Understanding and Improving Disk-based Intermediate Data Caching in Spark

Kaihui ZHANG
University of Tsukuba
Tsukuba, Japan
Email: neilyo.chou@aist.go.jp

Yusuke TANIMURA*
Hidemoto NAKADA*
Hirotaka OGAWA
National Institute of Advanced Industrial Science and Technology (AIST)
Tsukuba, Japan
{yusuke.tanimura, hide-nakada, h-ogawa}@aist.go.jp
**Also with University of Tsukuba, Japan*

***Abstract*—Apache Spark is a parallel data processing framework that executes fast for iterative calculations and interactive processing, by caching intermediate data in memory with a lineage-based data recovery from faults. The Spark system can also manage data sets larger than memory capacity by placing some cache or all of them on disks on processing nodes. However, the disadvantage is potential performance degradation due to disk I/O and/or serialization. This study aims to clarify efficient/inefficient use of disks in intermediate data caching in Spark and also to improve the usability of disks for end users. In order to achieve the purpose, influence of disk use in data caching was firstly investigated in various aspects, such as caching options, data abstractions and storage devices. The results indicate that serialization cost is dominant rather than disk I/O in most cases. Secondly, a method of combined use of memory and disk was further evaluated under a high memory pressure. Then the method was improved to avoid an excessive re-caching problem, which achieved at most 20-30% reduction of total execution time under a high memory pressure and did not degrade the performance under a low memory pressure, in our experiment with 4 machine learning benchmarks. Finally, this paper summarizes important factors and potential improvements for efficiently using disks in data caching in Spark.**

***Keywords*-big data; Apache Spark; disk-based caching; performance analysis;**

I. Introduction

Apache Spark (Spark) [1] is an open source parallel data processing framework, attracting high attention in the field of big data analytics and artificial intelligence. While Spark basically holds data in memory during computation, disks on worker nodes are used for the shuffle operation and user's specified caching operation. For example, the user can specify use of disks for intermediate data caching instead of use of memory, or in addition to use of memory, for mainly handling larger data. However, the use of disks may negatively impact on performance of the Spark application and thus it is necessary to decide whether use it or not carefully. An obvious problem is that the choice of using disks is left to users and it is not easy for the users to appropriately judge it. A combined use of memory and disk which can handle larger data than memory-only choice, is expected to efficiently use both memory and disks inside of the Spark execution system though the actual performance and usability have not been clarified yet.

In order to solve these issues, we aimed to clarify efficient/inefficient use of disks in Spark and also to improve the performance and usability of disks for end users. In this study, we performed the following investigation and improvement by using machine learning benchmarks and our synthetic benchmark.

1) We investigated influence of disk use in intermediate data caching on execution performance of Spark applications, by a comparison of disk-based caching and other caching options. The result was confirmed in both conventional and newer data abstractions of Spark, each of which are RDD and DataFrame.
2) We compared serialization of RDD with encoding of DataFrame in intermediate data caching, from the view point of processing performance and space utilization in storing it.
3) We examined a speed-up of disk-based caching with faster storage devices, where a single thread or multiple threads performed the caching operation.
4) We evaluated a method of combined use of memory and disk, and then improved its performance by restricting excessive re-caching from disk to memory.

Based on these results, we summarized important factors to determine use of disks for intermediate data caching, including the performance difference between RDD and DataFrame, and potential improvements of efficiently using disks for caching in Spark.

II. Background

A. Apache Spark

Spark can perform distributed processing on a cluster consisting of nodes having both compute and storage functions. In addition to the advantages of Hadoop [2] and MapReduce [3], Spark basically holds intermediate data in memory, and has features that can efficiently execute computation requiring repetitive operations such as machine learning and data mining. The larger proportion of iterative operations to the total computational complexity, the greater the merit of using Spark.

Spark supports several programming languages such as Scala, Java, Python, etc. It is possible to perform batch execution of Spark applications using a resource management framework like YARN, or interactive execution using Spark-Shell. In addition, Spark works well with Hive, HBase, HDFS and others which form the Hadoop ecosystem.

B. RDD and DataFrame

Spark uses a read-only distributed data structure called RDD (Resilient Distributed Datasets). The RDD data is internally divided into partitions. Each partition is a unit of data processing and it can be processed in parallel on a distributed environment. Caching for iterative computation and fault tolerance in Spark are realized with the RDD. When multiple operations are applied to the same intermediate data, the cache is utilized to obtain the next result promptly. If the intermediate data is not cached or a failure occurs on any node holding the cache, Spark will reconstruct the data by referring "Lineage" which recorded the procedure of generating the data, and continue to execute computation. The reconstruction task instantiates only the partition of the lost cache.

The intermediate data to be cached is explicitly stored in memory or disk. When stored in disk, the data is serialized by some means. The default serialization method of RDD is the standard Java serialization. The Java serialization is flexible but often quite slow, resulting in a large serialization format for many classes. Other than that, the Kryo serialization can be selected. Kryo is very fast but classes to be serialized need to be registered in the users' programs.

