disk drives (HDDs) [2]. These datasets are accessed sequentially. Therefore, use of sequential storage-access methods may improve Hadoop performance by storing files at quickly accessible locations of an HDD to facilitate faster I/O operation [2], [3]. Such methods can be employed to improve the performance of I/O-intensive Hadoop jobs. This paper presents such an approach and evaluates it by using an experimental system operating on a realistic workload generated by a statistical workload injector for MapReduce (SWIM). SWIM, which has been recently proposed [4], [5], facilitates meaningful experimental performance evaluations of the Hadoop platform, whilst emulating several workload types [1].

Manuscript received December 30, 2019.
Manuscript revised May 17, 2020.
Manuscript publicized June 30, 2020.

[†]The authors are affiliated with the Kogakuin University, Tokyo, 163–8677 Japan.
[††]The author is affiliated with the University of Florida, Florida 32611, United States of America.
a) E-mail: cm19036@ns.kogakuin.ac.jp

DOI: 10.1587/transinf.2019EDP7337

Of primary focus in this study is a Hadoop use case, wherein a sequence of several Hadoop jobs must be executed. In our study, SWIM jobs are categorized as being map-heavy, shuffle-heavy, or reduce-heavy, and their I/O and CPU behaviors are thoroughly investigated. Based on the above investigation, map-heavy jobs are determined to be CPU-intensive, whereas the shuffle- and reduce-heavy jobs are determined to be I/O-intensive. Next, optimization of the file-storage strategy for the said three job types is performed, taking into consideration the extent to which temporary files and permanent files are used. Finally, comparisons are made of the total times required for completion of all three jobs using the (1) default approach (non-optimized; hereinafter referred to as "normal"), (2) authors' previously proposed method (referred to as "existing"), and (3) newly proposed file-storage strategies (referred to as "proposed").

The remainder of this paper is organized as follows. Section 2 reviews extant related work, whereas Sect. 3 discusses features of SWIM jobs as well as the relationship between HDD file locations and sequential read/write speeds. Section 4 proposes the new method for improving SWIM-job performance. Section 5 presents a comparative evaluation of the three above-mentioned methods. Section 6 discusses results obtained in this study, and lastly, Sect. 7 lists major conclusions drawn from this study.

2. Related Work

2.1 MapReduce

As depicted in Fig. 1, a MapReduce job comprises three phases—map, shuffle, and reduce [6]. In the map phase, input data are divided into multiple input splits, each of which is received by a mapper which executes the user-defined map process and generates key-value pairs, which are stored in intermediate files. The said key-value pairs are sorted in the shuffle phase, grouped based on the key, and transmitted to reducers. In the reduce phase, each reducer executes a user-defined reduce task based on received key-values and generates outputs that serve as results for the current Hadoop job, and the same are stored as output data.

2.2 SWIM

SWIM is a workload emulator capable of generating realistic MapReduce jobs based on actual workloads of production Hadoop clusters, such as those processed by

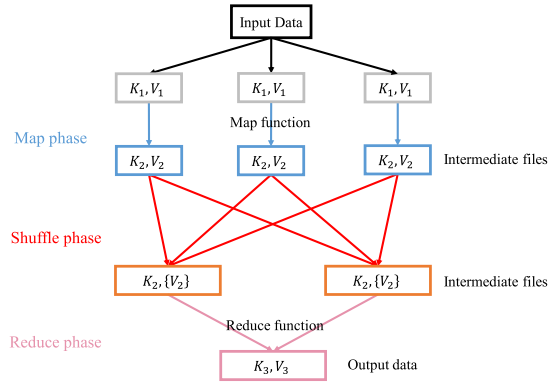

Fig. 1 Overview of MapReduce

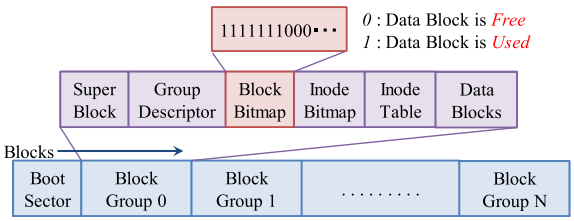

Fig. 2 Overview of ext2/3/4 file system

Facebook. Additionally, SWIM jobs can be configured by changing parameter values. Specifically, each SWIM job has several configurable parameters, such as job ID, input size (bytes) per map operation, shuffle size, output size per reduce operation, and number of reducers. Moreover, each job-submission interval can be controlled [4], [5]. This paper evaluates Hadoop performance for different usage scenarios by varying parameters from Facebook traces.

2.3 Extant File-Storage Methods

Methods proposed in [2] and [3] serve to improve sequential I/O performance by optimizing file-storage strategies, and are based on the principle that HDDs employing zone-bit recording (ZBR) provide faster sequential access to outer disk zones compared to inner zones. The mechanism underlying these methods uses block-usage information of a file system to determine zones, wherein data can be stored, thereby avoiding their storage in inner disk zones to expedite file access. As depicted in Fig. 2, several file systems, such as ext2/3/4 [7], divide the drive space into blocks of 4-KB each (methods described in [2], [3] were implemented with ext2/3 file systems). These file systems construct block groups containing a fixed number of blocks. The default size of each block group is 128 MB, and a single block group comprises a block bitmap, inode bitmap, inode table, and data blocks. The block bitmap indicates block usage and availability using states of 0 and 1, which indicate whether a block is available or occupied, respectively. The inode bitmap controls inode usage, whereas the inode table manages file information as well as its storage location. Lastly, data blocks store file data. The two previously proposed methods [2], [3] modify block bitmap information of inner zone blocks to avoid their usage, thereby facilitating exclusive utilization of outer zones that permit high sequential-access speeds.

The method described in [3] is static, since it sets inner-zones block-bitmap states to 1 prior to job execution. In contrast, the method reported in [2] is dynamic because it continuously observes the size of usable disk area, the block-bitmap state of which is not set to 1. The said area is expanded or shrunk depending on threshold values of the usable-area size. Whereas the above dynamic method stores all files in outer disk zones, the proposed method aims to optimize file storage based on job features.

