Job-aware Optimization of File Placement in Hadoop

Makoto Nakagami
Electrical Engineering and Electronics
Kogakuin University Graduate School
Tokyo, Japan
cm19036@ns.kogakuin.ac.jp

Jose A. B. Fortes
Advanced Computer and Information Systems (ACIS) Lab
University of Florida
Gainesville, USA
fortes@ufl.edu

Saneyasu Yamaguchi
Department of Information and Communications Engineering
Kogakuin University
Tokyo, Japan
sane@cc.kogakuin.ac.jp

***Abstract*—Hadoop is a popular data-analytics platform based on the MapReduce model. When analyzing extremely big data, hard disk drives are commonly used and Hadoop performance can be optimized by improving I/O performance. Hard disk drives have different performance depending on whether data are placed in the outer or inner disk zones. In this paper, we propose a method that uses knowledge of job characteristics to place data in hard disk drives so that Hadoop performance is improved. Files of a job that intensively and sequentially accesses the storage device are placed in outer disk tracks which have higher sequential access speed than inner tracks. Temporary and permanent files are placed in the outer and inner zones, respectively. This enables repeated usage of the faster zones by avoiding the use of the faster zones by permanent files. Our evaluation demonstrates that the proposed method improves the performance of Hadoop jobs by 15.0% over the normal case when file placement is not used. The proposed method also outperforms a previously proposed placement approach by 9.9%.**

Keywords—Hadoop, MapReduce, SWIM, Filesystem

I. INTRODUCTION

Hadoop is a popular big-data processing platform based on the MapReduce model [1]. Hadoop is used for a variety of applications which could be I/O-intensive and/or CPU-intensive. Intuitively, Hadoop performance can be improved by taking into account the nature of the applications when using storage resources.

In many cases, Hadoop is used to analyze large-scale datasets stored in massive storage devices, such as hard disk drives (HDDs) accessed sequentially [2]. There are methods to improve the performance of such sequential storage access that work by placing files into locations of an HDD where I/O access is faster than in other locations [2][3]. These methods can be the basis of approaches to improve the performance of I/O-intensive Hadoop jobs. This paper proposes such an approach and evaluates it in an experimental system running a realistic workload using the Statistical Workload Injector for MapReduce (SWIM). SWIM has been recently proposed [4] to enable meaningful experimental evaluations of Hadoop performance. It can emulate many types of workloads [1].

We focus on a Hadoop use-case where a sequence of Hadoop jobs of several kinds must be executed. First, we categorize SWIM jobs as Map-heavy, Shuffle-heavy, or Reduce-heavy jobs and thoroughly investigate their I/O behaviors and CPU behaviors in order to reveal their features. We then show that Map-heavy jobs are CPU-intensive and that Shuffle-heavy and Reduce-heavy jobs are I/O-intensive. Second, we discuss the optimization of file placement for these three types of jobs. Third, we compare the total times to complete all these jobs when using (1) a default (i.e. non-optimized) file placement determined by the operating system underlying Hadoop (hereon called the *normal* method), (2) a file placement approach previously proposed by the authors (hereon referred to as the *existing* method), and (3) the new *proposed* method.

The rest of this paper is organized as follows. Section II provides background information. Section IV reviews related work. Section V discusses the features of SWIM jobs and the relationship between file location in an HDD and the speeds of sequential reads and sequential writes. Section VI proposes a method for improving SWIM job performance. Section VII comparatively evaluates the three methods. Section VIII discusses results. Section VIII concludes the paper.

II. BACKGROUND

Here, we introduce MapReduce. As shown in Fig. 1, each MapReduce job is composed of three phases, namely the Map phase, Shuffle phase, and Reduce phase. In the Map phase, the Input Data is divided into multiple Input splits. Each Mapper receives an Input split, executes the user-defined Map process, and creates the key-value pairs. These key-value pairs are stored in the intermediate files. In the Shuffle phase, the intermediate key-value pairs are sorted, grouped by the key and transmitted to the reducers. In the Reduce phase, each reducer executes the user-defined Reduce task on its received key-values and creates outputs. These outputs are the results of the Hadoop job and are stored in the Output data.

