Adaptive Control of Apache Spark's Data Caching Mechanism Based on Workload Characteristics

Hideo Inagaki*, Tomoyuki Fujii†, Ryota Kawashima‡ and Hiroshi Matsuo§

Nagoya Institute of Technology,in Nagoya, Aichi, 466-8555, Japan
Email: *inagaki@matlab.nitech.ac.jp, †t-fujii@matlab.nitech.ac.jp, ‡kawa1983@ieee.org, §matsuo@nitech.ac.jp

***Abstract*—Apache Spark caches reusable data into memory/disk. From our preliminary evaluation, we have found that a memory-and-disk caching is ineffective compared to disk-only caching when memory usage has reached its limit. This is because a thrashing state involving frequent data move between the memory and the disk occurs for a memory-and-disk caching. Spark has introduced a thrashing avoidance method for a single RDD (Resilient Distributed Dataset), but it cannot be applied to workloads using multiple RDDs because prior detection of the dependencies between the RDDs is difficult due to unpredictable access pattern. In this paper, we propose a thrashing avoidance method for such workloads. Our method adaptively modifies the cache I/O behavior depending on characteristics of the workload. In particular, caching data are directly written to the disk instead of the memory if cached data are frequently moved from the memory to the disk. Further, cached data are directly returned to the execution-memory instead of the storage-memory if cached data in the disk are required. Our method can adaptively select the optimal cache I/O behavior by observing workload characteristics at runtime instead of analyzing the dependence among RDDs. Evaluation results showed that execution time was reduced by 33% for KMeans using the modified Spark memory-and-disk caching rather than the original.**

***Index Terms*—Distributed processing, Big Data, Apache Spark, RDD, Memory cache, memory-and-disk caching**

I. Introduction

The amount of data is dramatically increasing due to the high popularity and adoption of IoT [1]. Apache Spark [2] is a widely used distributed processing framework for various gigantic workloads such as machine learning [3] and graph processing [4] to analyze gathered IoT data. Spark can efficiently process such large workloads by keeping intermediate data on the memory [5].

Spark explicitly caches reusable data to prevent them from being reconstructed as persisted data. This feature is effective for iterative processing workloads [6] because input and intermediate data are frequently reused. Such a caching mechanism achieves 15 times better performance than that of a traditional disk-based caching seen in Apache Hadoop [7]. However, performance gets worse when caching data cannot be stored into the storage-memory [8] because uncached data are deleted and they must be regenerated as needed. To prevent from performance degradation, Spark supports three types of cache stores, MEMORY_ONLY, DISK_ONLY, and MEMORY_AND_DISK. The DISK_ONLY usage always writes caching data to the disk to prevent the deletion and regeneration of data. The MEMORY_AND_DISK usage puts out least recently used data to the disk, while keeping heavily used data on the memory. However, this usage can raise a thrashing state involving frequent data move between the memory and the disk. The thrashing heavily degrades the performance because cached data are moved from the memory to the disk ($drop$) when caching data in the memory ($write\ operation$) and when using cached data in the disk ($read\ operation$).

Spark avoids the thrashing state when workloads involve a single RDD that is a collection of data distributed to multiple nodes. Specifically, caching data are directly written to the disk instead of the memory and cached data are directly returned to the execution-memory instead of the storage-memory. However, this method cannot be applied to workloads involving multiple RDDs because prior detection of the dependencies among RDDs is difficult.

Existing studies [9] [10] [11] showed that replacing the cache management algorithm (LRU [12]) is effective for avoiding frequent data drops for such workloads. LRC [9] keeps track of the reference count of data to determine adequate data to drop when the storage-memory has no space. This algorithm provides a more accurate indicator for the likelihood of future data access. LERC [10] detects dependencies between data in workloads to cache whole dependent data in the memory. WR [11] determines candidates to be dropped based on computation cost, reference count and size of data comprehensively. These three algorithms determine dropping data using various characteristics of data access pattern. However, preliminary information and analysis with high computation cost are necessary if workloads involve a large amount of input/intermediate data and multistage processing.