The goals of the following previous studies are the improvement of sequential I/O performance. Ozawa et al. proposed a method for improving the sequential I/O performance on Hadoop [8]. They focused on the single-Reduced WordCount job and proposed a method for improving job performance. The job was I/O bounded and sequentially accessed the data files. The method decreased the size of sequentially accessed data by compressing the data. Like our work, they aimed to improve the performance of sequential file accesses. However, their approach was quite different from our method. Their work did not take job features into account. Their method can be applied together with our method. Wang et al. proposed a log-structured filesystem (LFS) called PROFS [9]. The method places data considering the characteristics of the ZBR hard disk drive. The filesystem puts active and inactive data in the faster and slower zones, respectively. The method is specialized for LFS and reorganizes data on the disk during LFS garbage collection. The method utilizes the performance characteristics of ZBR hard disk drives like our work. However, this does not take job features into account, unlike our work.

The following past studies focus on I/O scheduling. I/O scheduling on HDD and flash memory are discussed in [10] and [11], respectively. The AS I/O scheduler [12] is specialized for improving sequential storage access. An I/O request is first issued by an application and subsequently processed by the operating system. Past work on schedulers addresses the second phase while our work is for the first phase. Past work does not modify sequential access after I/O requests by applications, and does not consider file placement optimization. Naturally, optimizations in both phases are important and both optimizations can be utilized together. Schindler et al. proposed to utilize disk-specific knowledge to match access patterns to the strengths of disks [13]. Most rotational latency and seeking overheads could be avoided by allocating and accessing related data on disk track boundaries. They mainly focused on rotation delay and seek overhead. They did not discuss placing large files in the faster zones in the HDD or considering job features.

Schlosser et al. discussed methods for improving the performance to access multidimensional datasets by placing the data in the disk according to the spatial locality of the data [14]. They discussed also the relationship between the logical address and the physical place. However, they did

not present a method for improving sequential I/O throughput while considering job features.

Next, we refer to studies on filesystems. Modern filesystems have sophisticated block placement algorithms. In the case of Ext2/3/4 [15], [16], an allocator reserves a range of blocks for a new file using the *reservation* feature. Blocks are reserved in the memory using *reservation* immediately and an allocation to disk is delayed until writeback time while blocks are allocated in the disk immediately by using the filesystem *preallocation* mechanism. This method improves the "sequentiality" of accesses to the disk by reducing the fragmentation of a file on the disk [15]. However, this method does neither consider the performance characteristics of ZBR HDDs nor utilize the outer zones for improving sequential access speed. Additionally, it does not take job features into account. The Ext4 filesystem also has the following five advanced placement policies. First is the multi-block allocator, which speculatively allocates multi blocks on disk to a file at file creation. The second is delayed allocation. The third tries to place the inode and the data block of a file in the same block group. The fourth tries to place all the inodes of files in a directory into the same block group. The fifth creates any new directory in the root directory in the least heavily loaded block group. These policies decrease the fragmentation of a file and increase the sequentiality of storage accesses. However, these do not improve sequential access performance by making use of the outer zones.

3. Basic Evaluation

This section investigates resource-consumption behaviors of map-, shuffle-, and reduce-heavy Hadoop jobs.

3.1 Basic Behavior of Hadoop Jobs

I/O and CPU utilizations as well as disk-location usage of map-, shuffle-, and reduce-heavy SWIM jobs are investigated in this study, since the proposed method stores individual files based on these features. SWIM jobs were executed using an experimental Hadoop system, and their parameter values were set as follows.

Values of the submit-time and inter-job submit gap were set to unity for all jobs. The input-file size equaled 20 GB. For map-heavy jobs, the number of map-input bytes was set to 10^{12}, whereas those of shuffle and reduce-output bytes were set to unity. Likewise, for shuffle-heavy jobs, the number of shuffle bytes was set to 10^{12}, whereas it was set to unity for other byte types, and, for reduce-heavy jobs, 10^{12} was used for reduce-output bytes and unity for other byte types. The Hadoop system was set to run in the pseudo-distributed mode. Specifications of the computer and HDD used in this study are described in Tables 1 and 2, respectively.

Average usages of I/O and CPU observed during execution of the map-, shuffle-, and reduce-heavy jobs are depicted in Fig. 3, whereas maximum usages of disk during and after execution of said jobs are depicted in Fig. 4.

Table 1 Specifications of computer used in this study

CPU	AMD Phenom 2 X4 965 Processor
OS	CentOS 6.10 x86_64 minimal
Kernel	Linux 2.6.32.57
Main Memory	4GB
HDD	500GB(ext3)
Hadoop Ver.	2.0.0-cdh4.2.1

Table 2 Specifications of HDD used in this study

Model Number (manufacturer)	DT01ACA050 (TOSHIBA)
Interface	SATA 3.0
Interface Speed	6.0Gbps
Device Size	500GB
Buffer Size	32MB
Rotation Rate	7200rpm

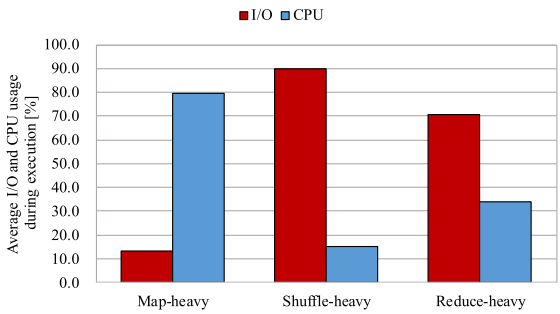

Fig. 3 Average I/O and CPU usage by SWIM Jobs

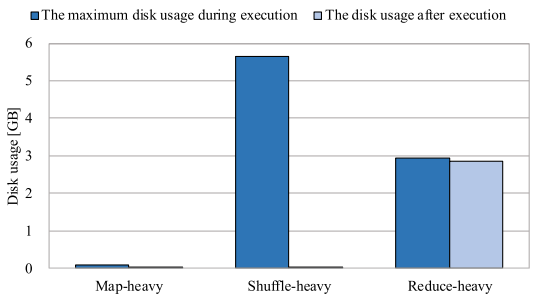

Fig. 4 Maximum disk usage during execution and disk usage after execution of SWIM jobs

Figure 5 depicts time histories of the percentage I/O usage and CPU utilization by a map-heavy job, whereas corresponding trends of the shuffle- and reduce-heavy jobs are depicted in Figs. 6 and 7, respectively. Temporal changes in the size of the used disk space when executing the said jobs are depicted in Figs. 8, 9, and 10, respectively.

Results of the above investigation lead to several conclusions, as listed below.