III. RELATED WORK

A. SWIM

SWIM is a workload emulator that can generate realistic MapReduce jobs based on real workloads from a production Hadoop cluster in Facebook. In addition, SWIM jobs can be configured by changing some parameter values. Specifically, each SWIM job has configurable parameters such as job ID, input size (bytes) per map operation, shuffle size, output size per reduce operation and the number of reducers. Moreover, each job submission interval can be controlled [4]. In this paper, we evaluate the performance of Hadoop for different usage scenarios by varying the parameters in these Facebook traces.

B. Previously proposed file placement method

Here, we explain methods proposed in [2][3] for improving sequential I/O performance by optimizing file placement. These methods use the fact that hard disk drives using Zone Bit

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.10284

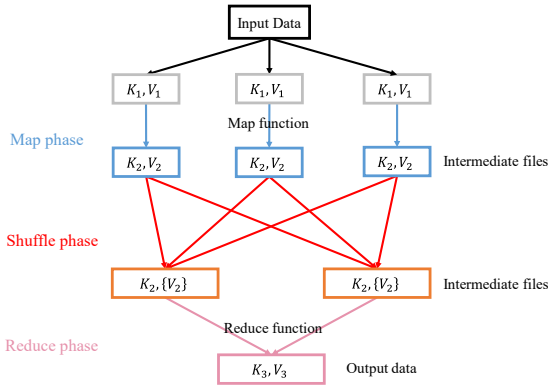

Fig.1. Overview of MapReduce.

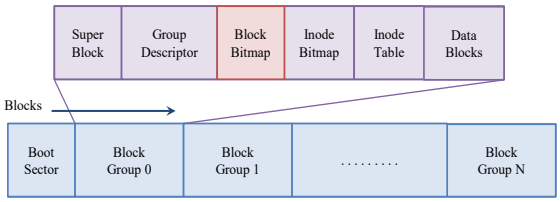

Fig.2. Overview of ext2/3

TABLE I. Specifications of the computer used in experiments.

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 II. Specifications of the HDD used in experiments.

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

Recording (ZBR) provide faster sequential access in outer disk zones than in inner zones. They avoid placing files in the inner zones by controlling the block usage information of the file system. Many file systems, for example ext2/3/4, divide the entire space into 4KB blocks (the methods in [2][3] were implemented with ext2/3/4). These file systems construct block groups that contain a defined number of the blocks. The size of each block group is 128 MB when using the default setting. Every block group has its own block bitmap, an inode bitmap, inode table, and data blocks. The block bitmap manages block usage in the block group. 0 and 1 in the bitmap indicate whether the block is free or in-use, respectively. Inode bitmap manages the usage of inodes in the group. The inode table manages file information including file placement location. The data blocks store data of files. The two previously proposed methods [2][3] change the bitmap bits for the blocks in the inner zones in order to avoid using these inner zones. As a result, only the outer zones, which have higher sequential access speeds, are utilized by these methods.

The method in [3] is static, i.e. it sets the block bitmaps of the inner zones to 1 prior to job execution. On the other hand, method in [2] is dynamic, i.e. it keeps watching the size of the usable area, whose block bitmaps are not set as 1, and then expands or shrinks the usable area according to threshold values, dynamically. While this dynamic method always places every file in the outmost zones, the proposed method optimizes the file placement according to the features of the job.

IV. Basic Performance Evaluation

In this section, we explore the resource consumption behaviors of Map-heavy, Shuffle-heavy and Reduce-heavy jobs.

A. Basic Behavior of Hadoop Jobs

Here, we investigate the I/O and CPU utilization and Disk location usage of Map-heavy, Shuffle-heavy, and Reduce-heavy SWIM jobs. Our proposed method places the files according to these features.

We executed SWIM jobs on an experimental Hadoop system. The parameters were set as follows (italics are used to refer to parameters). *Submit time* and *inter job submit gap* were set to one for all the jobs. The *Input file size* is 4GB. In the case of Map-heavy jobs, *map input bytes* was set to 3.0×10^{11} and *shuffle bytes* and *reduce output bytes* were set to one. In the case of Shuffle-heavy jobs, *shuffle bytes* was set to 1.0×10^{12}, and the others were set to one. In the case of Reduce-heavy jobs, *reduce output bytes* was set to 1.0×10^{12}, and the others were set to one. The Hadoop system was set to run in the pseudo distributed mode. The specifications of the computer and HDD in the experiments are described in Table I and Table II, respectively.

I/O usage and CPU utilization by a Map-heavy job are shown in Fig. 3 and those of Shuffle-heavy and Reduce-heavy jobs are shown in Fig. 4 and Fig. 5, respectively. The temporal changes of the size of the used disk space of these jobs are depicted in Figs. 6, 7, and 8.