In this paper, we propose a thrashing avoidance method for workloads involving multiple RDDs. Our method adaptively modifies the cache I/O behavior depending on characteristics of workloads and does not need preliminary analysis or execution. In particular, caching data are directly written to the disk instead of the memory if frequent data drops occur while executing write operations. Further, cached data are directly returned to the execution-memory instead of the storage-memory for read operations. Our method can adaptively select the optimal cache I/O behavior by observing workload characteristics at runtime instead of analyzing the dependence among RDDs.

The remaining of this paper is organized as follows. Details of data cache management for Spark and problems related to the MEMORY_AND_DISK type of cache are shown in Section II. Influence of the thrashing is estimated and the optimal cache I/O behavior is researched by preliminary evaluation in Section III. In Section IV, we propose a thrashing avoidance method and describe its implementation details. The proposed method is evaluated by measuring execution time and the number of dropped data to check to avoid thrashing in Section V, and while the related work is analyzed in Section VI. Finally, this study is concluded and some future work directions are supplied in Section VII.

II. Cache Management for Apache Spark

In this section, problems of data cache management of Spark are described.

A. RDD

Spark uses RDD as a data format to process data in the memory for iterative processing workloads. RDD is a collection of data distributed to multiple nodes. Input data are converted into RDDs while the RDDs in turn are partitioned into multiple data blocks (*partitions*). Programmers can easily implement distributed processing by the use of RDD because RDD operations are automatically converted to operations to each partition.

B. Details of the caching mechanism

Spark caches input and intermediate RDDs to reuse them in the following execution steps to prevent the regeneration of RDDs. Spark supports three types of cache stores, MEMORY_ONLY, DISK_ONLY, and MEMORY_AND_DISK. Caching partitions are always written into the storage-memory, and cached partitions are discarded (MEMORY_ONLY) or dropped to the disk (MEMORY_AND_DISK) based on LRU when there is no free space in the storage-memory respectively. In the MEMORY_ONLY usage, heavily loaded regeneration of deleted partitions is needed when they are required. In the DISK_ONLY usage, caching partitions are always written to the disk to prevent the regeneration of data. However, the storage-memory is not utilized and disk accesses are increased.

The processing flow of the MEMORY_AND_DISK-based caching is shown in Fig. 1. Caching partitions are always written into the storage-memory. Already cached partitions are dropped to the disk based on LRU if the storage-memory has no space. Cached partitions are returned to the storage-memory and copied to the execution-memory. In addition, the returning partition results in dropping another one to the disk if the storage-memory has no space. These implementations have an advantage such that frequency used partitions are generally allocated in the memory. However, write/read operations always drop partitions to the disk if the storage-memory is filled. Spark can avoid such drops only if a single RDD is cached. Specifically, caching partitions are directly written to the disk instead of the memory and cached partitions are directly returned to the execution-memory instead of the storage-memory when the storage-memory is filled with partitions in the same RDD. On the other hand, prior detection of the

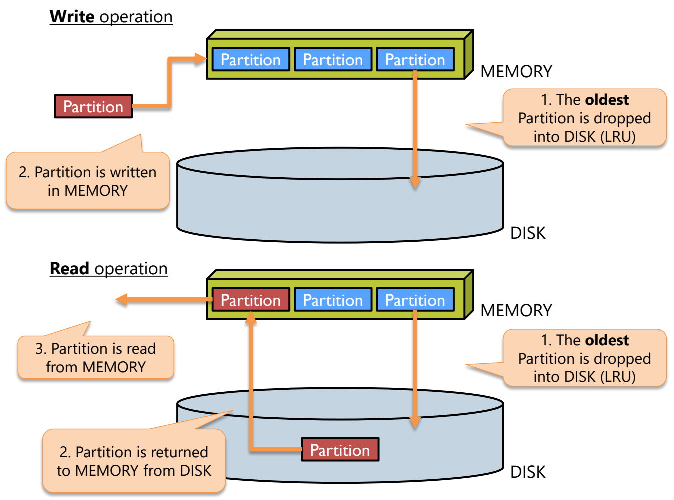
Write operation
Partition
Partition
Partition
Partition
MEMORY
1. The oldest Partition is dropped into DISK (LRU)
2. Partition is written in MEMORY
DISK
Read operation
Partition
Partition
Partition
MEMORY
3. Partition is read from MEMORY
1. The oldest Partition is dropped into DISK (LRU)
2. Partition is returned to MEMORY from DISK
Partition
DISK