- Map-heavy jobs are CPU-intensive and temporarily store intermediate data, most of which are deleted

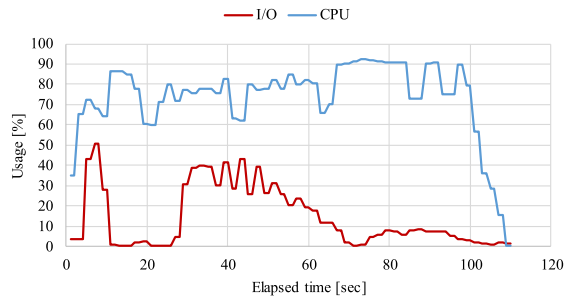

Fig. 5 I/O and CPU utilization by map-heavy jobs

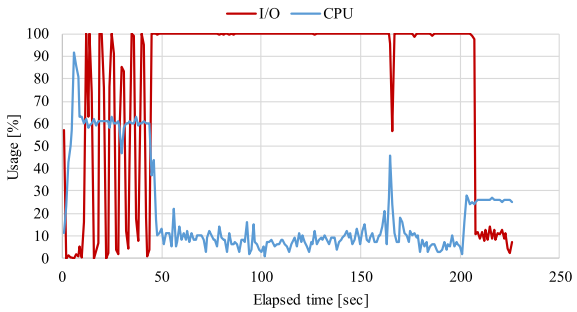

Fig. 6 I/O and CPU utilization by shuffle-heavy jobs

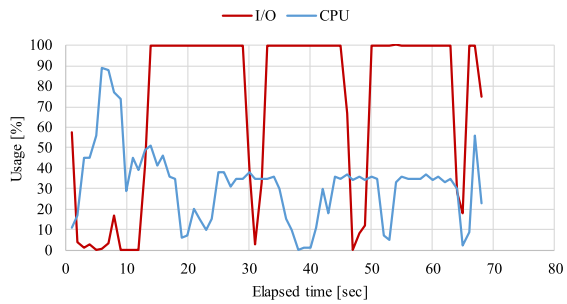

Fig. 7 I/O and CPU utilization by reduce-heavy jobs

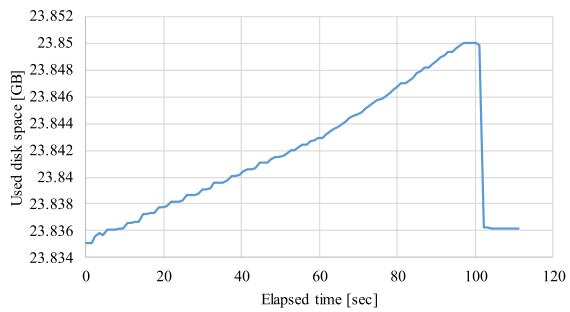

Fig. 8 Used disk space by map-heavy job

during execution.

- Shuffle-heavy jobs are I/O-intensive and temporarily store intermediate data, most of which are deleted.
- Reduce-heavy jobs are also I/O-intensive and permanently store output data, i.e. they are not deleted.

The first step of our proposed method is job classification in terms of CPU vs. I/O intensity and of usage of temporary vs. permanent files, in order to decide how the job's data should be placed on disks. This classification step can be automated for applications that are known in advance and executed repeatedly (see further discussion in Sect. 4. and Sect. 6.).

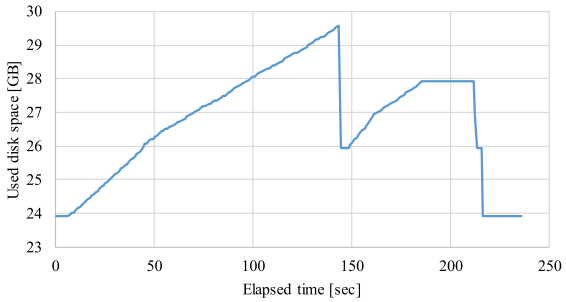

Fig. 9 Used disk space by shuffle-heavy job

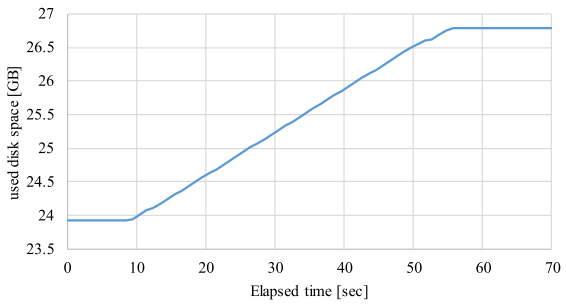

Fig. 10 Used disk space by reduce-heavy job

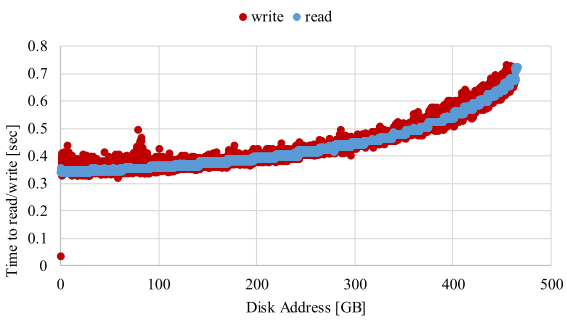

Fig. 11 Read/write times at HDD locations with different addresses

3.2 Sequential Storage Access

This subsection describes our investigation of the relationship between HDD data location and sequential read/write speeds. In this study, 64-MB read and write commands were repeatedly issued between the disk's first and last addresses, which correspond to the outmost and innermost disk zones, respectively. Figure 11 depicts the time required to complete the said 64-MB read and write operations at each given address. Results indicate a reduction in read/write speeds as data addresses correspond to successively inner disk zones. The access latency in the innermost disk zone was observed

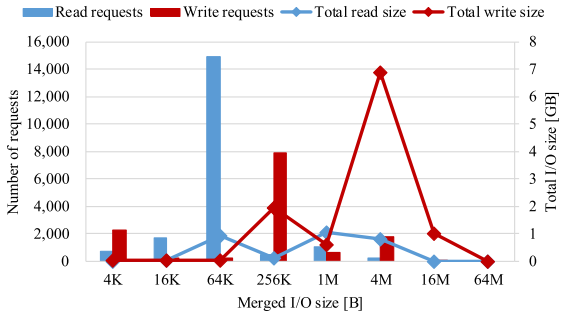

Fig. 12 Merged I/O request size for shuffle-heavy job

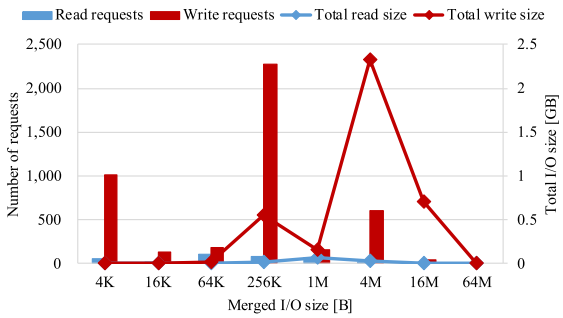

Fig. 13 Merged I/O request size for reduce-heavy job

to be nearly twice fhat in the outermost zone.