These results lead to several conclusions. First, a Map-heavy job is CPU-intensive. It temporarily stores the intermediate data in the storage and deletes almost all of these intermediate data during the execution. Second, a Shuffle-heavy job is I/O-intensive, temporarily stores the intermediate data, and deletes almost all the data. Third, a Reduce-heavy job is I/O-intensive and permanently stores the output data, i.e. the data are not deleted.

B. Sequential Storage Access

In this subsection, we investigate the relationship between the location of data in an HDD and the speeds of a sequential read and a sequential write. We repeatedly issued 64-MB read and write commands from the disk's first address to its last address. The first and last addresses correspond to the outmost and innermost zones, respectively. Fig. 9 shows the time to

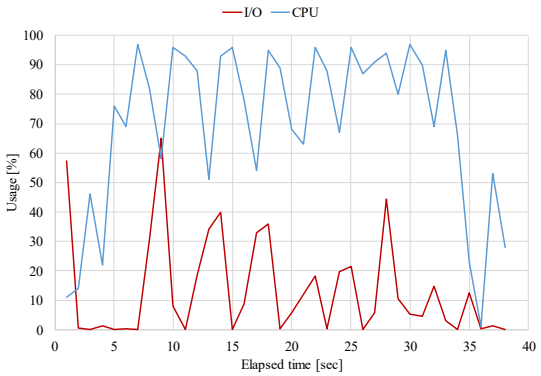

Fig.3. CPU and I/O utilization of Map-heavy job.

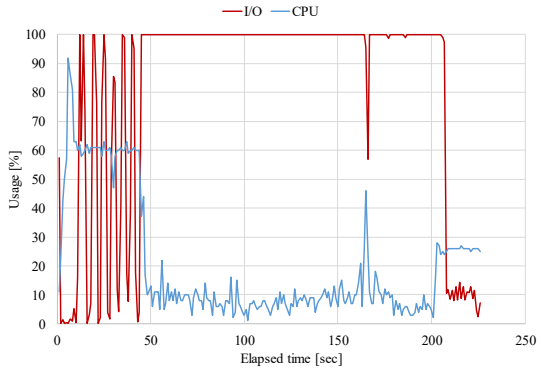

Fig.4. CPU and I/O utilization of Shuffle-heavy job.

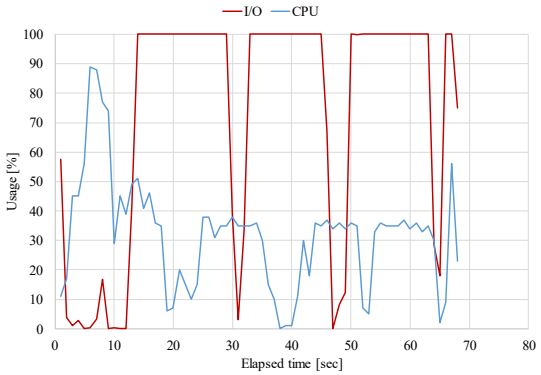

Fig.5. CPU and I/O utilization of Reduce-heavy job.

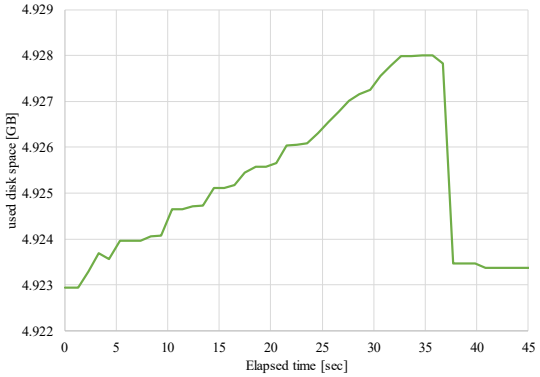

Fig.6. Used disk space by Map-heavy job.

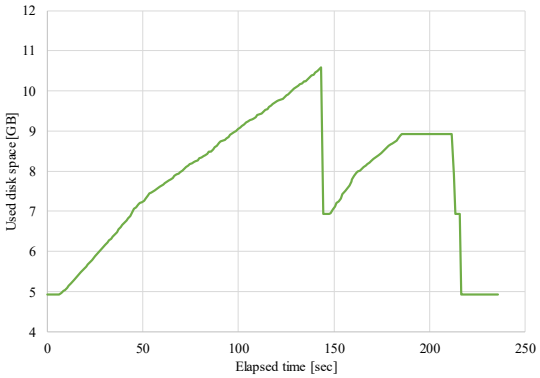

Fig.7. Disk space used by Shuffle-heavy job.