DataFrame is a distributed data structure that has a concept of columns with rows, names and data types like the table. Unlike RDD, DataFrame allows the Spark system to perform optimization with the type information. DataFrame works well with Scala and Java which are often used for programming Spark applications. The DataFrame implementation benefits from the results of the Tungsten project [4], including memory optimization inside Spark at compile time. Mutual conversion between RDD and DataFrame is easy and migration from the RDD-based code to the DataFrame does not trouble users. In DataFrame, serialize/deserialize uses encoders which are different form either the standard Java serialization or Kryo. By default, the encoders are prepared for each case class that defines types such as Int, Long, String, Double, etc. and a proper encoders from run length encoder, dictionary encoder, etc. is chosen. This encoding works with a module called as Catalyst optimizer [5] that optimizes execution of SparkSQL [6].

C. Storage Options and Internal Behavior of Intermediate Data Caching

In Spark, intermediate data caching is executed by calling the *persist()* method for RDD and DataFrame with specifying a storage level. The storage level designates use of memory-only, use of disk-only, or use of both memory and disk, etc. In the case of RDD, the default is memory-only and in the case of DataFrame, the default is memory-and-disk.

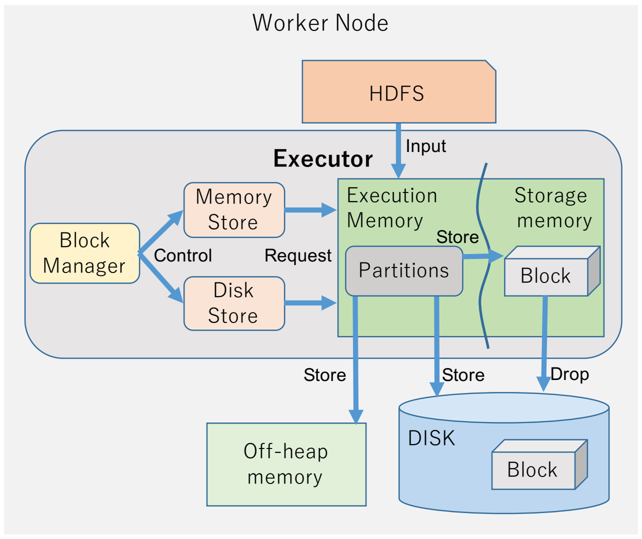

Figure 1. Internal processing in intermediate data caching

Figure 1 shows an internal structure of *Executor* running on worker nodes of Spark. Memory of *Executor* is dynamically divided into *ExecutionMemory* and *StorageMemory* by *MemoryManager*. While *ExecutionMemory* is used to process intermediate data, *StorageMemory* is managed by *BlockManager* and is used to store the intermediate data partitions in the form of *Block*. That is, *StorageMemory* is used when storing the intermediate data cache in memory.

The following describes major storage levels that can be specified for intermediate data caching and their behaviors.

[NONE (NOCACHE)] A method that do not cache intermediate data at all. Since the data is discarded instantly after use, regeneration of it is required when the same data is referenced again.

[MEMORY_ONLY] A method of holding an intermediate data cache in memory-only. When intermediate data caching is executed, *BlockManager* reserves a necessary amount of memory in *StorageMemory* through the interface of *MemoryStore* and saves each partition of the intermediate data in the form of *Block*. There is also a method called MEMORY_ONLY_SER which serializes and holds it in memory with consuming less memory space.

[OFF_HEAP] A method of holding an intermediate data cache in Off-Heap memory-only. When intermediate data caching is executed, *BlockManager* serializes the partitions in memory and stores them in Off-Heap memory through the interface of *MemoryStore*. This method has the advantage that influence of garbage collection is reduced.

[DISK_ONLY] A method of holding an intermediate data cache in disk-only. When intermediate data caching is

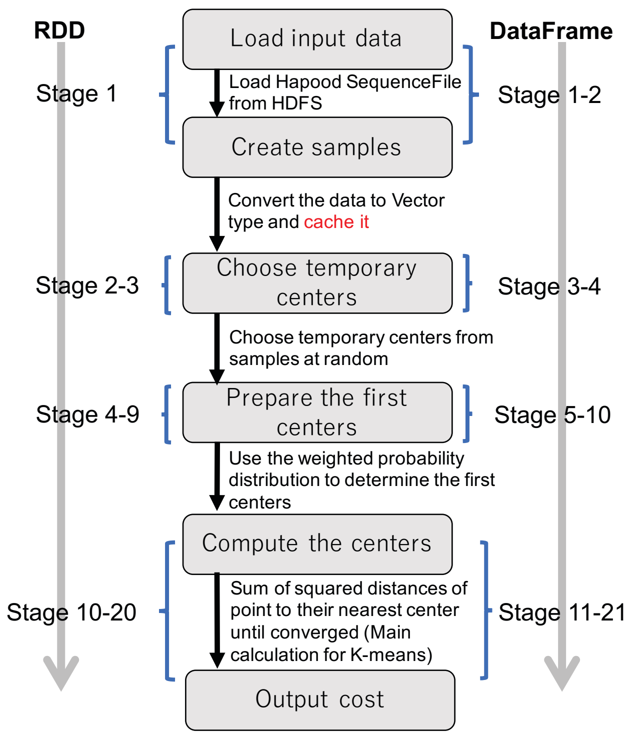
RDD
DataFrame
Load input data
Load Hapood SequenceFile from HDFS
Create samples
Convert the data to Vector type and cache it
Choose temporary centers
Choose temporary centers from samples at random
Prepare the first centers
Use the weighted probability distribution to determine the first centers
Compute the centers
Sum of squared distances of point to their nearest center until converged (Main calculation for K-means)
Output cost
Stage 1
Stage 2-3
Stage 4-9
Stage 10-20
Stage 1-2
Stage 3-4
Stage 5-10
Stage 11-21