Fig. 1. MEMORY_AND_DISK write/read operations when the storage-memory is filled with partitions of multiple RDDs

TABLE I
THE SPECIFICATION OF MASTER/SLAVE NODES AND SPARK

Evaluation environment	
OS	Ubuntu 14.04
CPU	Core i5- 3476K (3.20 GHz), 4 cores
Size of memory	4.2 GB (size of storage-memory: 2.4 GB)
HDD	1 TB
Spark version	2.0.1
The Number of machines	Master x1, Slave x1
File system	HDFS [13] (version 2.7.3)

TABLE II
THE SPECIFICATION OF KMEANS (USED HIBENCH [14])

Program	Configuration
KMeans	20 million samples 20 dimensions of samples k=10 5.0 GB size of cached data (vertex data: 4.0 GB, center distance: 1.0 GB)

TABLE III
THE SPECIFICATION OF NWEIGHT

Program	Configuration
NWeight	5,000,000 edges 3 degrees max 30 out edges 8.0 GB size of cached data (vertexes and edges)

dependence among RDDs is difficult if multiple RDDs are cached. Therefore, this raises a thrashing state involving frequent partitions move between the memory and the disk. The thrashing heavily degrades the performance because cached partitions are dropped for write/read operations in iterative processing workloads.

III. Preliminary Evaluation

A. Preliminary evaluation of the caching mechanism

Execution time has been measured with the three cache store types to estimate the influence of the thrashing mentioned in Section II. The evaluation environment and benchmark details are shown in Tables I and II, respectively. Execution time of KMeans is shown in Fig. 2. The execution time of MEMORY_ONLY usage was the longest because regeneration of deleted data became the performance bottleneck. The execution time of DISK_ONLY usage was the shortest because regeneration of deleted data and conversion from

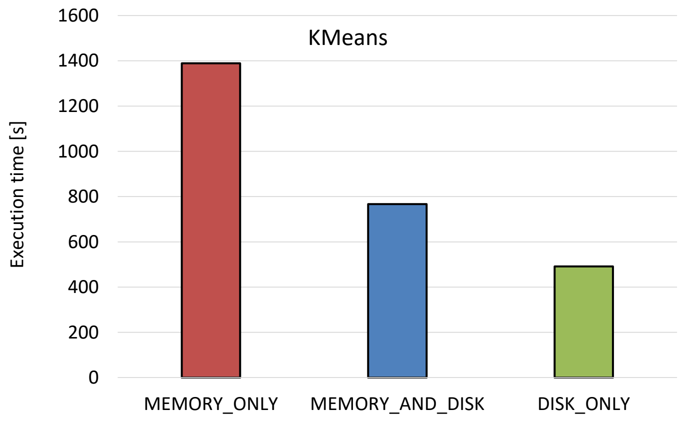

Fig. 2. Evaluation results of execution time with three types of cache stores (MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY)