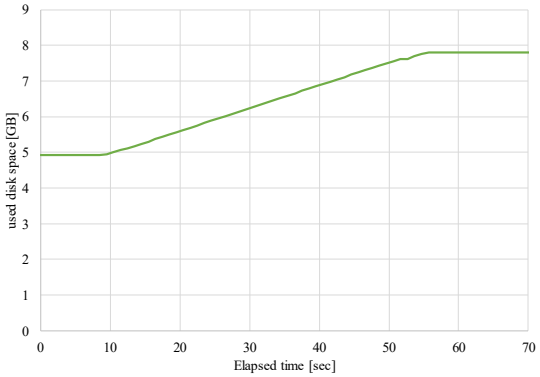

Fig.8. Disk space used by Reduce-heavy job.

complete 64MB of a read or write at every address. The results indicate that the speeds decrease as the address increases. The access latency in the innnermost zone is almost twice that in the outmost zones.

C. Merged I/O Sizes

In addition, we investigated the frequencies of the sizes of the merged I/O requests [8], which were obtained by merging the temporarily and spatially continuous I/O requests into one request, of Shuffle-heavy and Reduce-heavy jobs.

We did expect that I/O throughput improves by placing files in outer zones in case of the job is I/O-intensive and the sizes of their merged I/O requests are large. Map-heavy jobs are CPU-intensive and we did not expect improvement of their performance by optimizing file location. Then, we investigated the merged I/O request sizes of Shuffle-heavy and Reduce-heavy jobs. Fig.10 and Fig.11 depict the results of Shuffle-heavy and Reduce-heavy jobs, respectively. The results show that many large I/O requests were issued, i.e. the storage device was accessed in a highly sequentially manner. Therefore, the rationale for the method proposed in Section V is to improve sequential I/O speed by actively utilizing the outer zones in the HDD in order to improve the performance of the Shuffle-heavy and Reduce-heavy jobs.

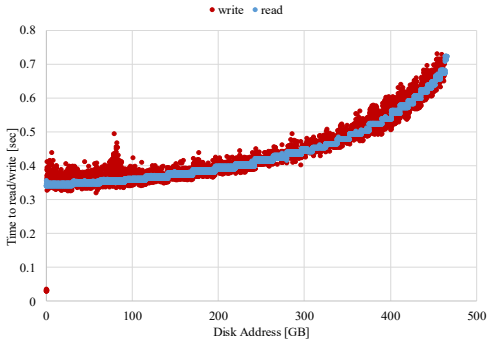

Fig.9. Read/Write times of HDD locations with different addresses.

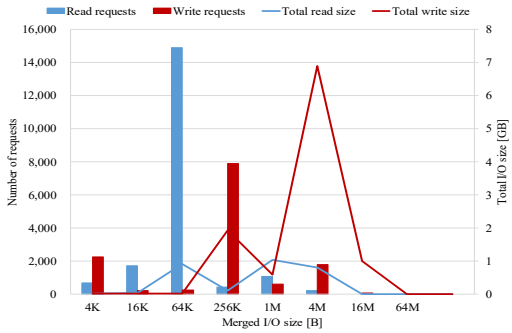

Fig.10. Merged I/O request size of Shuffle-heavy job.

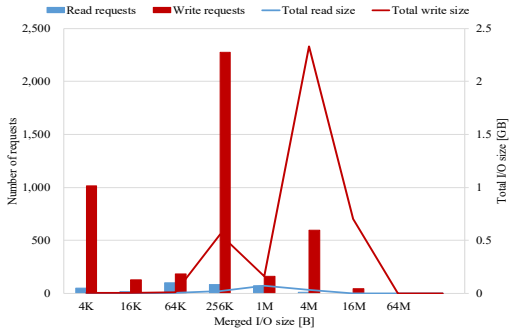

Fig.11. Merged I/O request size of Reduce-heavy job.

V. Proposed Method

A. File placement policy

In this section, we propose a method for improving the I/O performance of a sequence of Hadoop jobs by optimizing file placement based on job features. We assume that the jobs are submitted and executed sequentially (see Section VII for comments regarding the case when jobs are executed concurrently).