Figure 2. A processing flow of the K-means benchmark

executed, *BlockManager* serializes the partitions in memory and stores it on a local disk through the *DiskStore* interface.

[MEMORY_AND_DISK] A method of holding an intermediate data cache by using memory and disk. This tries to use memory preferentially and save it as a block in *StorageMemory*. However, if it does not fit in the memory space, *BlockManager* will move some old blocks to a disk, in order to get the amount of memory required by the new *Blocks*. The moved blocks will be chosen based on the LRU (Least Recently Used) algorithm until the required space is available. When referring the intermediate data stored in the disk, *BlockManager* checks the necessary space in *StorageMemory* to expand the corresponding data, and if possible, it reads the blocks from the disk and puts them back to *StorageMemory*. The largest advantage of this method is that it does not require users to select memory or disk, and thus it is expected to use both efficiently by the Spark system.

III. Experiments

A. Methods and Environment

We investigated the execution performance of Spark applications when caching intermediate data using disks, in order to clarify efficient/inefficient use of disks. Firstly, three types of evaluation experiments were carried out:

Experiment 1: In order to investigate how much performance is degraded by using disks for caching intermediate data, the execution performance in disk-only use was compared with other storage levels. The performance was also confirmed with both RDD and DataFrame. The result is described in Section III-B.

Experiment 2: In order to investigate performance improvement of using DataFrame, serialization of RDD and encoding of DataFrame was compared. The result is described in Section III-C.

Experiment 3: In order to investigate whether there is an effect of improving performance by speeding up disk-only caching by using faster storage devices, we measured the time required for the RDD caching operation by using multiple storage devices, each of which has different I/O performance. The result is described in Section III-D.

Secondly, the use of both memory and disk was evaluated further to examine how Spark can efficiently use both:

Experiment 4: The Spark application was executed under circumstances where cache drops from memory to disk occurred frequently. Then the performance was compared with other storage options.

In the experiment 4, we found out unexpected performance degradation and therefore improved the caching algorithm in Spark. The details are described in Section III-E.

In the experiment 1, 2 and 4, a K-means program (DenseKMeans.scala) included in Spark Machine Learning Library (MLlib) [7] was used as a benchmark. Input data was generated using HiBench (6.0) [8] and stored in HDFS. The file format was SequenceFile of a key-value pair and the file size was 4 GB. A processing flow of the K-means is shown in Figure 2. Importantly, there is pre-processing before the main calculation: 1) Read input data from HDFS, 2) Extract only the values from the key-value pairs, 3) Make the extraction an array of the Double type, 4) Call the *Vector.dense()* method, and 5) Cache the Vector type as intermediate data. This Vector type data is used as input by the main calculation of K-means. The K-means parameters were set as $K = 10$, $numIterations = 5$ and $dimensions = 20$.

In the experiment 2 and 3, our synthetic benchmark program called as RDDTest was used. The RDDTest generates intermediate data of arbitrary size in the Spark worker and performs caching the data for the purpose of these experiments.

In the experiment 4, for investigating wider applicability, we used three other benchmarks which were Sparse Naive Bayes, Alternating Least Squares (ALS) and Logistic Regression (LR), in addition to K-means. They are all included in MLlib and input data was generated by using HiBench and stored in HDFS. The Sparse Naive Bayes parameters were set as $pages = 100,000$, $classes = 100$ and $ngrams = 2$. The ALS parameters were set as $user = 10,000$, $products = 10,000$, $sparsity = 0$, $implicitperfs = true$, $rank = 10$, $Lamdsa = 1.0$ and $kyro = false$. The LR parameters were set as $examples = 10,000$ and $features = 100,000$.

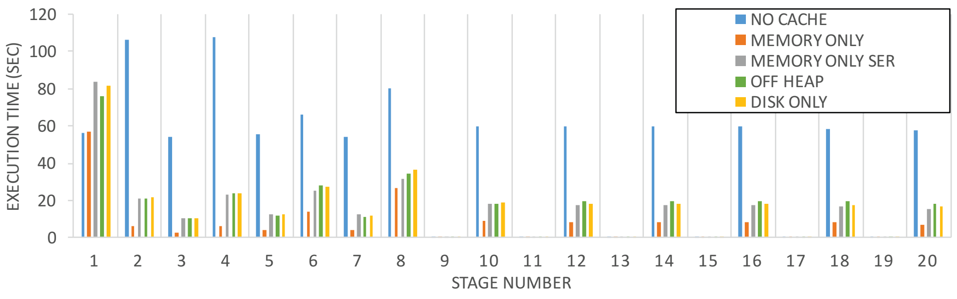

Figure 3. The impact on caching performance when using a disk (RDD)