3.3 Merged I/O Sizes

Size frequencies of merged I/O requests [17]–[19] in the shuffle- and reduce-heavy jobs were also investigated in this study. These requests were obtained by merging temporally and spatially contiguous I/O requests into a single request.

The I/O throughput was expected to improve by placing files in outer zones, given that the jobs were I/O-intensive and sizes of their merged I/O requests were large. As already stated, map-heavy jobs are CPU-intensive and their performance was not expected to improve by optimizing file-storage location. These jobs were, therefore, excluded from this analysis. Figures 12 and 13 depict results obtained for the shuffle- and reduce-heavy jobs, respectively. These results demonstrate that multiple large I/O requests, whose sizes are equal to or larger than 4 MB, were made, and that the storage device was accessed in a sequential manner. Furthermore, a long time was consumed to process sequential I/O requests, implying that a decrease of this time is effective for improving the job performance. In summary, the rationale behind the proposed method is to enhance sequential I/O speeds by actively utilizing outer zones of an HDD to improve the performance of shuffle- and reduce-heavy jobs.

4. Proposed Method

This section proposes a method for improving Hadoop-job performance. This method is composed of two steps: application classification, described in Sect. 4.1, and optimization of file placement, described in Sects. 4.2 and 4.3.

4.1 Application Classification

The first step of proposed method classifies jobs into two groups, which are jobs with temporary files and those with permanent files.

It is useful to classify any job as a temp-file job (when temporary files use a significant fraction of the maximum storage needed by the job) or a perm-file job (when permanent files use most of the job-needed storage). To do this classification, each job is executed with repeating *df* commands, which measure the total size of the unused area. As a result, a profile is obtained of the total size of the used disk areas, as illustrated by Fig. 4. From this profile, it is possible to obtain the maximum usage during execution and the usage after execution. This method then classifies each job according to the following condition: if the usage after execution is less than the product of β by the maximum usage size then the job is a temp-file job, otherwise it is a perm-file job. β is also a tuning parameter which in our experiments was set to 0.1 (in general, it should be a small value).

This classification supports the file storage strategy, as described below.

4.2 File-Storage Strategy

The proposed technique for improving the I/O performance of Hadoop job sequences via job-feature-based file-storage optimization [20], [21] is described in this section. All jobs are assumed to be submitted and executed sequentially (see Sect. 6 for comments regarding cases wherein jobs are executed concurrently).

In the proposed method, a target HDD is divided into two areas—outer and inner. Files or data can be stored in the outer area per the following order of priority if they qualify as

(1) temporary files/data, or

(2) permanent files/data

In case (1), this method places files in the outer zones. In case (2), it places them in the inner zones. Reflecting the above priority criteria, the files of the following types of jobs can be stored in the outer area:

(1) temp-jobs, or

(2) perm-jobs.

The class of every application is initially investigated in the *application classification* step. The class of each application at runtime is determined according to this initial investigation. The file placing location is controlled based on this class of application at runtime.

For example, Map-heavy, Shuffle-heavy, and Reduce-heavy jobs are classified as types (1), (1), and (2), respectively. If permanent files are stored in the fastest zone, other applications cannot utilize the fastest zone after the

occupation. To facilitate frequent utilization of outer HDD areas, the proposed method avoids populating the same with permanent files. Thus, only temporary files are actively stored in this area.

The size of the outer area is the maximum size required for storing all the temporary files of jobs selected by our method to store their data in the outer zone. This is because the outer area must be able to store the temporary data of every job that uses the outer area.

4.3 Proposed Method Implementation

In this study, the proposed method was implemented using the ext2/3 file system. As already mentioned in Sect. 3.1, the said file systems create block groups, and each block group possesses its own block bitmap.

The proposed method modifies bits corresponding to inner-area blocks to a value equal to 1, thereby indicating that these blocks are in use, and hence, forcing map- and shuffle-heavy jobs to utilize outer HDD regions. Similarly, the above approach prevents reduce-heavy jobs from using outer areas by altering their bit values to 1. Consequently, output files generated by reduce-heavy jobs are not stored in outer HDD regions.

In addition, this method limits the zones in the areas that are usable. As a result, every file is forced to be placed at the outmost zone in each area by dynamically controlling the usable zones. Three functions—monitoring, expansion and shrinking—were used to implement the above-described technique. The monitoring function periodically checks the number of free blocks within a file system. When the monitoring function detects that the usable space is smaller than a set threshold, the expanding function is invoked to expand the usable space until its size exceeds the threshold. In contrast, the shrinking function is activated when the monitoring function detects the usable space to be larger than the threshold.

To accomplish the above-described control of disk usage, this method directly opens the device special file, such as /dev/sda, seeks the block bitmap and changes the flags in the bitmap with the administrator (root) authority.

Our implementation identifies the place of the block bitmap in the filesystem as follows. Ext2/3/4 filesystems separate the entire storage space into many block groups [16]. With the default setup, the sizes of each group and each block are 128 MB and 4KB, respectively. Each block group includes 32,768 blocks. The information on each block group is described in its Group Descriptor. A Group Descriptor includes information such as the address of its block bitmap, the address of the inode bitmap, the address of the inode table, the number of free blocks and the number of free inode numbers. The Group Descriptors of all the block groups are stored in the Group Descriptors table in the filesystem. The table exists at the beginning of the block 1 in the Block group 0. The table is at byte address 4096. The size of a Group Descriptor is 32 bytes. Thus, the Group Descriptor of Group X is at byte address $4096 + 32 \times X$. The address of the block bitmap is described at the beginning of each Group descriptor and its size is four bytes. Therefore, the address of the block bitmap of Block Group X is located at the byte addresses ranging from $4096 + 32 \times X$ to $4096 + 32 \times X + 3$ in the filesystem partition.