The proposed method places files in outer zones according to the following order of priority:

(1) The file is temporary and is used by an I/O intensive process.

(2) The file is temporary and is not used by an I/O intensive process.

(3) The file is not temporary and is used by an I/O intensive process.

(4) The file is not temporary and is not used by an I/O intensive process.

In order to utilize the outer zones many times, the method avoids occupying the outer zones with permanent files. Instead, the proposed method actively places temporary files in the outer zones.

B. Implementation

In our implementation, the proposed method uses ext2/3/4 file systems. These file systems create block groups and every block group has its own block bitmap for the blocks in the group, as described in Section III.BB. The method forces Map-heavy and Shuffle-heavy jobs to use the fastest zones by changing the bits of the non-fastest blocks into 1, which indicates that they are being *used.* The proposed method prevents Reduce-heavy jobs from using the fastest zones by changing their bits into 1. As a result, the output files of the Reduce-heavy jobs are not placed in the fastest zones.

VI. Evaluation

In this section, we evaluate the performance of the proposed method. We executed a series of Hadoop jobs. A set of jobs is illustrated in Fig. 12. A set is composed of 27 job groups, sequenced as Map-heavy group, Shuffle-heavy group, Reduce-heavy group, Map-heavy group, Shuffle-heavy group, Reduce-heavy group, and so on. A set contains nine Map-heavy groups, nine Shuffle-heavy groups, and nine Reduce-heavy groups. Each job group consists of 20 jobs. We started each execution of a set of jobs with the hard disk drive empty and the drive was almost fully occupied by the output files of these Hadoop jobs after an execution of a set of jobs. As described, it is mainly consumed by the files of Reduce-heavy jobs.

According to the proposed policy in Section V, the files are practically placed as follows.

The files of Shuffle-heavy jobs are stored in the fastest zones because they are temporary and the process is I/O-intensive as described in Section IV.A. The files of Map-heavy jobs also stored in the fastest zones because they also are temporary as described. Reduce-heavy jobs' files are not stored in the fastest zone because they stay permanently in the storage device as described.

Figs. 13 and 14 illustrate the file placement of the existing and proposed methods. The existing method places files of Map-heavy and Shuffle-heavy jobs in an inner zone than the zone of the files of Reduce-heavy job after execution of an Reduce-heavy job. As the number of executions of Reduce-heavy jobs increases, file placement locations of the following jobs move to inner zones. As a result, the performance of jobs, especially I/O intensive Shuffle-heavy jobs, declines. On the contrary, the proposed method does not store permanent files in the fastest zones as shown in Fig. 14. The output files of the Reduce-heavy jobs are not stored in the fastest zones, which are for temporary files, and the files of Map-heavy and Shuffle-heavy jobs are always placed temporarily in the fastest zones.

Fig. 15 shows the average time to complete a set of jobs by executing five sets. Figs. 16, 17, and 18 depict the times to complete Map-, Shuffle-, and Reduce-heavy jobs of the first set, respectively.

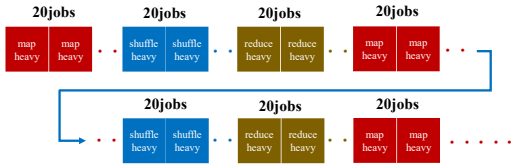

Fig.12. A set of sequential jobs.

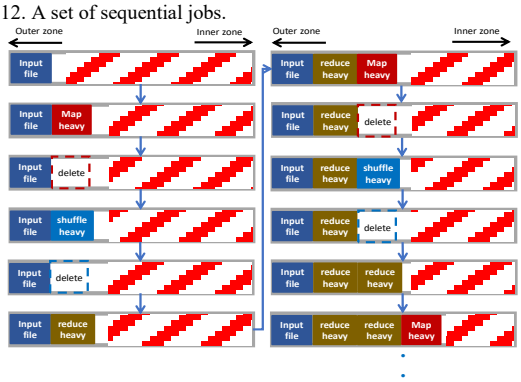

Fig.13. File placement using the previously proposed (existing) method [2].