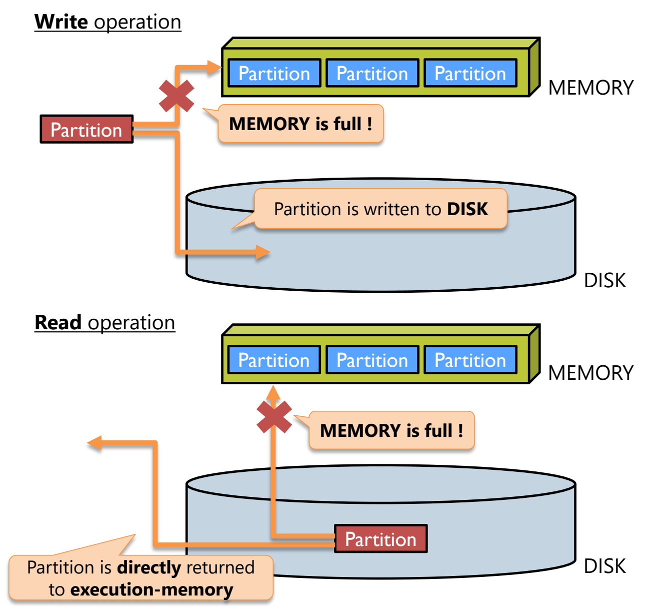

Fig. 3. Modified write/read operations

data to RDD were not necessary. The execution time of MEMORY_AND_DISK usage was longer than that of the DISK_ONLY usage. The number of drops of this situation was measured to check whether thrashing state occurred. As a result, drops occurred 36 times for write operations and 3,659 times for read operations. Therefore, the thrashing is the cause of the performance degradation.

B. Research of optimal cache I/O behavior with modified write/read operations

The MEMORY_AND_DISK usage caches data to the storage-memory for write/read operations of Spark. We modified the cache I/O behavior associated with write/read operations to evaluate its performance effect. Details of the changes are shown in Fig. 3. In write operations, caching partitions are directly written to the disk when the storage-memory is filled ($Modified\ write\ operations$). Partitions are written to

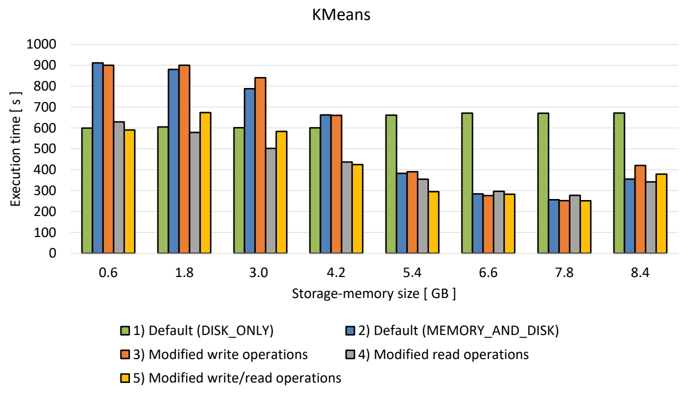

Fig. 4. KMeans evaluation result of execution time with five cache I/O behaviors

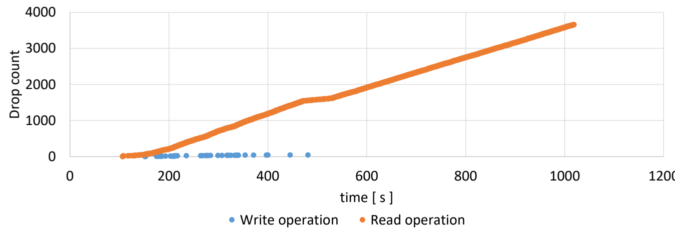

Fig. 5. KMeans transition of the number of drops

the memory as usual when the storage-memory has enough free space. In read operations, cached partitions are directly returned to the execution-memory when the storage-memory filled with other partitions ($Modified\ read\ operations$). Partitions are returned to the storage-memory as usual when the storage-memory has enough free space. These changes eliminate the thrashing state even when multiple RDDs are cached.

C. Preliminary evaluation of the cache I/O behavior

NWeight is a graph processing algorithm that involves iterative cached data processing as shown in the configuration in Table III. Execution time of two workloads, KMeans and NWeight, was measured with varying the cache I/O behavior to evaluate its effect. The following five cache I/O behaviors were used in the experiments.

1) Default (DISK_ONLY)
2) Default (MEMORY_AND_DISK)
3) Modified write operations
4) Modified read operations
5) Modified write/read operations

1) KMeans: Execution time of KMeans is shown in Fig. 4. Graphs represent execution time while changing the cache I/O behavior and the storage-memory size from 0.6 GB to 8.4 GB. The execution time of method (1) was constant because data were always cached in the disk. The execution times of methods (2), (3), (4) and (5) were reduced as the storage-memory size was increased from 0.6 GB to 7.8 GB due to the fact that the number of drops decreased as increasing memory-cached data and all cached data were written to the memory