The method disables the usage of all the blocks in a block group as follows. In the case this system disables the flags of all the blocks in the Block Group X, this opens the device special file of the filesystem partition, seeks to the address $4096 + 32 \times X$, and reads 4 bytes from the address. These four bytes indicate the address of the block bitmap of the Block Group X. This is a block address and its unit block, i.e. 4 KB. Thus, this system seeks to the address of $4096 \times$ (the block address) and reads 4KB to backup the original state of the bitmap. Because the number of blocks in the block group is 32,768 with the default setup, the number of bits in a table is 32768 (i.e. 4KB). The system then writes 0xff to all the 4KB to indicate that all the flags are used. In the case this method makes the blocks in a block group usable, this restores the backup bitmap data into its block bitmap.

5. Performance Evaluation

5.1 Basic Evaluation

To evaluate the performance of the proposed method, a series of Hadoop jobs generated by SWIM, which were the same as the jobs in Sect. 3, were executed in several experiments. One such job set, illustrated in Fig. 14, comprised 27 job groups, wherein a sequence of map-, shuffle-, and reduce-heavy groups was repeated 9 times. Thus, each set contained nine instances each of map-, shuffle-, and reduce-heavy groups. Additionally, each job group comprised 20

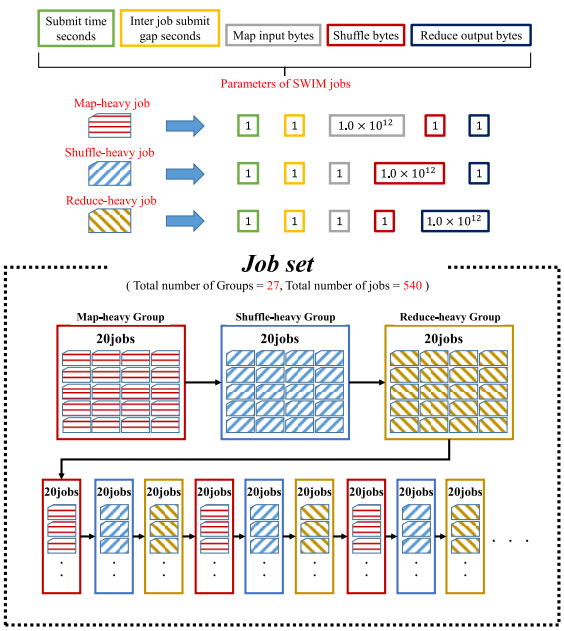

Fig. 14 Job set

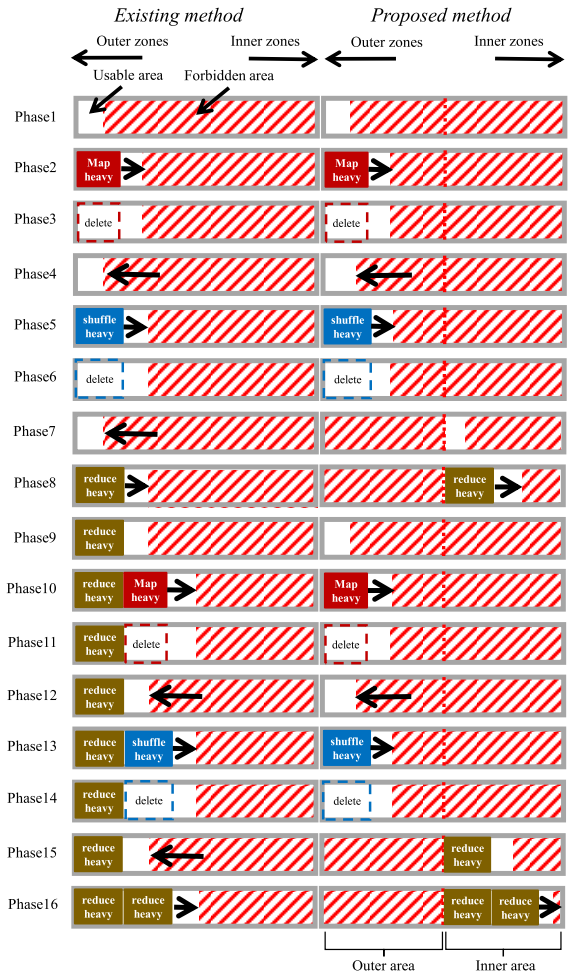

Fig. 15 File-storage strategies of existing and proposed methods

jobs. Execution of each job set was initiated with an empty HDD. At the end of each execution cycle, the entire HDD storage space was occupied by output files generated by reduce-heavy jobs. The specification of the used computer is the same as that in Sect. 3. The parameters α and β are 0.3 and 0.1, respectively. The existing method is implemented and evaluated with Ext3 filesystem. Thus, we compare the performances of the existing and proposed methods in this subsection.

According to the file-storage strategy described in Sect. 4, the files were stored in accordance with the priorities described in Sect. 4.1.

Figure 15 illustrates file-storage patterns of the existing and proposed methods. Right arrows indicate expansion of usable area, and left arrows indicate shrinkage of usable area. In this case, files qualified as (1)-(2) and (3)-(4) in Sect. 4.2 are stored in the outer and inner areas, respectively.
Phase 1:
(Existing) It prevents the use of areas other than the first 5GB of the disk. Thus the first 5GB is the usable area.
(Proposed) Same as for the existing method.
Phase 2:
(Existing) The files of a Map-heavy job are stored in the usable area. The size of the usable area may become less than the threshold for expansion and thus cause the expansion of the usable area.
(Proposed) Same as the existing method. Files are stored in the outer area because the job is classified as type (2).
Phase 3 and 4:
(Existing) The files are deleted after the Map-heavy job execution. The size of the usable area becomes larger than the threshold for shrinking and it is shrunk to the threshold, which is 8 GB.
(Proposed) Same as the existing method.
Phase 5:
(Existing) The files of the Shuffle-heavy job are stored in the usable area. The size of the area becomes less than the threshold and the area is expanded.
(Proposed) Same as the existing method. The job is classified as type (1).
Phases 6 and 7:
(Existing) The files are deleted and the size of usable area is shrunk to 8 GB.
(Proposed) Same as the existing method.
Phases 8 and 9:
(Existing) The files of the Reduce-heavy job are stored in the fastest zone in the usable area. The usable area is expanded.
(Proposed) The files of the Reduce-heavy job are stored in the fastest zone in the inner area because the job is classified as type (3). The usable area is expanded. The usable area for the next Map-heavy job is changed from the inner to the outer areas.
Phase 10:
(Existing) The files of the Map-heavy job are stored in the usable area. The place is inner than that in Phase 2. The usable area is expanded.
(Proposed) The files of the Map-heavy job are stored in the usable area. They are placed in the fastest zones like in Phase 2. The usable area is expanded.
Phases 11 and 12:
(Existing) Same as Phase 3 and 4.
(Proposed) Same as Phase 3 and 4.
Phase 13:
(Existing) The files of the Shuffle-heavy job are stored in the usable area. The place is inner than that in Phase 2. The usable area is expanded.
(Proposed) The files of the Shuffle-heavy job are stored in the usable area. They are placed in the fastest zones like in Phase 2. The usable area is expanded.
Phase 14 and 15:
(Existing) Same as Phase 6 and 7.
(Proposed) Same as Phase 6 and 7.
Phase 16:
(Existing) The files of the Reduce-heavy job are stored in the usable area. The usable area is expanded.
(Proposed) Same as for the existing method.