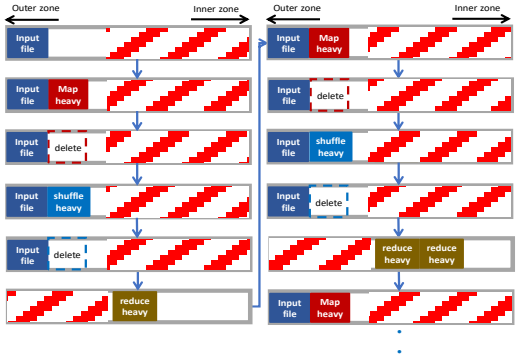

Fig.14. File placement using the method proposed in this paper.

Fig. 15 indicates that the execution time of the jobs when using the proposed method is smaller than the execution times when using the normal and existing methods by 15.0% and 9.9%, respectively.

Fig. 16 implies that both the existing and proposed methods did not reduce the execution time of Map-heavy jobs. This is due to Map-heavy jobs being CPU-intensive jobs. Fig. 17 shows that both the existing and proposed methods improve Hadoop performance over the normal method. The size of improvement of the proposed method is larger than that of the existing method. In the case of the existing method, the time to complete a job increased as the number of executed jobs increased. This is mainly because faster zones were increasingly occupied as Reduce-heavy jobs were executed.

Fig.18 indicates that the proposed method did not reduce times to complete the Reduce-heavy jobs while the existing method reduced them. The proposed method did not place the files of the Reduce-heavy jobs in the fastest zones while the existing method did the best effort for every Reduce-heavy job, which means providing the fastest zone at every execution. However, the difference between times of the Reduce-heavy jobs of the existing and proposed methods was not large because the time to complete a Reduce-heavy job increased with the existing method as the number of completed Reduce-heavy jobs increased. 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 to complete all the groups of the proposed method was shorter than that of the existing method.

VII. Discussion

The proposed method relies on knowledge of characteristics of Hadoop jobs to be executed, namely whether they are Map-heavy, Reduce-heavy or Shuffle-heavy. This knowledge is often available in use-cases when jobs are repeatedly executed on different data. For example, a search engine system updates its index, which is a typical Shuffle-heavy job, according to newly crawled web pages every day. Similarly, in an electronic shopping site, similar online transaction processing (OLTP) jobs are executed every day. In the case of online analytical processing (OLAP) applications, the features of repeated jobs are also very similar. In addition, the features of jobs that our method required can be easily obtained. We measured the CPU and I/O usages by simply executing the `vmstat` and `iostat` commands, respectively. We got the temporal changes of the disk usage by simply repeating the `df` command. Therefore, we argue that the proposed method is applicable to many situations.

An alternate method for actively using the outer zones can rely on splitting a storage device into multiple partitions. Placing volatile files in a fast partition is one of the ways to implement the proposed approach. This is effective only when the total size of the volatile files is known and strictly bounded. On the other hand, the method proposed in this paper is applicable to more situations because it can adapt to dynamic file size changes by modifying the bitmap.

The proposed method evaluation considered the case when a set of jobs are executed sequentially. While there are many practical instances of this use-case, another important Hadoop use scenario is the case when jobs are executed concurrently. This case is the subject of current research, as it requires some modifications to the proposed method and a more extensive evaluation.

Hadoop applications access files on HDFS and local file system of its nodes. Files on HDFS are mainly accessed by Hadoop for reading its input data and writing its output data. Files on the local file system are used for intermediate data. The proposed method is realized by controlling the local file system information. Since HDFS is constructed over the local file system, the proposed method is effective on files on both HDFS and the local file system.

In this paper, we proposed a method to improve I/O performance for representative application targets generated by SWIM. As we describe, the method is based on characteristics of SWIM jobs, specifically on CPU usage, I/O usage, temporal changes of the size of the used disk space, and file volatility. By monitoring the execution of other applications we can obtain similar metrics thus generalizing the applicability of our method to those applications.

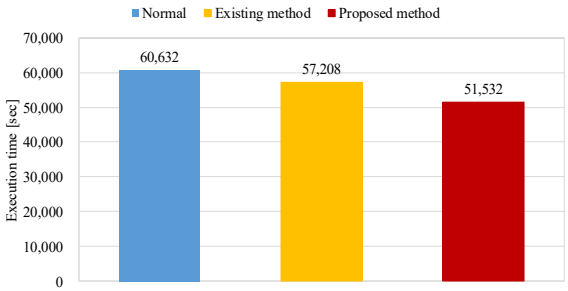

Fig.15. Total execution time of job sequence.