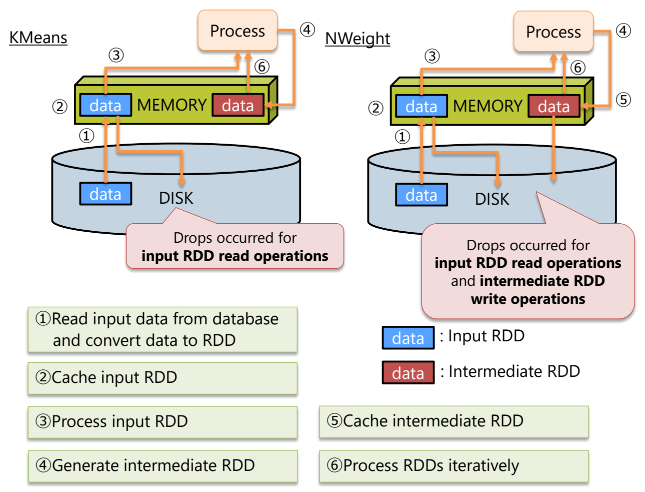
KMeans
Process
③
④
⑥
②
data
MEMORY
data
①
data
DISK
Drops occurred for **input RDD read operations**
NWeight
Process
③
④
⑥
⑤
②
data
MEMORY
data
①
data
DISK
Drops occurred for **input RDD read operations** and **intermediate RDD write operations**
data : Input RDD
data : Intermediate RDD
①Read input data from database and convert data to RDD
②Cache input RDD
③Process input RDD
④Generate intermediate RDD
⑤Cache intermediate RDD
⑥Process RDDs iteratively

Fig. 6. Dataflow of KMeans (left) and NWeight (right)

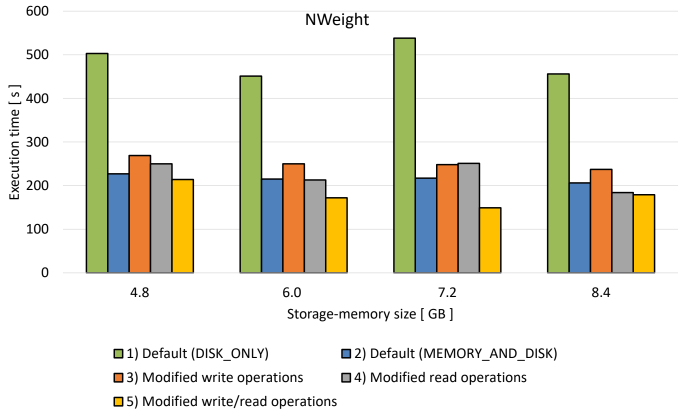

Fig. 7. NWeight evaluation result of execution time with five cache I/O behaviors

when the storage-memory size was larger than 5.4 GB. The execution time of memory usage was increased when the storage-memory size was 8.4 GB because memory thrashing occurred between physical and virtual memory. The execution times of methods (4) and (5) were reduced compared with the method (2) when the storage-memory size was less than 4.2 GB. In particular, the method (4) is the most effective (decreased by 33% on average).

Furthermore, the transition of the number of drops was measured to unveil why the modified read operations is the most effective method. The respective evaluation result are shown in Fig. 5. The moderate increase of the number of drops can be observed for write operations at the start up time ($\sim$ 500 s). In contrast, the number of drops for read operations increased until the end of execution. From the dataflow of KMeans, as shown in Fig. 6, the primary cause of the drops is reading input RDD. According to these results, the cause of thrashing is a large amount of drops for read operations. Therefore, thrashing can be avoided by modifying read operations from the start of execution.

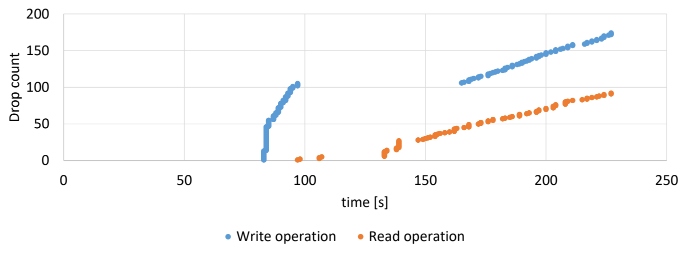

Fig. 8. NWeight transition of the number of drops

2) NWeight: Execution time of NWeight is shown in Fig. 7. Graphs represent execution time while changing the cache I/O behavior and the storage-memory size from 4.8 GB to 8.4 GB. Method (1) has a degraded performance due to increase in disk accesses. The execution time of method (5) was reduced compared with methods (1) and (2). Unlike in the case of KMeans, the method (5) is the most effective (decreased by 12% on average).