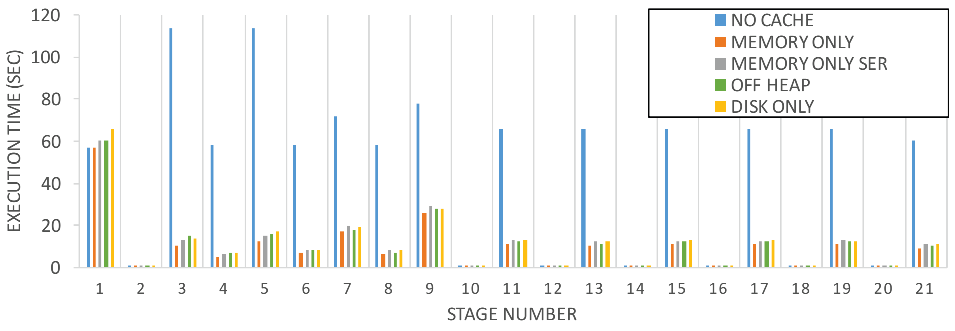

Figure 4. The impact on caching performance when using a disk (DataFrame)

Table I
MACHINE SPECIFICATION USED IN THE EXPERIMENTS

CPU	Intel Xeon CPU E5-2620v3 2.40GHz, 6 cores x2
Memory	128 GB
Network	10 Gbps (for HDFS connection)
NVMe-SSD	Intel SSD DC P3700
SSD	OCZ Vertex3 (240GB, SATA6G I/F)
HDD	Hitachi Travelstar 7K320 (SATA3G I/F)
OS	Ubuntu 14.04 (Kernel v.3.13)
File System	Ext4

We used the machine shown in Table I for all of our experiments. We ran each Spark benchmark program in a local mode. In practical use, Spark is executed in a distributed environment, but since this evaluation experiment focuses only on the performance of intermediate data caching of each worker node, we used just one node for simplifying the experiment environment. In the experiment 2, four types of storage devices, which are RAM disk, SSD with NVMe connection, SSD with SATA 6G connection and HDD with SATA 3G connection, were used. We used Spark v.2.1.0 that contains several achievements of the Tungsten project [9], with Scala v.2.10.6 and Java v.1.8.0_66.

B. Performance Impact of Intermediate Data Caching When Using Disks

In this experiment, the number of threads was set to 1 and the memory allocated to *Executor* was set to 60 GB. The K-means benchmark was executed with four kinds of storage levels with and without caching. Both RDD and DataFrame versions were examined for comparison, too.

Figure 3 shows the result for RDD and Figure 4 shows the result for DataFrame. In these figures, the execution time for every stage of the K-means benchmark was shown. When without caching, the stage 1 was completed in a shorter time than in the cases of caching. In the subsequent stages, however, it took longer time than the caching cases. Conversely, when caching was enabled, the stage 1 took more time but the subsequent stages were completed in a shorter time.

In the case of using RDD, the storage level MEMORY_ONLY showed better a result than other storage levels. The difference in total execution time was less than half. The results of MEMORY_ONLY_SER and OFF_HEAP were almost equivalent to DISK_ONLY, which indicates that serialization of the intermediate data was a bottleneck in caching.

In the case of using DataFrame, the storage level MEMORY_ONLY[1] showed a better result than other storage levels, too. However, since the DataFrame encoding was faster than the RDD serialization, the performance in other storage levels were not so much different from MEMORY_ONLY unlike the RDD result. This means that DataFrame is a better choice when using disk-based caching.

[1]The off-heap flag was false with a MEMORY_ONLY mode and true with an OFF_HEAP mode in all of our DataFrame experiments.

Table II
EXECUTION TIME AND CACHE SIZE (RDD VS DATAFRAME)

	RDD		DataFrame	
	Time(s)	Size(MB)	Time(s)	Size(MB)
NOCACHE	936	-	999	-
MEMORY_ONLY	170	4,578	203	4,653
MEMORY_ONLY_SER	321	3,830	232	4,653
OFF_HEAP	331	3,830	228	4,653
DISK_ONLY	335	3,830	241	4,653

C. A Comparison Between RDD and DataFrame Caching

Table II shows a comparison between RDD and DataFrame in execution time and cache size, for the experiments of Figure 3 and 4. Both of them showed that caching was very effective for execution performance. MEMORY_ONLY of RDD showed the best performance because intermediate data was directly stored in memory without serialization. DataFrame took more time than RDD because it did encoding and decoding to manage intermediate data even in the MEMORY_ONLY mode. MEMORY_ONLY_SER, OFF_HEAP and DISK_ONLY of DataFrame had better performance than those of RDD. The reason for this is considered that the DataFrame encoders processed intermediate data faster than the Java serialization though the encoded data size became larger.