The existing method stores the map- and shuffle-heavy job files in disk regions located inwards compared to those occupied by reduce-heavy job files. With increase in the number of reduce-heavy job executions, files corresponding

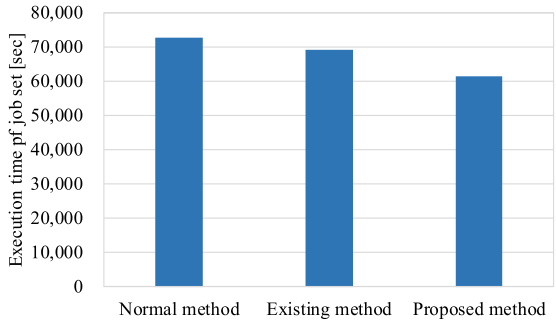

Fig. 16 Total execution time of job set

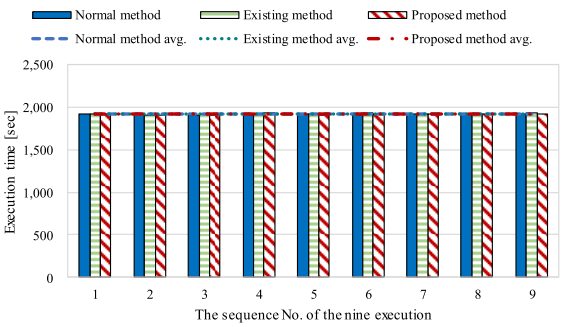

Fig. 17 Execution time of each map-heavy group

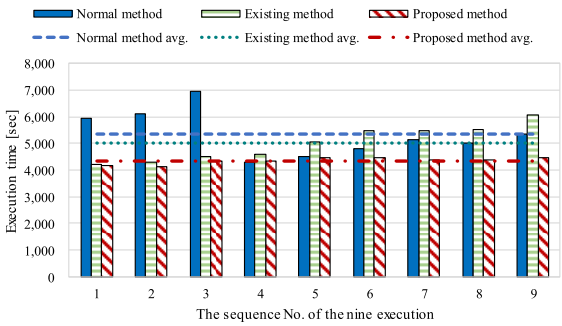

Fig. 18 Execution time of each shuffle-heavy group

to subsequent jobs are stored in increasingly inner disk zones. Consequently, the performance of jobs—especially I/O-intensive shuffle-heavy jobs—deteriorates. In contrast, the proposed method does not store output files of reduce-heavy jobs in the fast (outer) disk zones (Fig. 15), which are ideal for storing temporary files. In the proposed method, files corresponding to the map- and shuffle-heavy jobs are always temporarily stored in the fastest HDD zones to facilitate optimum disk and system performance.

Figure 16 shows the average time required to execute 5 job sets, whereas Figs. 17–19 depict times required to complete the map-, shuffle-, and reduce-heavy jobs from the first set, respectively.

Figure 16 indicates that, when using the proposed method, the overall job-execution time is reduced by 15.4% and 11.1% compared to that observed when employing the normal and existing methods, respectively.

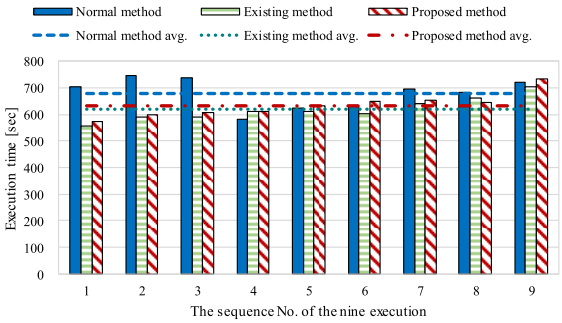

Fig. 19 Execution time of each reduce-heavy group

Table 3 Specifications of computer used in this study

CPU	AMD Ryzen Threadripper 1900X 8-Core Processor
OS	CentOS 6.10 x86_64 minimal
Kernel	Linux 2.6.32.57
Main Memory	4GB
HDD	500GB(ext3)
Hadoop Ver.	2.0.0-cdh4.2.1

Figure 17 shows that both the existing and proposed methods do not significantly reduce the execution time for CPU-intensive map-heavy jobs. Figure 18 clearly indicates that both the existing and proposed methods yield improved Hadoop performance over the normal method. The observed improvement when employing the proposed method is significant, when compared to that observed when using the existing method, wherein the time required to complete a job increases with increase in the number of jobs to be executed. The primary reason for this is that outer HDD zones are increasingly used for data/file storage as more reduce-heavy jobs are executed.

Figure 19 indicates that, when compared to the existing method, the proposed method does not effectively reduce the completion time of reduce-heavy jobs. The observed difference between completion times of reduce-heavy jobs when employing the existing and proposed methods is not considerable. The time required to complete a job increases with the number of completed jobs when employing the existing and proposed methods. The impact of this increase is significant, potentially overcoming the difference between execution times that result from the two methods. On the contrary, the difference of Shuffle-heavy jobs was large because the times to complete the jobs with the proposed method was the shortest at every execution. As a result, the time taken by the proposed method to complete all the groups was shorter than that of the existing method. After the Hadoop space becomes full, the output files are moved from the Hadoop space to a user's space. This frees fast-access storage space which eliminates the above-mentioned execution time increases for the next reduce-heavy jobs to be executed.