The transition of the number of drops is shown in Fig. 8. The number of drops for write operations increased more than 100 times after 100 seconds from the start of execution. The number of drops for write/read operations increased after that. From the dataflow of NWeight as shown in Fig. 6, the primary cause of the drops is reading input RDD and writing intermediate RDD. According to these results, the cause of thrashing is the number of drops for write operations (from start to 100 seconds) and the number of drops for write/read operations (from 100 seconds to end). Therefore, thrashing can be avoided by modifying write operations (from start to 100 seconds) and write/read operations (from 100 seconds to end).

IV. Proposed Method

In this section, we propose an adaptive control cache mechanism based on workload characteristics. This method can avoid thrashing state for workloads involving multiple RDDs.

The optimal cache I/O behavior differs depending on workload characteristics such as modified read operations are the most effective for KMeans and modified write/read operations are the most effective for NWeight. Fixed optimal cache I/O behavior is difficult because access pattern of the workload have to be known before execution. Therefore, we propose an adaptive control of cache I/O behavior based on workload characteristics by measuring the access pattern of the workload at runtime.

The number of drops is introduced as an indicator of modifying the cache I/O behavior. The number of drops of KMeans and NWeight are shown in Table IV. From these results, the tendency of the ratio of drops for write operations to drops for read operations was different immediately after the start of execution and the whole execution for NWeight. Therefore, the cache I/O behavior is modified at first when the number of drops becomes 100 times. The ratio of drops for write operations to drops for read operations was 7%

TABLE IV
THE NUMBER OF DROPS OF EACH WORKLOAD (FROM START OF EXECUTION TO 100 TIMES OF THE NUMBER OF DROPS AND WHOLE EXECUTION)

Program	No. of drops (write)	No. of drops (read)
KMeans (from start of execution to 100 times of the number of drops)	7	93
KMeans (whole execution)	40	3,659
NWeight (from start of execution to 100 times of the number of drops)	100	0
NWeight (whole execution)	171	96

(from start of execution to 100 times of the number of drops) and 1% (whole execution) for KMeans. It was 100% (from start of execution to 100 times of the number of drops) and 64% (whole execution) for NWeight. According to these observations, the ratio of modifying the cache I/O behavior is determined according to the content of Table V. The cache I/O behavior is not modified when the number of drops is less than 100 times.

TABLE V
RATIO OF MODIFYING THE CACHE I/O BEHAVIOR

Ratio of write operation drops	Modified write operations	Modified read operations
80% ∼	✓	-
30% ∼ 80%	✓	✓
∼ 30%	-	✓

V. EVALUATION

In this section, the proposed method is evaluated by measuring the number of drops and execution time of four workloads (KMeans, NWeight, Bayes and MovieLensALS). The evaluation environment is shown in Table I while benchmark details are shown in Tables II, III and VI.

TABLE VI
THE SPECIFICATION OF BAYES AND MOVIELENSALS

Program	Configuration
Bayes	100 classes 6.0 GB size of cached data
MovieLensALS	5.0 GB size of cached data

The transition of the number of drops used by the proposed method for KMeans and NWeight are shown in Fig. 9. For KMeans, the number of drops of the default method moderately increased for write operations immediately after the start up time, and the number of drops increased until the end of execution for read operations. Modification of cache reading behaviors by the proposed method occurred by dropping 97 times for read operations out of 100 times of drops and prevented drops from happening after that. For NWeight, the number of drops of the default method increased for write operations more than 100 times after 100 seconds