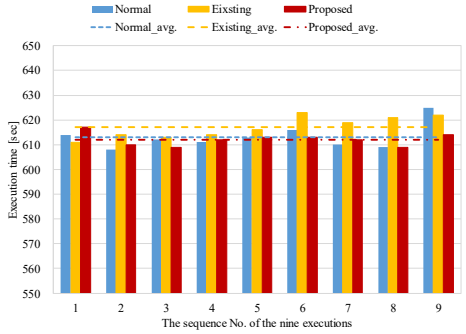

Fig.16. Execution time of each Map-heavy job

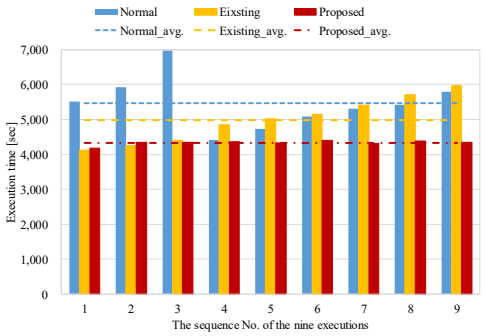

Fig.17. Execution time of each Shuffle-heavy job

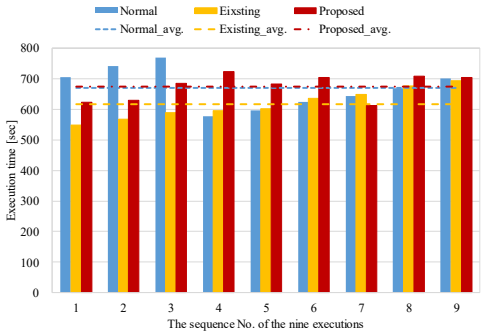

Fig.18. Execution time of each Reduce-heavy job

VIII. Conclusion

In this paper, we investigated the CPU and I/O resource consumptions and used disk space of SWIM jobs, which were categorized as Map-heavy, Shuffle-heavy, and Reduce-heavy. We then proposed a method for improving I/O performance considering the features of the target jobs. Our evaluation has demonstrated that the proposed method has improved the performance of the Hadoop jobs by 15.0% while the existing method did so by 5.6%. The new method also outperformed the existing method by 9.9%.

In future work, we plan to extend and evaluate similar methods for concurrent jobs when Hadoop runs in fully distributed mode.

Acknowledgment

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

References

[1] G. Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113. DOI: https://doi.org/10.1145/1327452.1327492

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

[3] Eita Fujishima and Saneyasu Yamaguchi, "Improving the I/O Performance in the Reduce Phase of Hadoop," 2015 Third International Symposium on Computing and Networking (CANDAR), Sapporo, 2015, pp. 82-88. 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] R.Card and T.Ts'o and S.Tweedle, "Design and Implementation of the Second Extended Filesystem," First Dutch International Symposium on Linux, 1994

[6] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi and T. N. Vijaykumar, " MapReduce with Communication Overlap (MaRCO)", Journal of Parallel and Distributed Computing, May 2013, Pages 608-620 ,DOI: https://doi.org/10.1016/j.jpdc.2012.12.012

[7] Yanpei Chen, Archana Ganapathi, Rean Griffith, Randy 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 , July. 2011 ,10 pages, DOI: 10.1109/MASCOTS.2011.1

[8] Eita FUJISHIMA Kenji NAKASHIMA Saneyasu YAMAGUCHI, "Hadoop I/O Performance Improvement by File Layout Optimization", IEICE TRANSACTIONS on Information and Systems, Vol.E101-D No.2 pp.415-427, doi: 10.1587/transinf.2017EDP711

[9] S. Yamaguchi, M. Oguchi and M. Kitsuregawa, "Trace system of iSCSI storage access," *The 2005 Symposium on Applications and the Internet*, Trento, Italy, 2005, pp. 392-398. doi: 10.1109/SAINT.2005.65

[10] Saneyasu Yamaguchi, Masato Oguchi, Masaru 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), Volume 89, Issue 4, Pages 55-69, Wiley Subscription Services, Inc., A Wiley Company, April 2006. DOI: 10.1002/ecjc.20238

[11] Yuta Nakamura, Kyosuke Nagata, Shun Nomura, and Saneyasu Yamaguchi. 2014. I/O scheduling in Android devices with flash storage. In Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication (ICUIMC '14). ACM, New York, NY, USA, Article 83, 7 pages. DOI: https://doi.org/10.1145/2557977.255802