Table 4 Specifications of HDD used in this study

Model Number (manufacturer)	DT01ACA050 (TOSHIBA)
Interface	SATA 3.0
Interface Speed	6.0Gbps
Device Size	500GB
Buffer Size	32MB
Rotation Rate	7200rpm

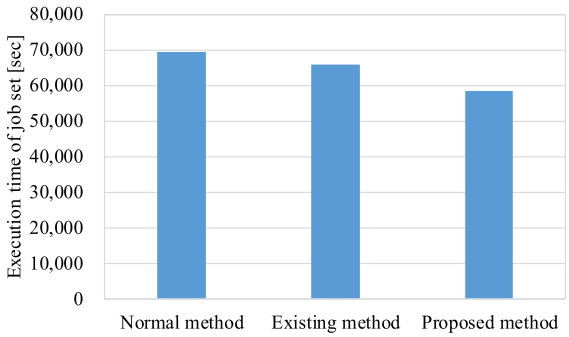

Fig. 20 Total execution time of job set (ext3)

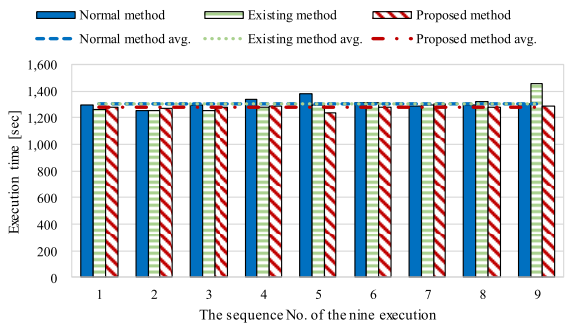

Fig. 21 Execution time of each map-heavy group (ext3)

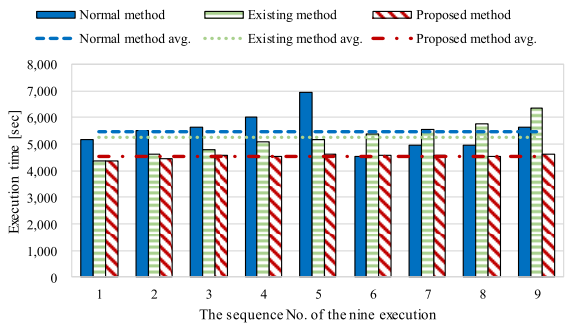

Fig. 22 Execution time of each shuffle-heavy group (ext3)

5.2 Dependency on Machine

Here, we present an evaluation of our proposed method with another computer. The specifications of the used computer and HDD are described in Tables 3 and 4, respectively.

Figure 20 shows the average time required to execute 5 job sets, whereas Figs. 21–23 depict times required to complete the map-, shuffle-, and reduce-heavy jobs from the first set, respectively. Figure 20 indicates that, when using the proposed method, the overall job-execution time is reduced by 16.1% and 11.5% compared to that observed when employing the normal and existing methods, respectively. These results indicate that the proposed method is similarly effective also with this different computer.

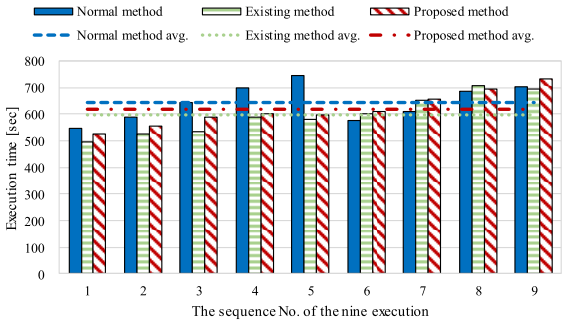

Fig. 23 Execution time of each reduce-heavy group (ext3)

6. Discussion

The proposed method relies on the knowledge of Hadoop job characteristics; i.e., whether the jobs are map-, shuffle- or reduce-heavy. This knowledge is often available in cases where jobs are repeatedly executed on different data. For instance, a search engine updates its index based on data concerning freshly crawled web pages every day. This is a typical example of a shuffle-heavy job. Similarly, on an electronic shopping site, online transaction processing (OLTP) jobs are executed every day. This is also true of online analytical processing (OLAP) applications, wherein repeated jobs possess similar features. Additionally, the knowledge of job characteristics required by the proposed method can be easily obtained. CPU and I/O usages can be calculated by executing simple commands—"vmstat" and "iostat," respectively. Likewise, temporal changes in disk usage can be obtained by repeating the "df" command. Therefore, the proposed method is applicable in several situations.

An alternate approach for actively using outer HDD zones involves splitting the storage device into multiple partitions. Storing temporary files in fast partitions is one possible means of implementing the proposed method. However, this is only effective in cases where the total size of temporary files is known and strictly limited. The proposed method can be applied in more situations by adopting dynamic file-size changes via bitmap modifications.

In this study, evaluation of the proposed method was performed by sequentially executing a set of jobs. Although several practical instances of such a use case exist, an equally important case is the one wherein jobs are executed concurrently. Such a case is, presently, a subject of intensive research because it requires certain modifications to be made to the proposed method along with extensive experimental evaluation.

Hadoop applications can access files on Hadoop distributed file system (HDFS) and local file systems of nodes on which they are run. HDFS files are mainly accessed by Hadoop for reading and writing input and output data, respectively, whereas files on a local system are used for intermediate data processing. The proposed method can be implemented by accessing local file-system information. Since HDFS is constructed over a local file system, the proposed method is effective for both HDFS and local system files.

Next, we discuss the performance improvement of the proposed method in a distributed environment. We expect that similar performance improvement is obtained in a distributed environment because the method can be applied to each data node individually. Namely, the method improves the I/O performance of each data node and the performance of the entire system is improved by this method. The effect of the optimization of file placing location in each node was studied in [17].

Lastly, we discuss implementation using Ext4, which is a newer filesystem. Ext4 supports some additional functions, such as the metadata checksum. Thus, the proposed method has to modify not only block bitmaps but also its checksum or to disable the checksum. Ext4 also supports block management with extent. The proposed method has to modify both its block bitmap and extent information in order to control file placing location.

7. Conclusions

This study investigates CPU- and I/O-resource consumptions as well as disk-space utilization by map-, shuffle-, and reduce-heavy SWIM jobs. A method has been proposed for improving I/O performance of Hadoop applications via consideration of target-job features. Results obtained via performance evaluation of the proposed method demonstrate that the proposed method enhances the performance of Hadoop jobs by 15.4% versus 4.8% by a method previously proposed by the authors. The new method outperformed the existing method by 11.1%.