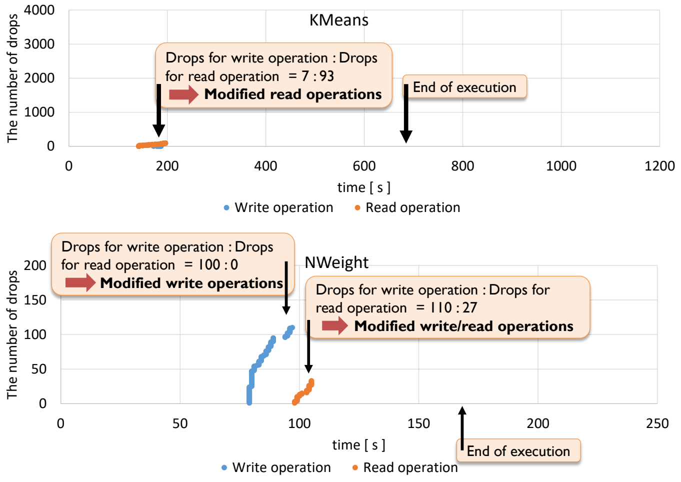

Fig. 9. KMeans and NWeight transition of the number of drops used the proposed method

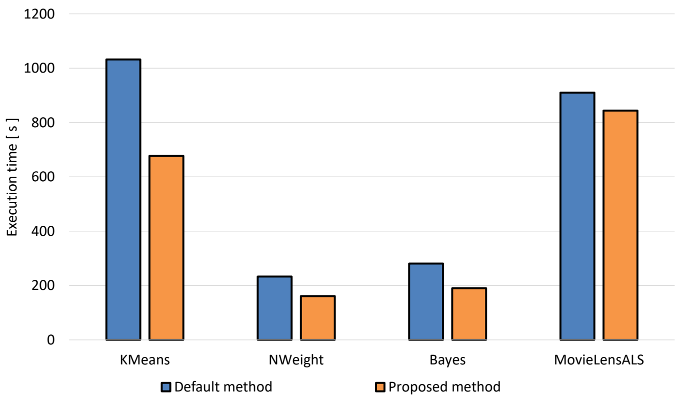

Fig. 10. Evaluation results of execution time for the default method and the proposed method

from the start of execution, and the number increased after that for write/read operations. Modification of cache writing behaviors by the proposed method occurred by dropping 100 times for write operations. Subsequently, modification of cache writing/reading behaviors by the proposed method occurred by dropping 27 times for read operations and prevented drops from happening after that.

Execution times of default MEMORY_AND_DISK and the proposed method are shown in Fig. 10. The execution time of proposed method was reduced by 33% for KMeans compared with the default method by the effect of modified read operations. The execution time was reduced by 31% for NWeight by the effect of modified write/read operations. The execution time was reduced by 28% for Bayes and 9% for MovieLensALS by the effect of modified read operations because the number of drops related to read operations arose in these two benchmarks. For these results, the proposed method improves performance by changing the adaptive cache I/O behavior depending on workload characteristics.

VI. Related Work

Koliopoulos et al. [15] and Xu et al. [16] proposed automatic mechanisms of deciding cache store types. In [15], the authors designed this mechanism based on a ratio of the storage-memory size to the disk size. However, this method cannot use the memory when the ratio is too low although data can be cached in the memory. Neutrino [16] determined caching partitions and optimal cache store types based on RDD access order. This method needs preliminary analysis on the workload to extract RDD access order, while our method does not need it.

Ho et al. [17] proposed PRDD that can efficiently update cached data. Spark returns whole RDD to the storage-memory even when a part of data of RDD is updated. PRDD returns the individual partition of RDD to the storage-memory and updates it to prevent from drops for read operations. This method is effective if many update operations to a part of RDD are executed. On the other hand, our method is effective if many write/read operations to whole RDD are executed.

Jiang et al. [18] mathematically derived a criteria for selecting optimal cached data compression to save memory space and reduce the disk I/O time. Our method can be more effective by incorporating this method because optimal cached data compression can increase the number of memory-cached data and decrease the number of drops. Further, Kryo serializer [19] can save memory space by converting cached data to a serialized form as Spark stores partitions as large byte arrays.