In the above K-means benchmark and many other machine learning benchmarks, the type of data we cached is a vector of doubles, and in the experiment (RDDTest) shown in the next subsection, the type of data we cached is a string. In order to know performance difference for caching between the data types, we also compared RDD and DataFrame performance with the data types by using the RDDTest program which caches a similar amount of string or vector-of-doubles typed data. The result showed that caching string-typed data cost 3.418s with RDD and 4.989s with DataFrame. Caching vector-of-doubles data cost 2.057s with RDD and 3.753s with DataFrame. For the vector-of-doubles that would be used for our target applications, RDD showed better performance than DataFrame.

D. A Performance Comparison of Disk-based Caching with Different Storage Devices

In this experiment, RDD was chosen as an example to investigate caching performance with different storage devices when holding a RDD cache by using a disk. We conducted this experiment using the RDDTest program and set the cached RDD data size to 1,000 MB. The number of threads was set to 1.

Figure 5 shows caching performance measured by RDDTest and raw performance of storage devices measured by the Fio [10] benchmark. From this result, it can be seen that the performance difference among the storage devices did not much affect on the performance difference of generating the RDD cache. Serialization process seems to be a bottleneck when the storage device was fast like RAM disk or NVMe SSD, and the performance of the devices was not fully utilized. Even if the storage device was at a low speed like HDD, the influence on the caching performance was small due to the effect of the buffer cache of the operating system.

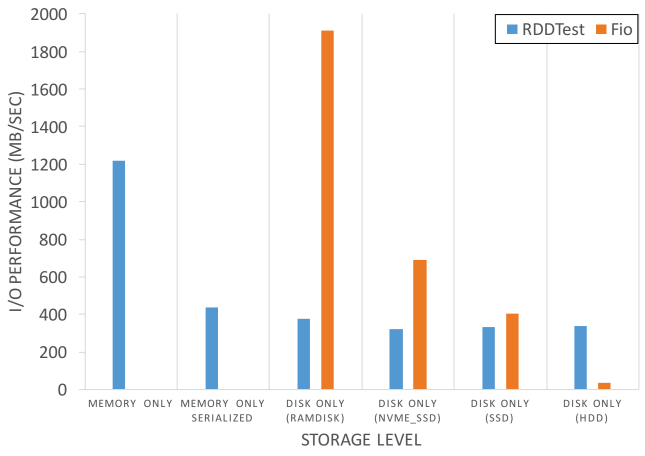

Figure 5. A single caching performance with different storage devices

Figure 6 shows average I/O performance when multiple threads did their own caching operations. The same RDDTest as the previous experiment was executed by each thread and the number of threads was changed to 1, 4, 8, 12. The cached RDD data size of each thread was also changed to 500 MB, 1,000 MB, 1,500 MB, 2,000 MB. While the cache size and the number of threads increased, the I/O performance of the RDD caching was maintained except the HDD case. In the case of HDD, when the number of threads was 8 and the RDD size was 2,000 MB, the I/O performance of the RDD caching was degraded. These results indicate that disk performance becomes important only when RDD caching was executed at the same time by many tasks so that the effect of the buffer cache of the operating system was fairly reduced.

E. Evaluation and Improvment of A Memory and Disk Mode

1) Preliminary Evaluation: In order to investigate an effect of the memory and disk mode, we executed the K-means benchmark with MEMORY_AND_DISK, under the situation where the amount of memory allocated to *Executor*