As a future endeavor, the authors intend to develop and evaluate similar methods for concurrent jobs, once the experimental capability to run Hadoop jobs in the fully distributed mode has been established. For evaluation, we plan to consider Ext4 implementation and complex workloads that use Hadoop as a component, such as search engines and OLTP systems.

Acknowledgments

This work was supported in part by JST CREST, Japan—grant number JPMJCR1503—and JSPS KAKENHI—grant numbers 26730040, 15H02696, and 17K00109. This work was also funded in part by a grant (NSF ACI 1550126 and supplement DCL NSF 17-077) received from the National Science Foundation, USA.

References

[1] G.J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM, vol.51, no.1, pp.107–113, Jan. 2008. DOI: https://doi.org/10.1145/1327452.1327492

[2] E. Fujishima and S. Yamaguchi, "Dynamic File Placing Control for Improving the I/O Performance in the Reduce Phase of Hadoop," Proc. 10th International Conference on Ubiquitous Information Management and Communication (IMCOM '16), ACM, New York, NY, USA, Article 48, 7 pages, 2016. DOI: http://dx.doi.org/10.1145/2857546.2857595

[3] E. Fujishima and S. Yamaguchi, "Improving the I/O Performance in the Reduce Phase of Hadoop," 2015 Third International Symposium on Computing and Networking (CANDAR), Sapporo, pp.82–88, 2015. doi: 10.1109/CANDAR.2015.24

[4] GitHub - SWIMProjectUCB/SWIM: Statistical Workload Injector for MapReduce-Project at UC Berkeley AMP Lab, https://github.com/SWIMProjectUCB/SWIM

[5] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, 10 pages, July 2011. DOI: 10.1109/MASCOTS.2011.12

[6] F. Ahmad, S. Lee, M. Thottethodi, and T.N. Vijaykumar, "MapReduce with Communication Overlap (MaRCO)," Journal of Parallel and Distributed Computing, vol.73, no.5, pp.608–620, May 2013. DOI: https://doi.org/10.1016/j.jpdc.2012.12.012

[7] R. Card, T. Ts'o, and S.Tweedle, "Design and Implementation of the Second Extended Filesystem," First Dutch International Symposium on Linux, 1994.

[8] T. Ozawa, M. Onizuka, Y. Fukumoto, and S. Moriai, "MapReduce optimization using mapper-side aggregation," IPSJ Transaction on Advanced Computer Systems, vol.6, no.3, pp.71–81, Oct. 2013. (in Japanese)

[9] J. Wang and Y. Hu, "PROFS-performance-oriented data reorganization for log-structured file system on multi-zone disks," MASCOTS 2001, Proc. Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Cincinnati, OH, pp.285–292, 2001.

[10] J. Axboe, "Linux block io - present and future," Proc. Ottawa Linux Symposium, pp.51–61, Ottawa Linux Symposium, 2004.

[11] Y. Nakamura, S. Nomura, K. Nagata, and S. Yamaguchi, "I/O Scheduling in Android Devices with Flash Storage," 8th International Conference on Ubiquitous Information Management and Communication ACM IMCOM (ICUIMC), Article 83, 7 pages, 2014.

[12] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," SIGOPS Oper. Syst. Rev., vol.35, no.5, pp.117–130, Oct. 2001.

[13] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger, "Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics," Conference on File and Storage Technologies 2002 (FAST 2002), pp.259–274, 2002.

[14] S.W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G.R. Ganger, "On multidimensional data and modern disks," Conference on File and Storage Technologies 2005 (FAST 2005), pp.225–238, 2005.

[15] M. Cao, T.Y. Ts'o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas, "State of the Art: Where we are with the Ext3 filesystem," Ottawa Linux Symposium 2005 (OLS 2005), pp.69–95, 2005.

[16] djwong, "Block and Inode Allocation Policy," https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Block and Inode Allocation Policy, accessed Nov. 18, 2019.

[17] E. Fujishima, K. Nakashima, and S. Yamaguchi, "Hadoop I/O Performance Improvement by File Layout Optimization," IEICE Trans. Inf. & Syst., vol.E101-D, no.2, pp.415–427, 2018. doi:

10.1587/transinf.2017EDP7114
[18] S. Yamaguchi, M. Oguchi, and M. Kitsuregawa, "Trace system of iSCSI storage access," The 2005 Symposium on Applications and the Internet, Trento, Italy, pp.392–398, 2005. doi: 10.1109/SAINT.2005.65
[19] S. Yamaguchi, M. Oguchi, and M. Kitsuregawa, "iSCSI analysis system and performance improvement of sequential access in a long-latency environment," Electronics and Communications in Japan (Part III: Fundamental Electronic Science), vol.89, no.4, pp.55–69, Wiley Subscription Services, Inc., A Wiley Company, April 2006. DOI: 10.1002/ecjc.20238
[20] M. Nakagami, J.A.B. Fortes, and S. Yamaguchi, "Job-aware Optimization of File Placement in Hadoop," BDCAA 2019 The 1st IEEE International Workshop on Big Data Computation, analysis, and Applications, Conference (COMPSAC), Milwaukee, WI, USA, pp.664–669, 2019. doi: 10.1109/COMPSAC.2019.10284
[21] M. Nakagami, J.A.B. Fortes, and S. Yamaguchi, "Usable Space Control Based on Hadoop Job Features," 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW), Takayama, 2019.

Makoto Nakagami received his degree (B.E.) in Engineering from the Kogakuin University in 2019. He is presently pursuing Masters in electrical engineering and electronics at the Kogakuin University.

Jose A.B. Fortes is the AT&T Eminent Scholar and Professor of Electrical and Computer Engineering at the University of Florida where he founded and is the Director of the Advanced Computing and Information Systems Laboratory. His research interests are in the areas of distributed computing, autonomic computing, cyberinfrastructure and human-machine intelligent systems. José Fortes is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) professional society and a Fellow of the American Association for the Advancement of Science (AAAS).

Saneyasu Yamaguchi received his Ph.D. degree in Engineering from The University of Tokyo in 2002. From 2002 to 2006, he studied I/O processing at the Institute of Industrial Science, The University of Tokyo. He is now with Kogakuin University. His current research interests include operating systems, virtualized systems, and storage systems.