VII. Conclusion and Future Work

In this paper, we proposed a thrashing avoidance method that adaptively modifies the cache I/O behavior of Spark depending on workload characteristics for workloads involving multiple RDDs. The number of drops is introduced as an indicator which is exploited based on a respective decision table to dynamically modify the cache I/O behavior at runtime according to the ratio of the number of write to read drops. From the respective evaluation results, the proposed method with modified read operations reduced execution time by 33% for KMeans, 28% for Bayes and 9% for MovieLensALS. Further, the method with modified write/read operations reduced execution time by 31% for NWeight. According to these results, the proposed method is effective for workloads involving multiple RDDs.

First timing of modifying the cache I/O behavior and the ratio of modifying the cache I/O behavior are determined heuristically at present time. In future, these values are changed to optimal by investigating further benchmarks and considering other information such as the reference count to make our method adaptable to various workloads. Further, the proposed method is needed to validate in terms of incorporating other drop-preventing methods such as serialization and compression.

Acknowledgment

This work was supported in part by MEXT KAKENHI Grant Number 18K11324.

References

[1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Byers, "Big data: The next frontier for innovation, competition, and productivity." May 2011, https://bigdatawg.nist.gov/pdf/MGI_big_data_full_report.pdf.

[2] M. Zaharia, "Spark: In-Memory Cluster Computing for Iterative and Interactive Applications," in *Invited Talk. NIPS Big Learning Workshop: Algorithms, Systems, and Tools for Learning at Scale*, Granada, Spain, Dec. 12–17, 2011.

[3] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." vol. 14, pp. 599–613, Colorado, USA, Oct. 2014.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *Proc. 9th USENIX conference on Networked Systems Design and Implementation*, p. 2, California, USA, April 2012.

[6] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," *Proc. 9th USENIX conference on Networked Systems Design and Implementation*, pp. 267–280, California, USA, April 2012.

[7] "Apache Hadoop," http://hadoop.apache.org/[Accessed 21 February 2018].

[8] L. Gu and H. Li, "Memory or time: Performance evaluation for iterative operation on hadoop and spark," *Proc. 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, pp. 721–727, Hunan Province, China, Nov. 2013.

[9] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "LRC: Dependency-aware cache management for data analytics clusters," *Proc. IEEE international Conference on Computer Communications (INFOCOM)*, pp. 1–9, Georgia, USA, May, 2017.

[10] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "LERC: Coordinated Cache Management for Data-Parallel Systems," *Proc. IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, Singapore, Dec. 2017.

[11] M. Duan, K. Li, Z. Tang, G. Xiao, and K. Li, "Selection and replacement algorithms for memory performance improvement in Spark," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2473–2486, 2016.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," *Proc. IEEE 26th symposium on Mass storage systems and technologies (MSST)*, pp. 1–10, Nevada, USA, May, 2010.

[14] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai, "Hibench: A representative and comprehensive hadoop benchmark suite," *Proc. ICDE Workshops*, pp. 41–51, California, USA, April, 2010.

[15] A.-K. Koliopoulos, P. Yiapanis, F. Tekiner, G. Nenadic, and J. Keane, "Towards Automatic Memory Tuning for In-Memory Big Data Analytics in Clusters," *Proc. 2016 IEEE International Congress on Big Data (BigData Congress)*, pp. 353–356, San Francisco, USA, June, 2016.

[16] E. Xu, M. Saxena, and L. Chiu, "Neutrino: Revisiting Memory Caching for Iterative Data Analytics," *Proc. 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Colorado, USA, June, 2016.

[17] L.-Y. Ho, J.-J. Wu, P. Liu, C.-C. Shih, C.-C. Huang, and C.-W. Huang, "Efficient Cache Update for In-Memory Cluster Computing with Spark," *Proc. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 21–30, Madrid, Spain, May, 2017.

[18] Z. Jiang, H. Chen, H. Zhou, and J. Wu, "An Elastic Data Persisting Solution with High Performance for Spark," *Proc. 2015 IEEE International Conference on Smart City*, pp. 656–661, Chengdu, China, Dec. 2015.

[19] "Kryo," https://github.com/EsotericSoftware/kryo[Accessed 28 March 2018].