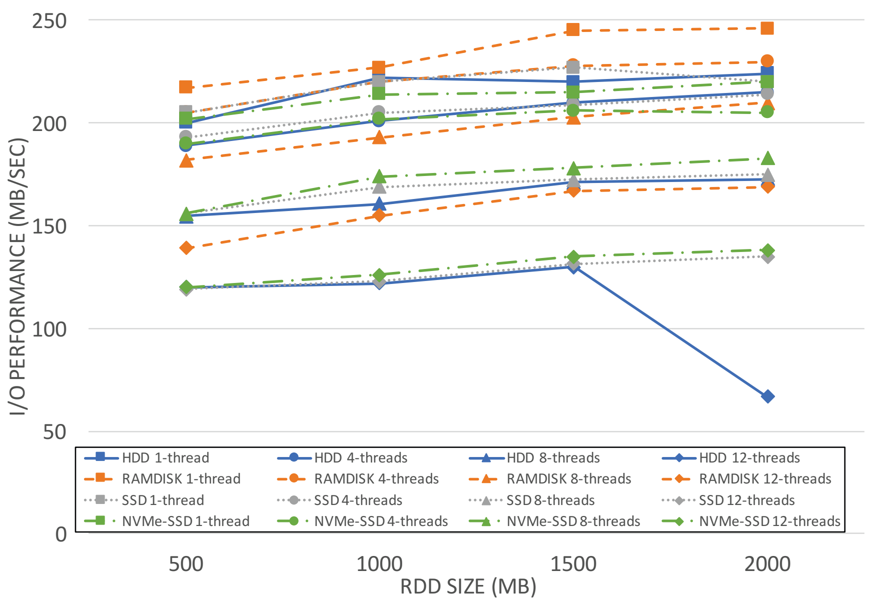

Figure 6. Scalability of caching performance with different storage devices

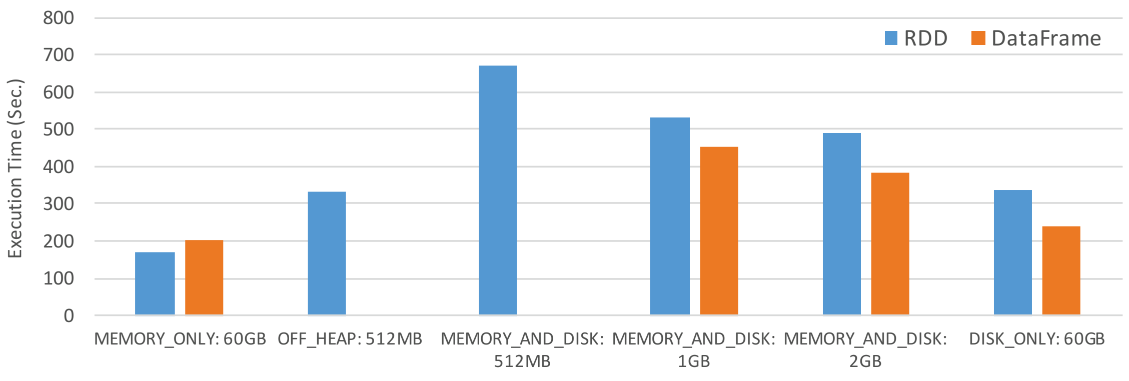

Figure 7. Evaluation of using memory and disk

was severely restricted with three patterns: 512 MB, 1 GB, 2 GB. Then we compared the result with the case of MEMORY_ONLY, OFF_HEAP and DISK_ONLY. In the cases of MEMORY_ONLY and DISK_ONLY, the amount of memory allocated was set to 60 GB. We tried both RDD and DataFrame versions but when the allocation was set to 512 MB with DataFrame, OOM (Out of memory) occurred and it was impossible to obtain the result.

The results of both RDD and DataFrame versions are shown in Figure 7. In the cases of MEMORY_AND_DISK: 512M for RDD and MEMORY_AND_DISK: 1G for DataFrame, both took more than twice the processing time compared with DISK_ONLY: 60GB. The execution performance of storage level MEMORY_AND_DISK was very poor in both.

As shown in Table III, the blocks' movement between memory and disk was examined for finding the cause of the

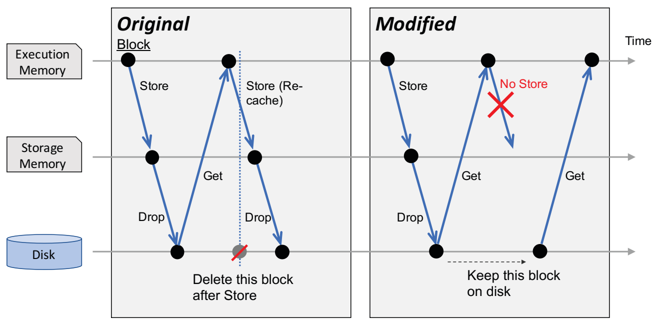

Figure 8. Suppression of re-caching in the improved version of Spark

poor performance. In the case of MEMORY_AND_DISK: 512M for RDD, the number of dropped blocks was higher but the dropped size was smaller compared with other cases. However, we could see that the occurrence rate of garbage

Table III
FREQUENCY OF CACHE DROPS AND THE AMOUNT OF THE DROPPED CACHE

	RDD		DataFrame	
	# of Blocks	Size(MB)	# of Blocks	Size(MB)
MEMORY_AND_DISK_512MB	600	2,577	-	-
MEMORY_AND_DISK_1GB	746	26,793	701	27,234
MEMORY_AND_DISK_2GB	715	20,936	709	27,316
OFF_HEAP_512MB	305	1,332	-	-

Table IV
THE NUMBER OF DROP OCCURRENCES, TOTAL SIZE AND EXECUTION TIME (RDD)

	Original			Modified		
	# of Blocks	Size(MB)	Time(s)	# of Blocks	Size(MB)	Time(s)
MEMORY_AND_DISK_512MB	600	2,577	670	133	610	422
MEMORY_AND_DISK_1G	746	26,793	533	33	486	331
MEMORY_AND_DISK_2G	715	20,936	489	58	473	318
MEMORY_AND_DISK_60G	0	0	185	0	0	183

Table V
THE NUMBER OF DROP OCCURRENCES, TOTAL SIZE AND EXECUTION TIME (DATAFRAME)

	Original			Modified		
	# of Blocks	Size(MB)	Time(s)	# of Blocks	Size(MB)	Time(s)
MEMORY_AND_DISK_512MB	-	-	-	-	-	-
MEMORY_AND_DISK_1G	701	27,234	452	50	481	337
MEMORY_AND_DISK_2G	709	27,316	383	54	504	265
MEMORY_AND_DISK_60G	0	0	224	0	0	223

collection was much higher and thus the garbage collection seemed to be the main cause of the performance degradation in this case.

In the cases of MEMORY_AND_DISK: 1G and 2G, many cache drops from *MemoryStore* occurred and their sizes were also very large. The reason for many drops was confirmed that the dropped blocks were returned to *MemoryStore* and then dropped repeatedly.

OFF_HEAP: 60GB for RDD had better performance than MEMORY_AND_DISK: 512M because of lower occurrence of cache drops. However, there was an overhead of serialization and performance was close to DISK_ONLY: 512MB.

2) Improvement and Evaluation: In order to solve the excessive re-caching problem described in Section III-E1, we modified the internal behavior of Spark. Figure 8 shows a difference between the original Spark and the modified Spark. In the original Spark, after data is read, it is preferentially saved in memory as a block. If the new block does not fit in the memory space, Spark will drop some old blocks to a disk to save the new block in memory. If a future task requires the dropped blocks on the disk, Spark will get the blocks from the disk before processing. After the task is completed, the blocks will be taken from the disk, saved in memory again and deleted on the disk. When a new block comes into memory and there is no space, the same blocks might be dropped to the disk. Thus the problem is that it takes a large cost to repeatedly move the same blocks between the memory and the disk.

In the modified spark, after the task is completed, the blocks taken from the disk will not be moved back into memory again and kept on the disk. Reading the blocks on the disk is slower than reading from the memory, but it will reduce the cost of repeated caching and dropping. This re-caching suppression does not cause cache inconsistency because the cached data in Spark is read-only and will never be modified.

Table IV and Table V show a comparison result of our improved version of Spark. Each table shows the number of dropped blocks, total drop size and execution time in both original and modified versions. It can be seen that the number of drop occurrences and the total size can be greatly reduced by suppression of re-caching. The execution time for RDD was reduced to about 60% by the modification. Likewise the DataFrame was reduced to 70%, which means that performance has been improved by our suppression of re-caching. In the case of MEMORY_AND_DISK: 60G, the modified Spark keeps the performance of the original Spark. This shows that even at low memory pressures, suppression of re-caching does not affect Spark performance.

In order to investigate wider applicability of the proposed method, we examined three additional benchmarks with the RDD. Table VI shows a comparison result among the benchmarks: Sparse Naive Bayes, Alternating Least Squares(ALS) and Logistic Regression (LR). Note that the results of ALS_512MB and LR_512MB were not obtained because out of memory (OOM)) occurred. In case of SparseNaiveBayes_512MB, duo to the size of the dropped blocks is not large, even if the block re-caching was sup-

Table VI
DROP BLOCKS SIZE AND EXECUTION TIME (RDD)

	Sparse Naive Bayes				Alternating Least Squares				Logistic Regression			
	Original		Modified		Original		Modified		Original		Modified	
	Size (MB)	Time (s)	Size (MB)	Time (s)	Size	Time	Size	Time	Size	Time	Size	Time
512 MB	988	139	203	136	-	-	-	-	-	-	-	-
2 GB	0	79	0	78	31,787	980	2,720	897	405,130	1,514	5,536	1,194
60 GB	0	82	0	81	0	662	0	663	0	860	0	863

pressed, performance improvement is not obvious. In case of ALS_2GB, the dropped size between the original and modified Spark is 10 times difference. The dropped size was greatly reduced but the re-caching overhead was a low proportion of the total execution time and thus performance improvement was not so obvious. In case of LR_2GB, the dropped size reached 400GB in the original Spark while the size reduced to 5GB in the modified Spark. Then the total execution time was reduced by 20%. According to these results, all of the benchmarks performance has been improved under a high memory pressure. Besides under a low memory pressure (memory=60G), all of the benchmarks showed almost the same performance as original Spark which means the proposed method does not cause a negative performance impact.

IV. DISCUSSION

In this section, we summarize important factors to determine use of disks for intermediate data caching, which are learned from the results of Section III, and potential improvements of efficiently using disks for caching in Spark.

- Performance degradation occurs when using disk in intermediate data caching, but the performance of disk is less likely to be a bottleneck due to the buffer cache of the operating system. The serialization seems to be the bottleneck in most cases. When using RDD, the performance degradation is more obvious than DataFrame, due to the inefficiency of the standard Java serialization.
- When using memory-only for intermediate data caching, serialization process is not executed in the RDD case but encoding is executed in the DataFrame case. Therefore the RDD is faster than the DataFrame. On the contrary, encoding of DataFrame is faster than serialization of RDD when storing the cache on disk or off-heap memory.
- It is conceivable that influence of disk performance appears in the execution performance of the application only when many tasks are executed in parallel on each worker node and the tasks simultaneously perform a caching operation. Thus, in many situations, use of HDD which has larger capacity is superior in cost performance than other types of disks. Besides, improvement of serialization performance in Spark is highly expected.
- In the same environment, the benchmark with DataFrame failed because of OOM. This indicates that DataFrame requires more execution memory than RDD, for encoding and decoding operations.
- Although the memory and disk mode is convenient for users because the Spark system selectively uses both, actual performance was terrible and the execution performance was worse than caching with disk-only, in the situation where memory capacity is largely insufficient. On the other hand, the problems can be solved by suppression of re-caching. However, because blocks dropped from memory will never return to memory in our improvement, the remaining memory might not be effectively used after the memory pressure becomes low. There would be an opportunity to implement more intelligent algorithm which returns the blocks to memory when more free memory get available during computation.
- While use of off-heap memory has the advantage of reducing the influence of garbage collection (GC), it was not able to confirm the superiority to use of disk-only, due to the overhead of serialization. On the other hand, influence of the GC algorithms and their parameters have not been investigated in this study. We would like to tackle the issue in the future and clarify whether the results shown in this paper would be true when using other GC algorithms.

V. RELATED WORK

Project Tungsten [4] has attempted to improve efficiency of memory and CPU use in Spark. The project achieved performance improvement of Spark programs and SparkSQL including intermediate data caching, by eliminating the overhead of the JVM object model and garbage collection, implementing an off-heap caching mode, speeding-up serialization, etc. In order to select RDD data to be dropped when available memory is insufficient, a weight replacement algorithm that takes into account the cost of reconstructing the RDD partitions was studied [11]. Neutrino [12] implements fine-grained memory caching of the RDD partitions and a conversion mechanism to one cache level (e.g., deserialized) to another (e.g., serialized), for achieving efficient memory utilization. Luna Xu et al. designed a dynamic memory manager for in-memory data analytics to dynamically tunes execution/storage memory partitions at runtime, depending on workloads and in-memory data cache demands [13]. However, their studies are based on memory-only caching with Spark version 1.5 that implements old memory management and their focus is different from our aim at clarifying and improving performance of intermediate data caching by efficiently using disks.

J. Shi et al. profiled task execution of Spark to analyze the performance, and investigated an effect of intermediate data caching, performance influence of the storage level, and bottlenecks of execution [14]. Their investigation using the K-means program is similar to our approach. However, their purpose was to clarify reasons of the performance difference between Hadoop MapReduce and Spark, and to analyze fault tolerance from the system architecture and implementation. S. Canon et al. pointed out a scalability issue for metadata access when Spark uses a shared file system (i.e., Lustre) instead of a local disk on the worker nodes, in the large HPC system [15]. The issue could be solved by a user-level file pooling that caches open files as well as use of a local disk. M. Zaharia et al. proposed a memory-based distributed file system called Tachyon [16], [17], which achieves reliable read and write at memory speed, by avoiding synchronous data replication on writes. The file transfer speed of Tachyon is much faster than HDFS, which makes Spark applications have better performance.

VI. Conclusion and Future Work

In this paper, we clarified performance influence of disk-based caching in Spark including bottlenecks, important factors to decide use of disk and how to use disk more efficiently. We also investigated the caching performance in a combined use of memory and disk and then improved it by modifying the re-caching algorithm. The results were confirmed in both RDD and DataFrame as data abstraction in Spark.

As future work, first we would like to confirm whether these results could be applied not only to 4 benchmarks with MLlib but also to other machine learning programs or different types of applications other than MLlib programs, and then improve our approach for the re-caching problem. In order to generalize the guidelines obtained this time, further evaluation with larger and various data sets on a large cluster environment would be needed and think about a method to make it better to cooperate with memory management of Spark.

Acknowledgment

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). This work was partly supported by JSPS KAKENHI Grant Numbers JP16K00116, JP16K21675.

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012, pp. 15–28.

[2] Apache Software Foundation, "Apache Hadoop," 2006. [Online]. Available: http://hadoop.apache.org/

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004, pp. 1–13.

[4] R. Xin and J. Rosen, "Project Tungsten: Bringing Apache Spark Closer to Bare Metal," 2015. [Online]. Available: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[5] M. Armbrust, Y. Huai, C. Liang, R. Xin, and M. Zaharia, "Deep Dive into Spark SQL's Catalyst Optimizer," 2015. [Online]. Available: https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *SIGMOD'15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1383–1394.

[7] X. Meng and et al., "MLlib: Machine Learning in Apache Spark," *Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[8] Intel-hadoop, "HiBench Suite," 2017. [Online]. Available: https://github.com/intel-hadoop/HiBench

[9] Apache Software Foundation, "Project Tungsten (SPARK-7075)," 2015. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-7075

[10] J. Axboe, "Fio (Flexible I/O Tester)," 2006. [Online]. Available: https://github.com/axboe/fio

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

[13] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE: Dynamic Memory Management forIn-memory Data Analytic Platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium*, 2016, pp. 383–392.

[14] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics," in *Proceedings of the VLDB Endowment - Proceedings of the 41st International Conference on Very Large Data Bases, Kohala Coast, Hawaii*, vol. 8, no. 13, 2015, pp. 2110–2121.

[15] N. Chaimov, A. Malony, S. C. K. Z.Ibrahim, and C. I. J. Srinivasan, "Scaling Spark on HPC Systems," in *HPDC'16 Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 97–110.

[16] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica, "Tachyon: Memory Throughput I/O for Cluster Computing Frameworks," in *Large-Scale Distributed Systems and Middleware (LADIS 2013)*, 2013. [Online]. Available: http://www.cs.berkeley.edu/ haoyuan/papers/2013_ladis_tachyon.pdf

[17] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and IonStoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *Proceeding SOCC'14 Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–15.